

# Gravity Shift Connect Four

How the program works, part by part, and where each part appears in the proposal.  
Code version of 19 October 2026 · Proposal export of 19 October 2026 (19 pages)

## 1. Start here

### 1.1 The game in brief

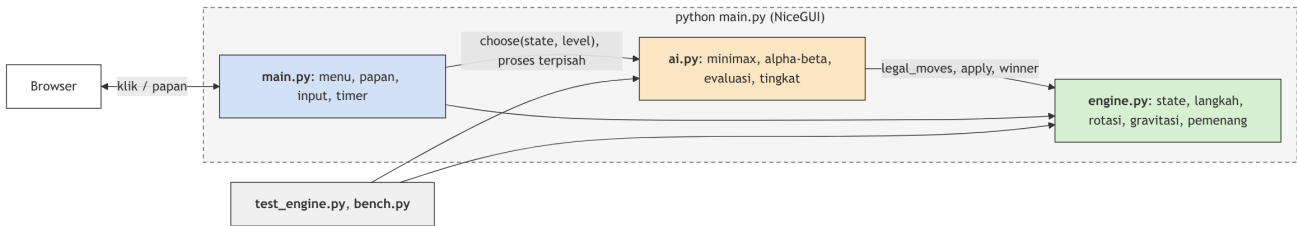
- Board 6 × 6. The human (red) moves first, then the agent (yellow), in turns.
- A move is either a drop into a column that is not full, or a turn of the whole board 90° left or right. After a turn every disc falls to the new bottom.
- Each player has 3 turns of the board per game.
- Four in a row in any direction wins. If a turn of the board gives four to only one player, that player wins, whoever turned it. If it gives four to both, the game is a draw. A full board with no four is a draw.
- The human has 2 minutes for the whole game. The agent has 1 second per move. There are five levels, from Beginner to Expert.

Proposal: **1.1 Latar Belakang**, **1.2 Rumusan Masalah**, **2.1 Aturan Permainan**

### 1.2 The files

| File           | Job                                                          | Uses                          |
|----------------|--------------------------------------------------------------|-------------------------------|
| engine.py      | the rules: state, legal moves, drops, turns, gravity, winner | nothing                       |
| ai.py          | the agent: minimax with alpha-beta, evaluation, levels       | engine.py                     |
| main.py        | the web pages, the clock, input and animation                | engine.py , ai.py , style.css |
| style.css      | colours, sizes, light and dark theme, animation curves       | nothing                       |
| test_engine.py | 22 checks on the rules and the agent                         | engine.py , ai.py             |

This document follows the same order. Each Python file only uses files explained before it, so nothing is used before it is explained.



Proposal Gambar 2. `main.py` knows the other two files. `ai.py` knows only `engine.py`. `engine.py` knows nothing else.

## 1.3 Running it

```
pip install -r requirements.txt      # NiceGUI 3.18.0, the only library
python main.py                      # open http://localhost:3117
python test_engine.py               # one line per test, then "22 tests passed"
```

## 1.4 How to read the code

### Proposal words and code names.

| Proposal                 | Code                                                   | Note                                                          |
|--------------------------|--------------------------------------------------------|---------------------------------------------------------------|
| bidak                    | disc, a value in <code>cells</code>                    | 1 human (red), 2 agent (yellow), 0 empty                      |
| langkah                  | <code>move</code>                                      | a column 0-5, or "L" / "R"                                    |
| jatah rotasi             | <code>rotations_left</code>                            |                                                               |
| state permainan          | <code>state</code>                                     |                                                               |
| giliran                  | <code>to_move</code>                                   |                                                               |
| window (4 sel)           | line, <code>engine.LINES</code>                        | The code says "line", the proposal says "window". Same thing. |
| kedalaman                | <code>depth</code>                                     | counted in moves, one per player turn                         |
| node                     | node, <code>Search.nodes</code>                        | a position the search visits                                  |
| nilai minimax v(m)       | <code>values[move]</code>                              |                                                               |
| softmax temperature<br>T | <code>temperature</code>                               |                                                               |
| tingkat kesulitan        | <code>level</code> , <code>LEVELS</code>               |                                                               |
| fungsi evaluasi          | <code>evaluate</code>                                  |                                                               |
| terminal state           | <code>winner(state)</code> is not<br><code>None</code> |                                                               |

### The state layout.

```
state = (cells, to_move, rotations_left)
state[0] # cells: 36 values
state[1] # whose turn: 1 or 2
state[2] # (human's rotations left, agent's rotations left)
```

- Cell (row, col) is `cells[row * 6 + col]`, which is what `index(row, col)` computes. Row 0 is the top of the screen and row 5 the bottom. Gravity always pulls toward row 5.
- One player's rotations: `rotations_left[player - 1]`. Players are numbered 1 and 2, and a tuple counts from 0.
- `move` in `engine.ROTATIONS` is true for a rotation. Any other move is a column number.

**Whose point of view a number has.** Every value in `ai.py` is from the point of view of me, the player the search chooses for. In the game that is always the agent. Positive is good for the agent and negative is good for the human. `moves_played` counts moves from the position

being searched, not from the start of the game, so `1000 - moves_played` makes a nearer win worth more.

**What `None` means.** It always means “nothing yet” or “nothing here”.

| Expression                                | Meaning when it is <code>None</code> |
|-------------------------------------------|--------------------------------------|
| <code>winner(state)</code>                | the game goes on                     |
| <code>lowest_empty_row(cells, col)</code> | the column is full                   |
| <code>leaf_or_none(...)</code>            | keep searching below this position   |
| <code>deadline</code>                     | no time limit                        |
| <code>Game.result</code>                  | the game has not ended               |

**Each layer uses only the ones below it.**

```
main.py → uses ai.py and engine.py
ai.py   → uses engine.py
engine.py → uses nothing
```

Reading `engine.py` needs nothing else. Reading `ai.py` needs only the five engine names it calls: `legal_moves`, `apply`, `winner`, `other` and `LINES`.

**Reading the NiceGUI code in `main.py`.**

| You see                                                                      | It means                                                                                                                          |
|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <code>with ui.element("div").classes("board"):</code>                        | Everything indented under it goes inside that element, like nested HTML tags.                                                     |
| <code>.classes("falling"),</code><br><code>.classes(remove="falling")</code> | adds or removes a CSS class, which starts or stops an animation in <code>style.css</code>                                         |
| <code>.props('aria-label="Kolom 1"')</code>                                  | sets an HTML attribute                                                                                                            |
| <code>lambda col=col: self.on_human_move(col)</code>                         | <code>col=col</code> stores the current column number. Without it, all six buttons would use the last value of <code>col</code> . |
| <code>async def, await asyncio.sleep(0.38)</code>                            | the game waits here while the animation plays, without freezing other players' tabs                                               |
| <code>await run.cpu_bound(ai.choose, ...)</code>                             | runs the agent in another process and waits for its answer                                                                        |
| <code>ui.timer(0.2, self.update_clock)</code>                                | calls <code>update_clock</code> every 0.2 seconds                                                                                 |

**Names say what they hold.**

- Booleans read as yes/no questions: `waiting_for_human`, `search_rotations`, `exact`.
- Functions say what they return or do: `legal_moves`, `lowest_empty_row`, `fall_rows_after_rotation`, `start_clock`.
- Constants are in `UPPER_CASE` at the top of each file. Every number the proposal names is one of them:

| Proposal | Code                                         |
|----------|----------------------------------------------|
| 1 detik  | <code>ai.TIME_LIMIT = 1.0</code>             |
| 1000     | <code>ai.WIN = 1000</code>                   |
| 2 menit  | <code>main.HUMAN_TIME = 120</code>           |
| 3 rotasi | <code>engine.ROTATIONS_PER_PLAYER = 3</code> |

### Checking a proposal claim yourself.

1. Find the proposal word in the first table above.
2. Find the function with that name. Section 7.1 lists them per proposal section.
3. Find the test whose name states the claim, for example  
`test_rotation_can_make_the_other_player_win`.
4. Run `python test_engine.py`. If the claim and the code disagree, that test fails.

## 2. The game state and the rules: `engine.py`

Proposal: **2.1 Aturan Permainan**, Gambar 1; **Arsitektur**, paragraph “State permainan akan disimpan sebagai tuple...”

### 2.1 The state is one tuple

```
state = (cells, to_move, rotations_left)
cells      36 values, row by row from the top-left: 0 empty, 1 human (red), 2 agent
(yellow)
to_move    1 or 2
rotations_left (rotations left for the human, rotations left for the agent), starts at (3,
3)
```

A tuple cannot be changed. Every move builds a new state with `apply`, so the search can try a move and simply drop the result. No undo step is needed. A tuple can also be sent to another process as it is, which `main.py` relies on.

Cell `(row, col)` sits at position `row × 6 + col` (`index`, line 16). Row 0 is the top of the screen and row 5 the bottom. Gravity always pulls toward row 5. A rotation moves the cells themselves, so the program never stores which way gravity points.

A move is a column number 0 to 5, or “L” / “R” for a turn to the left (counter-clockwise) or right (clockwise).

### 2.2 The 54 lines (`_all_lines`, line 20)

The function starts at every cell and walks 4 cells in each of 4 directions: right, down, down-right and down-left. A walk that stays on the board is a line. The count:

| Direction           | Start positions             | Lines     |
|---------------------|-----------------------------|-----------|
| Horizontal          | 6 rows × 3 starting columns | 18        |
| Vertical            | 6 columns × 3 starting rows | 18        |
| Diagonal down-right | 3 × 3                       | 9         |
| Diagonal down-left  | 3 × 3                       | 9         |
| <b>Total</b>        |                             | <b>54</b> |

The list is built once when the module loads and is used by both the win check and the evaluation function. The test `test_there_are_54_lines` checks the count.

### 2.3 Rotation (`_rotation_map`, `rotate`, `fall`)

A rotation is a fixed rearrangement of 36 positions, so the program computes it once. For each cell after the turn, `ROTATION_SOURCE` stores which cell it comes from:

Right (clockwise): new  $(r, c) \leftarrow \text{old } (5 - c, r)$

Left (counter-clockwise): new  $(r, c) \leftarrow \text{old } (c, 5 - r)$

Check with the right turn: the old bottom row is row 5. A cell of it, old  $(5, k)$ , lands where  $5 - c = 5$  and  $r = k$ , so at new  $(k, 0)$ . The bottom row becomes the left column, which is what a clockwise turn does. `test_rotate_right_turns_the_bottom_row_into_the_left_column` checks it.

After the turn, `fall` drops every disc to the bottom of its column and keeps the order of discs in that column. `apply` does both in one line: `fall(rotate(cells, move))`.



Left: before. Middle: after `rotate(..., "R")`, before gravity. Right: after `fall`. The bottom row X O became the left column read top to bottom, then fell.

## 2.4 Legal moves and applying a move

- `legal_moves` (line 84): every column whose top cell is empty, then "L" and "R" if the player to move has a rotation left. A rotation is legal even when it changes nothing, as on the empty board (proposal: "Rotasi tetap sah walaupun susunan bidak tidak berubah").
- `apply` (line 93): for a drop, puts the player's number at `lowest_empty_row`. For a rotation, turns and drops, then takes one from that player's count. Either way the turn passes to the other player. `apply` assumes the move is legal. The interface checks `legal_moves` first.

## 2.5 Who won (`four_in_a_row`, `winner`)

`four_in_a_row` returns the set of players who own at least one full line. `winner` then decides:

| Situation                   | Result                     | Proposal rule                                                                         |
|-----------------------------|----------------------------|---------------------------------------------------------------------------------------|
| Both players have four      | DRAW                       | "Jika kedua pemain sama-sama memiliki empat bidak berderet, permainan berakhir seri." |
| One player has four         | that player, whoever moved | "...pemain tersebut menang, walaupun rotasi dilakukan oleh lawannya."                 |
| Nobody has four, board full | DRAW                       | "Permainan juga seri jika papan penuh tanpa pemenang."                                |
| Otherwise                   | None, the game goes on     |                                                                                       |

A drop adds one disc of the mover's colour, so it can only complete a line for the mover. Only a rotation can give four to the opponent or to both. The two tests with real positions (`test_rotation_can_make_the_other_player_win`, `test_rotation_that_gives_both_players_four_is_a_draw`) show both cases.

The game always ends. Each drop fills one of 36 cells and there are at most 6 rotations, so a game has at most 42 moves (proposal: "paling lama 42 langkah").

---

### 3. The agent: `ai.py`

Proposal: **2.1 Algoritma, Kompleksitas Waktu dan Skalabilitas** (Tabel 1), **Fungsi Evaluasi** (Tabel 2), **Tingkat Kesulitan** (Tabel 3)

#### 3.1 Values of finished games ( `final_value` , line 56)

The agent looks ahead through possible moves and has to score every position where it stops looking. There are two kinds. A finished game gets an exact value:

$$\text{win} = 1000 - n \quad \text{loss} = -1000 + n \quad \text{draw} = 0$$

Here  $n$  is the number of moves from the current position to the end (`moves_played`). A win in 3 moves scores 997 and a win in 5 scores 995, so the agent takes the faster win. A loss in 7 scores  $-993$  and a loss in 3 scores  $-997$ , so a losing agent delays the loss.

#### 3.2 Evaluation function ( `evaluate` , line 46)

The second kind is a game that is not finished when the search reaches its depth limit. The board then gets a heuristic score from the agent's point of view. For each of the 54 lines it counts the agent's discs  $a$  and the opponent's discs  $h$  and looks up `LINE_SCORE[(a, h)]`:

| (agent, opponent) in the line                       | Score |
|-----------------------------------------------------|-------|
| (3, 0): three of mine, one empty                    | +5    |
| (2, 0): two of mine, two empty                      | +2    |
| (0, 2): two of theirs, two empty                    | -2    |
| (0, 3): three of theirs, one empty                  | -5    |
| Anything else, including any line with both colours | 0     |

A line with both colours can never become four of either, so it scores 0. A full line of four never reaches `evaluate`, because `leaf_or_none` checks `winner` first. Each line scores at most 5 in size, so:

$$|\text{evaluate}| \leq 54 \times 5 = 270 < 1000 - 42$$

The smallest possible win value is  $1000 - 42 = 958$ , so any found win beats any unfinished board. `test_evaluation_stays_far_below_a_win` checks the bound on 200 random boards.

### 3.3 Minimax with alpha-beta ( `Search.alphabeta`, line 96)

```
def alphabeta(self, state, depth, alpha, beta, moves_played):
    leaf = self.leaf_or_none(state, depth, moves_played)    # finished game or depth 0
    if leaf is not None:
        return leaf
    agent_to_move = state[1] == self.me
    for move in self.moves(state):
        value = self.alphabeta(engine.apply(state, move), depth - 1, alpha, beta,
moves_played + 1)
        if agent_to_move:
            alpha = max(alpha, value)    # best the agent is sure of
        else:
            beta = min(beta, value)    # best the opponent is sure of
        if alpha >= beta:
            break    # the rest of this node cannot change the answer
    return alpha if agent_to_move else beta
```

Minimax: on the agent's turn take the highest child value. On the human's turn take the lowest, because the human is assumed to play the reply that is worst for the agent. One function handles both by checking whose turn it is, so there is no separate max and min function.

Alpha-beta, with a small example. The agent has moves A and B. A's two replies give 3 and 5, so A is worth 3 and alpha becomes 3. B's first reply gives 2. B is now worth at most 2, since the human picks the lowest. That is already worse than A, so B's other replies are skipped. Minimax and alpha-beta both choose A, and alpha-beta looked at fewer positions.

This form is called fail-hard: a cut-off node returns the bound (alpha or beta), not its true value. The root value is still exact, which is what the proposal claims ("hasil keputusan sama dengan minimax biasa"). `test_alphabeta_gives_the_same_value_as_minimax` compares it with the plain `minimax` (line 113) on 30 positions, with and without rotations.

Measured on 10 mid-game positions at depth 4 with rotations: alpha-beta visited 4,987 positions and plain minimax 39,238, about 8 times fewer.

### 3.4 Move order ( `MOVE_ORDER`, `Search.moves`, line 73)

```
MOVE_ORDER = [2, 3, 1, 4, 0, 5, "L", "R"]
```

The centre columns come first, because a centre cell belongs to more lines: columns 2 and 3 each touch 39 of the 54 lines, columns 0 and 5 only 15. Alpha-beta prunes more when good moves come first. `moves` filters this list down to the legal moves, removes rotations when the level has `search_rotations=False`, and can put one move ( `first` ) at the front.

### 3.5 Iterative deepening and the time limit (`decide`, line 155)

```
for depth in range(1, level.depth + 1):
    search.deadline = None if depth == 1 else deadline
    first = best_move(values) if values else None
    try:
        values = search.root_values(state, depth, exact, first)
    except OutOfTime:
        break
    depth_done = depth
```

- The search runs at depth 1, then 2, then 3, up to the level's depth.
- Every position visited calls `leaf_or_none`, which raises `OutOfTime` once `time.monotonic()` passes the deadline. The unfinished depth is thrown away and `values` still holds the last finished depth.
- Depth 1 runs with no deadline, so there is always a move (proposal: “Kedalaman 1 selalu diselesaikan tanpa batas waktu”). At most 8 positions, so it takes about a millisecond.
- `exact` is explained in 3.7. The best move of the last depth is searched first in the next depth (`first`). This only reorders the root. Inside the tree the order is `MOVE_ORDER`.
- `time_limit=None` removes the deadline, so a level always reaches its full depth. The experiments use this so results can be repeated (proposal 2.2: “agen dijalankan dengan kedalaman tetap tanpa batas waktu”).

### 3.6 Levels (`Level`, `LEVELS`, line 27)

| Level    | depth | search_rotations | temperature |
|----------|-------|------------------|-------------|
| Beginner | 1     | False            | 200         |
| Easy     | 1     | False            | 2           |
| Medium   | 2     | False            | 5           |
| Hard     | 4     | True             | 0           |
| Expert   | 6     | True             | 0           |

These are exactly the values of proposal Tabel 3. They are fixed numbers in the source. Nothing in the program changes them while it runs, so no learning takes place.

What each level can see:

- **Beginner** looks at its own move only.  $T = 200$  makes all ordinary moves almost equally likely, so it plays close to random. It still takes a win in one about 97% of the time, because a win scores 999 and an ordinary move about 0. It never blocks, because it does not look at the human's reply.
- **Easy** also looks at its own move only, but  $T = 2$  makes it prefer the move that builds the most lines. It never blocks either.
- **Medium** looks at its move and the human's reply, so it blocks every threat to win in one. It never rotates.
- **Hard** looks 4 moves ahead, rotations included, and always plays its best move.
- **Expert** looks 6 moves ahead, rotations included.

Check that each level beats the one below it, without a time limit, from random 3-disc openings with the first move swapped:

| Match            | Games | Upper wins | Draws | Upper loses |
|------------------|-------|------------|-------|-------------|
| Easy vs Beginner | 50    | 37         | 0     | 13          |
| Medium vs Easy   | 30    | 27         | 1     | 2           |
| Hard vs Medium   | 30    | 28         | 2     | 0           |
| Expert vs Hard   | 30    | 17         | 8     | 5           |

These are small samples. Proposal test 4 repeats this with 100 games per pair in `bench.py`.

### 3.7 Values at the root ( `Search.root_values` , line 122)

The root is searched move by move, so the program has a value for every move:

- `exact=False` (Hard, Expert). Each move after the first is searched with  $\alpha$  = the best value so far. A move that cannot beat the best is cut off early and gets the best value as an upper bound. The best move's value is exact. The others are only bounds, which is enough to pick the best.
- `exact=True` (Beginner, Easy, Medium, all  $T > 0$ ). Every move is searched with the full window  $(-\infty, +\infty)$ , so every value is exact. Softmax needs this, because it weighs every move by its value (proposal: “pada tingkat dengan  $T > 0$  setiap langkah di root dicari dengan window alpha-beta penuh”). Pruning still happens inside each move's subtree.

### 3.8 Softmax choice ( `softmax_choice` , line 147)

$$P(m) \propto \exp( (v(m) - v^*) / T )$$

$v^*$  is the best value. Subtracting it makes the best move's weight exactly 1 and every other weight smaller, so `exp` never overflows. `random.choices` then draws one move with those weights. Example with four moves valued 10, 8, 4 and  $-1000$  (a move that loses):

| T              | v = 10 | v = 8 | v = 4 | v = -1000   |
|----------------|--------|-------|-------|-------------|
| 200 (Beginner) | 33.7%  | 33.4% | 32.7% | 0.2%        |
| 5 (Medium)     | 50.7%  | 34.0% | 15.3% | $\approx 0$ |
| 2 (Easy)       | 70.5%  | 25.9% | 3.5%  | $\approx 0$ |

At  $T = 5$  the losing move's weight is  $e^{-202}$ , which is 0 in practice. At  $T = 200$  it is  $e^{-5.05}$ , so 0.2% of picks. This cannot happen in the game: Beginner looks only at its own move and never rotates, and a drop can never lose at once, so no move of Beginner's ever scores  $-1000$ . At  $T = 0$  the code skips softmax and calls `best_move`, which takes the highest value. Ties go to the move searched first, because Python's `max` keeps the first maximum and the dictionary keeps search order.

### 3.9 The one entry point (`choose`, line 179)

`choose(state, level_name)` returns `decide(state, level_name).move`. This is the only function the interface calls (proposal: “UI akan memanggil agen melalui satu fungsi, `choose(state, level)`”). `decide` also returns the values, the depth reached and the positions visited, for the experiments.

### 3.10 Measured on this server

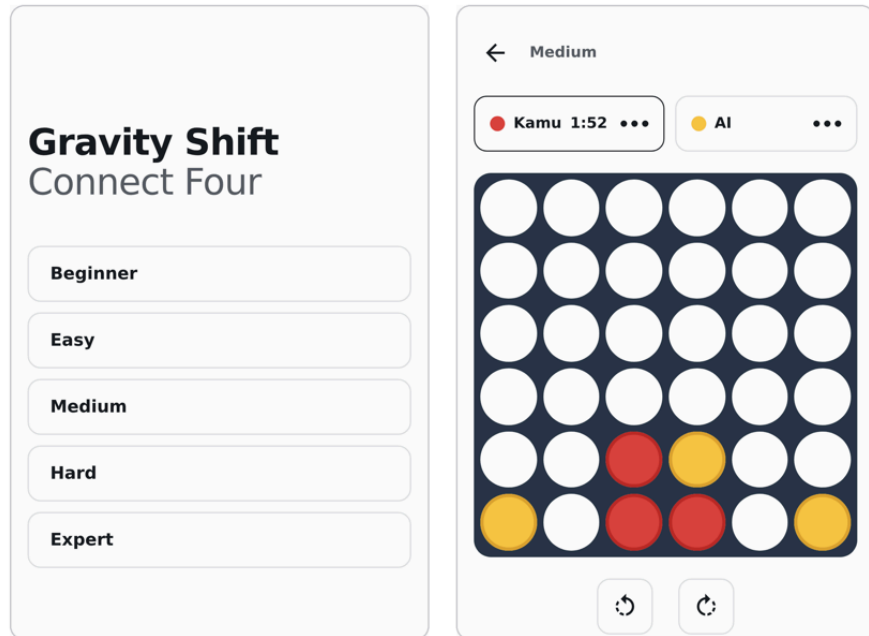
20 random mid-game positions (6 to 14 random moves played), agent to move, 1-second limit:

| Level    | Mean time | Max time  | Depth reached | Positions visited (mean) |
|----------|-----------|-----------|---------------|--------------------------|
| Beginner | < 0.001 s | < 0.001 s | 1             | 6                        |
| Easy     | < 0.001 s | < 0.001 s | 1             | 6                        |
| Medium   | 0.002 s   | 0.002 s   | 2             | 47                       |
| Hard     | 0.019 s   | 0.032 s   | 4             | 485                      |
| Expert   | 0.219 s   | 0.439 s   | 6             | 5,604                    |

Every level reaches its full depth well inside 1 second. The proposal limits the depth to 6 for this reason: “kedalaman terbesar dibatasi 6 agar tingkat Expert dapat selesai dalam 1 detik.” These are a small check, not the evaluation of proposal 2.2.

## 4. The interface: `main.py`

Proposal: **Input dan Output; Arsitektur Software**, Tabel 6, “Alur satu giliran”



The level menu and the game page.

### 4.1 Pages

- `/` (`menu_page`, line 74): the name and five links, one per level. Each link opens `/play/<level>`.
- `/play/<level>` (`game_page`, line 272): builds one `Game`. NiceGUI runs a `@ui.page` function again for every visitor, so each browser tab gets its own `Game` object and its own state (proposal: “Setiap tab browser mendapat permainan sendiri”). An unknown level name sends the visitor back to the menu.

### 4.2 The `Game` object (line 118)

| Field                                              | Holds                                                                 |
|----------------------------------------------------|-----------------------------------------------------------------------|
| <code>state</code>                                 | the engine state tuple                                                |
| <code>result</code>                                | <code>None</code> while playing, then HUMAN, AGENT, DRAW or “timeout” |
| <code>waiting_for_human</code>                     | True only while the human may move. Every input is ignored otherwise. |
| <code>time_left</code> , <code>turn_started</code> | the human's clock (4.4)                                               |

The page shows only what the proposal lists: the board, the human's clock, the rotations left for both players (three dots each) and two rotate buttons (proposal: “Halaman permainan menampilkan papan, timer pemain, sisa jatah rotasi kedua pemain, dan dua tombol rotasi”).

### 4.3 One turn, step by step

| Proposal step                                                                                                                                      | Code                                                                                                                                                                                                                                                                                          |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. The player taps a column or a rotate button. The interface checks <code>legal_moves</code> , stops the clock, and computes <code>apply</code> . | <code>on_human_move</code> (line 217): returns at once if <code>waiting_for_human</code> is <code>False</code> or the move is not legal. Sets <code>waiting_for_human = False</code> , calls <code>stop_clock</code> , then <code>play(move)</code> , which calls <code>engine.apply</code> . |
| 2. Animation, then <code>winner</code> . If the game is over, the result screen.                                                                   | <code>play</code> (line 238): animates (4.6), then <code>engine.winner</code> , then <code>finish</code> if there is a result.                                                                                                                                                                |
| 3. Waiting status, <code>choose</code> in a separate process, agent's move shown no sooner than 0.35 s.                                            | <code>agent_turn</code> (line 226): adds the <code>thinking</code> class (the yellow dot pulses), awaits <code>run.cpu_bound(ai.choose, state, level)</code> , then sleeps whatever is left of <code>MIN_THINK_SECONDS = 0.35</code> .                                                        |
| 4. The agent's move is applied and animated, <code>winner</code> again.                                                                            | <code>agent_turn</code> calls <code>play(move)</code> .                                                                                                                                                                                                                                       |
| 5. The clock runs again, wait for the next move.                                                                                                   | <code>start_human_turn</code> (line 205): <code>waiting_for_human = True</code> , <code>start_clock</code> , redraw the player chips and buttons.                                                                                                                                             |

Keyboard: keys 1 to 6 drop in that column, Q rotates left, E rotates right (`KEY_MOVES`, `on_key`). They go through the same `on_human_move`, so the same checks apply.

### 4.4 The clock

- `time_left` starts at `HUMAN_TIME = 120` seconds.
- `start_clock` notes `time.monotonic()`. `stop_clock` subtracts the time since then and clears the note. While the note is clear, the clock does not run. The note is set only in `start_human_turn` and cleared as soon as the human moves, so the clock stops during the agent's thinking and during both animations, as the proposal says.
- `time.monotonic()` never jumps when the system clock changes, unlike `time.time()`.
- `ui.timer(0.2, update_clock)` redraws the clock 5 times a second, turns it red at 15 s (`LOW_TIME`), and ends the game with `"timeout"` at 0.

### 4.5 Why the agent runs in another process

NiceGUI serves every player from one Python process with one event loop. A 1-second search in that loop would freeze every open tab for that second, clocks and clicks included. Python threads do not help, because the global interpreter lock (GIL) lets only one thread run Python code at a time. `run.cpu_bound` sends `ai.choose` and its arguments to a pool of worker processes and waits without blocking the loop. The state is a tuple of small integers and the level is a string, so they pass to the worker as they are.

- `run.process_pool_start_method = "spawn"` (line 289) starts each worker as a fresh Python process. This avoids copying the running server into the worker.
- `start_agent_worker` (line 281) calls the agent once when the server starts. Without it, the first move of the first game waited about 4 s for the worker to start.
- If the server shuts down mid-search, `run.cpu_bound` returns `None` and `agent_turn` stops.

## 4.6 Animations and their timing

Proposal: none. Animations are part of the Result.

| Animation           | Shows                        | Timing                                        | Where                                                                              |
|---------------------|------------------------------|-----------------------------------------------|------------------------------------------------------------------------------------|
| Drop                | where the new disc landed    | 110 ms $\times \sqrt{\text{rows}}$            | <code>play</code> , <code>fall_ms</code> ; CSS <code>@keyframes fall</code>        |
| Rotation            | the board turned             | 380 ms turn, then the fall                    | <code>play</code> , <code>ROTATE_MS</code> ; CSS <code>turn-left/turn-right</code> |
| Fall after rotation | where every disc landed      | 110 ms $\times \sqrt{\text{rows}}$ , per disc | <code>fall_rows_after_rotation</code>                                              |
| Pulse               | the agent is thinking        | 900 ms loop                                   | <code>agent_turn</code> ; CSS <code>.thinking</code>                               |
| Fade                | the winning line, at the end | 240 ms                                        | <code>finish</code> ; CSS <code>.faded</code>                                      |

**Why the square root.** An object falling from rest covers

$$s = \frac{1}{2} g t^2$$

so the time to fall a distance  $s$  is

$$t = \sqrt{2s / g}$$

which grows with  $\sqrt{s}$ . One row takes 110 ms, so 4 rows take  $110 \times \sqrt{4} = 220$  ms and 6 rows take 269 ms. The CSS curve `cubic-bezier(0.11, 0, 0.5, 0)` is the standard “ease-in-quad” curve, which approximates distance  $\propto \text{time}^2$ , so the disc speeds up as it falls instead of moving at a constant speed.

**How a drop is drawn.** `play` finds the landing row in the board before the move. A disc landing in row  $r$  starts  $r + 1$  rows above it, just above the board. `render_board` gives that one disc the class `falling` and two CSS variables, `--rows` and `--fall-ms`. The keyframe starts it `--rows  $\times$  (hole height + gap)` higher and moves it to its hole. Only `transform` is animated, so the browser does not recompute the layout.

**How a rotation is drawn.** `play` adds the class `turn-L` or `turn-R` to the board. CSS turns the whole board  $90^\circ$  in 380 ms on `cubic-bezier(0.77, 0, 0.175, 1)`, a curve that starts slowly and stops slowly. After 380 ms the server redraws the board with the new state and removes the class. `fall_rows_after_rotation` takes the board right after `rotate` and before `fall` and works out how many rows each disc drops. Those discs then fall with the same rule as a drop.

**Reduced motion.** If the device asks for less motion (`prefers-reduced-motion: reduce`), the fall, turn and pulse animations are switched off and the board just updates.

---

## 5. Look: `style.css`

- Colours are CSS variables on `:root` in OKLCH. A `prefers-color-scheme: dark` block redefines them, so the theme follows the device (proposal: “Tema terang atau gelap mengikuti pengaturan perangkat”).
- Red `--red` is the human and yellow `--yellow` is the agent, as in classic Connect Four. The board is dark slate in both themes.
- The board is a CSS grid of 6 column buttons, each a grid of 6 holes, with `aspect-ratio: 1` so it stays square at any width. The page is at most 440 px wide and fits a phone with 16 px side margins.
- Columns change colour on hover only on devices with a mouse (`@media (hover: hover) and (pointer: fine)`), so a phone does not show a hover colour that sticks after a tap.
- Buttons shrink to 97% while pressed. `touch-action: manipulation` stops a fast second tap from zooming the page on phones.

---

## 6. Tests: `test_engine.py`

Proposal: **2.2 Metode Evaluasi**, tests 1 and 2. Run with `python test_engine.py`. All 22 pass.

| Test                                                                           | Checks                               | Proposal                |
|--------------------------------------------------------------------------------|--------------------------------------|-------------------------|
| <code>new_game, there_are_54_lines</code>                                      | start state, 8 legal moves, 54 lines | Aturan, Fungsi Evaluasi |
| <code>drop_lands_on_the_lowest_empty_cell, full_column_is_not_legal</code>     | drops and full columns               | test 1                  |
| <code>rotate_right_then_left_gives_back_the_board</code>                       | the two maps are inverses            | test 1                  |
| <code>rotate_right/left_turns_the_bottom_row...</code>                         | direction of each turn               | test 1, Gambar 1        |
| <code>discs_fall_after_a_rotation</code>                                       | no disc floats over an empty cell    | test 1                  |
| <code>rotation_uses_up_the_players_rotations</code>                            | 3 rotations, then none               | test 1                  |
| <code>four_in_a_row_in_every_direction</code>                                  | 4 directions                         | test 1                  |
| <code>rotation_can_make_the_other_player_win</code>                            | human rotates, agent wins            | test 1                  |
| <code>rotation_that_gives_both_players_four_is_a_draw</code>                   | both four → draw                     | test 1                  |
| <code>full_board_without_four_is_a_draw, game_in_progress_has_no_winner</code> | full-board draw, game goes on        | test 1                  |

|                                                                              |                                                                          |                   |
|------------------------------------------------------------------------------|--------------------------------------------------------------------------|-------------------|
| alphabeta_gives_the_same_value_as_minimax                                    | 30 positions, depth 3, with and without rotations                        | test 2 (small)    |
| evaluation_stays_far_below_a_win                                             | $ eval  \leq 270$ on 200 boards                                          | Fungsi Evaluasi   |
| agent_takes_an_immediate_win,<br>agent_blocks_an_immediate_loss              | Hard and Expert win in one, and leave no winning reply                   | Algoritma         |
| softmax_almost_never_picks_a_losing_move,<br>softmax_picks_close_moves_often | 0 of 1000 losing picks; $P \approx 0.38$ for a move 10 below at $T = 20$ | Tingkat Kesulitan |
| low_levels_never_rotate                                                      | Beginner, Easy and Medium never return L or R                            | Tingkat Kesulitan |
| agent_answers_within_the_time_limit                                          | Expert returns within 1.2 s                                              | Rumusan Masalah   |

The blocking test checks the result, not one column. The agent may block a threat by rotating, which is also a correct defence. So the test plays every human reply after the agent's move and checks that none of them wins.

---

## 7. The proposal and the code

### 7.1 Where each proposal section is in the code

| Proposal section                               | What it claims                                                                                 | Where in the code                                                    |
|------------------------------------------------|------------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| 2.1 Aturan Permainan                           | 6 × 6 board, drops, rotation with fall, 3 rotations each, win and draw rules                   | engine.py: apply, rotate, fall, winner                               |
| 2.1 Aturan Permainan                           | 2-minute clock for the human, stops while the agent thinks and during animation                | main.py: Game.start_clock, stop_clock, update_clock                  |
| 2.1 Algoritma                                  | Minimax, alpha-beta, centre-first ordering, rotations as moves, at most 8 branches             | ai.py: Search.alphabeta, Search.moves, MOVE_ORDER                    |
| 2.1 Kompleksitas Waktu, Tabel 1                | Iterative deepening, 1-second limit, depth 1 always finishes, previous best searched first     | ai.py: decide, Search.leaf_or_none, OutOfTime                        |
| 2.1 Fungsi Evaluasi, Tabel 2                   | 54 lines, weights $\pm 5$ and $\pm 2$ , terminal values 1000 – moves, $ eval  \leq 270$        | ai.py: LINE_SCORE, evaluate, final_value; engine.LINES               |
| 2.1 Tingkat Kesulitan, Tabel 3                 | Depth, rotations in search, softmax temperature; full window at the root when $T > 0$          | ai.py: Level, LEVELS, softmax_choice, root_values(exact=True)        |
| 2.1 PEAS, Tabel 4 and 5                        | What the agent senses and does                                                                 | Sensors = the state tuple. Actuators = the move returned by choose   |
| 2.1 Input dan Output                           | Two pages. The game page shows the board, the timer, the rotations left and two rotate buttons | main.py: menu_page, Game                                             |
| 2.1 Arsitektur, Gambar 2, Tabel 6              | Three layers, NiceGUI, choose(state, level) in a separate process                              | main.py: agent_turn calls run.cpu_bound(ai.choose, ...)              |
| 2.1 Alur satu giliran (5 steps)                | The order of one turn                                                                          | main.py: on_human_move → play → agent_turn → play → start_human_turn |
| 2.2 Metode Evaluasi, tests 1 and 2             | Rule tests; alpha-beta equals minimax                                                          | test_engine.py                                                       |
| 2.2 Metode Evaluasi, tests 2 to 5 at full size | 200 positions, baseline matches, level matches, speed                                          | bench.py, not written yet (7.2)                                      |

## 7.2 Proposal claims checked against the code

| Claim in the proposal                                                                                                           | Status                                                                                     |
|---------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| 6 × 6, human first, drop or rotate, 3 rotations each                                                                            | <b>matches</b>                                                                             |
| Rotation legal even if nothing changes                                                                                          | <b>matches</b>                                                                             |
| One player with four after a rotation wins, even if the other player rotated. Both with four is a draw. A full board is a draw. | <b>matches, tested</b>                                                                     |
| 2-minute clock, stops while the agent thinks and during animation, timeout loses                                                | <b>matches</b>                                                                             |
| At most 42 moves                                                                                                                | <b>follows from the rules</b>                                                              |
| At most 8 branches, rotations as normal moves, rotation counts in the state                                                     | <b>matches</b>                                                                             |
| Centre-first order. Previous best searched first.                                                                               | <b>matches (previous best at the root only)</b>                                            |
| Alpha-beta gives the same result as minimax                                                                                     | <b>matches, tested</b>                                                                     |
| Iterative deepening, 1 s, depth 1 always finishes                                                                               | <b>matches</b>                                                                             |
| Tabel 2 weights, win = 1000 – moves,  eval  ≤ 270                                                                               | <b>matches, tested</b>                                                                     |
| Tabel 3 levels. Full window at the root when T > 0.                                                                             | <b>matches</b>                                                                             |
| Depth at most 6, so Expert finishes inside 1 s                                                                                  | <b>matches: max 0.439 s measured</b>                                                       |
| One call <code>choose(state, level)</code> in a separate process via <code>run.cpu_bound</code>                                 | <b>matches</b>                                                                             |
| Each tab has its own game                                                                                                       | <b>matches</b>                                                                             |
| Agent's move shown no sooner than 0.35 s                                                                                        | <b>matches</b>                                                                             |
| Animations: √distance fall, 0.38 s turn, pulse, fade, reduced motion                                                            | <b>matches</b>                                                                             |
| Only NiceGUI as a library                                                                                                       | <b>matches ( requirements.txt : nicegui==3.18.0 )</b>                                      |
| Tabel 6 lists <code>bench.py</code>                                                                                             | <b>not written yet</b>                                                                     |
| Test 2 at 200 positions and depth 5. Tests 3 to 5.                                                                              | <b>not run yet. They belong in bench.py . The current test is 30 positions at depth 3.</b> |
| Test 6, game length with at least 5 players                                                                                     | <b>not run yet, needs people</b>                                                           |

---

## 8. Questions a lecturer may ask

### Is the softmax level a learning algorithm?

No. The temperature is a fixed number per level, written in the source. The program never changes it from results. Softmax here is only a rule for drawing a random move with weights.

### **Why softmax and not “play a random move 20% of the time”?**

A fixed random rate can pick a move that loses at once. Softmax weighs every move by its minimax value, so a losing move (about  $-1000$ ) gets a weight near  $10^{-22}$ , while moves close to the best are picked often.

### **Why does the environment count as deterministic when the agent uses randomness?**

The rules have no chance in them, and the result of a move, rotations included, can be computed. The randomness is in how the agent chooses, not in what a move does (proposal Tabel 5).

### **Why is the bound on the evaluation 270?**

Each of the 54 lines scores between  $-5$  and  $+5$ , so the sum is at most 270 in size. The real maximum is lower, because the lines overlap. The bound only has to be below the smallest win value, 958.

### **Why subtract the number of moves from a win?**

Without it, a win in 1 and a win in 7 have the same value, and the agent may delay a win it already has.

### **Why does alpha-beta give the same answer as minimax?**

It only skips a branch once it has proof that the branch cannot change the value at the root. The test runs both on 30 positions and checks equal values.

### **Why does $T > 0$ need a full window?**

With a narrow window, a move that cannot beat the best gets a bound, not its value. Softmax would then weigh it by a wrong number.

### **Why iterative deepening? It repeats work.**

It guarantees a move inside 1 s whatever the position: the agent always has the last finished depth. The repeated shallow depths are small next to the deepest one. They also give the best move to search first, which helps pruning.

### **Why not use the remaining time of a fast move later?**

The proposal sets 1 second per move. A time bank would be extra code that the proposal does not need.

### **What if the human rotates on the empty board?**

It is legal and uses a rotation. Nothing moves, so no fall is animated.

### **Can a drop give the opponent four?**

No. A drop adds one disc of the mover's colour, so only the mover can complete a line. Only a rotation moves other discs.

### **Why do Beginner, Easy and Medium not search rotations?**

It is the second lever on difficulty, next to depth. The agent never rotates and does not see the human's rotations coming. The rules stay the same for the human.

**Why NiceGUI and not Flask with JavaScript?**

The whole program, the interface included, stays in Python, so there is no second language and no API between a front end and a back end. The game is a phone link away during development.

**Why another process for the agent?**

Python runs one thread of Python code at a time (the GIL), so a search in the server process would freeze every tab for up to a second (section 4.5).

**What does the clock do during the agent's turn?**

It is stopped. It only runs between `start_human_turn` and the human's next move.

---

## Appendix: full source

Line numbers match the references above.

## engine.py

```
1  """Rules of Gravity Shift Connect Four.
2
3  State: (cells, to_move, rotations_left). cells holds 36 values, row by row from the top left.
4  Move: a column 0-5, or "L" / "R" to turn the board left or right.
5  """
6
7  SIZE = 6
8  EMPTY, HUMAN, AGENT = 0, 1, 2
9  ROTATIONS_PER_PLAYER = 3
10 DRAW = 0
11
12 COLUMNS = range(SIZE)
13 ROTATIONS = ("L", "R")
14
15
16 def index(row, col):
17     return row * SIZE + col
18
19
20 def _all_lines():
21     """All 54 lines of 4 cells."""
22     directions = [(0, 1), (1, 0), (1, 1), (1, -1)]
23     lines = []
24     for row in range(SIZE):
25         for col in range(SIZE):
26             for d_row, d_col in directions:
27                 end_row, end_col = row + 3 * d_row, col + 3 * d_col
28                 if 0 <= end_row < SIZE and 0 <= end_col < SIZE:
29                     lines.append(tuple(index(row + i * d_row, col + i * d_col) for i in range(4)))
30     return lines
31
32
33 LINES = _all_lines()
34
35
36 def _rotation_map(clockwise):
37     """For each cell after the turn, the cell it comes from."""
38     source = []
39     for row in range(SIZE):
40         for col in range(SIZE):
41             if clockwise:
42                 source.append(index(SIZE - 1 - col, row))
43             else:
44                 source.append(index(col, SIZE - 1 - row))
45     return source
46
47
48 ROTATION_SOURCE = {"R": _rotation_map(clockwise=True), "L": _rotation_map(clockwise=False)}
49
50
51 def new_game():
52     """Empty board, human to move."""
53     return ((EMPTY,) * (SIZE * SIZE), HUMAN, (ROTATIONS_PER_PLAYER, ROTATIONS_PER_PLAYER))
54
55
56 def other(player):
57     return AGENT if player == HUMAN else HUMAN
58
59
60 def lowest_empty_row(cells, col):
61     """Landing row in `col`, or None if full."""
62     for row in reversed(range(SIZE)):
63         if cells[index(row, col)] == EMPTY:
64             return row
65     return None
```

```

66
67
68 def rotate(cells, direction):
69     """The board turned 90 degrees, before gravity."""
70     source = ROTATION_SOURCE[direction]
71     return tuple(cells[source[i]] for i in range(SIZE * SIZE))
72
73
74 def fall(cells):
75     """Every disc moved to the bottom of its column, order kept."""
76     result = [EMPTY] * (SIZE * SIZE)
77     for col in COLUMNS:
78         discs = [cells[index(row, col)] for row in range(SIZE) if cells[index(row, col)] != EMPTY]
79         for offset, disc in enumerate(reversed(discs)):
80             result[index(SIZE - 1 - offset, col)] = disc
81     return tuple(result)
82
83
84 def legal_moves(state):
85     """Open columns, then rotations if any are left."""
86     cells, to_move, rotations_left = state
87     moves = [col for col in COLUMNS if cells[index(0, col)] == EMPTY]
88     if rotations_left[to_move - 1] > 0:
89         moves += list(ROTATIONS)
90     return moves
91
92
93 def apply(state, move):
94     """State after a legal move."""
95     cells, to_move, rotations_left = state
96     if move in ROTATIONS:
97         cells = fall(rotate(cells, move))
98         left = list(rotations_left)
99         left[to_move - 1] -= 1
100         rotations_left = tuple(left)
101     else:
102         row = lowest_empty_row(cells, move)
103         cells = cells[:index(row, move)] + (to_move,) + cells[index(row, move) + 1:]
104     return (cells, other(to_move), rotations_left)
105
106
107 def four_in_a_row(cells):
108     """Players with a line of four."""
109     return {cells[a] for a, b, c, d in LINES if cells[a] != EMPTY and cells[a] == cells[b] ==
cells[c] == cells[d]}
110
111
112 def winning_cells(cells):
113     """Cells in any line of four."""
114     found = set()
115     for line in LINES:
116         first = cells[line[0]]
117         if first != EMPTY and all(cells[i] == first for i in line):
118             found.update(line)
119     return found
120
121
122 def winner(state):
123     """HUMAN, AGENT, DRAW (both have four, or board full), or None."""
124     cells = state[0]
125     players = four_in_a_row(cells)
126     if len(players) == 2:
127         return DRAW
128     if len(players) == 1:
129         return players.pop()
130     if EMPTY not in cells:
131         return DRAW
132     return None

```

## ai.py

```
1  """The agent: minimax with alpha-beta pruning and iterative deepening."""
2
3  import math
4  import random
5  import time
6  from dataclasses import dataclass
7
8  import engine
9
10 TIME_LIMIT = 1.0 # seconds per move
11 WIN = 1000
12 INFINITY = math.inf
13
14 # Line score by (my discs, opponent discs). Other mixes score 0.
15 LINE_SCORE = {
16     (3, 0): 5,
17     (2, 0): 2,
18     (0, 2): -2,
19     (0, 3): -5,
20 }
21
22 # Centre first, so alpha-beta prunes more.
23 MOVE_ORDER = [2, 3, 1, 4, 0, 5, "L", "R"]
24
25
26 @dataclass(frozen=True)
27 class Level:
28     depth: int
29     search_rotations: bool
30     temperature: float # 0 = always the best move
31
32
33 LEVELS = {
34     "Beginner": Level(depth=1, search_rotations=False, temperature=200),
35     "Easy": Level(depth=1, search_rotations=False, temperature=2),
36     "Medium": Level(depth=2, search_rotations=False, temperature=5),
37     "Hard": Level(depth=4, search_rotations=True, temperature=0),
38     "Expert": Level(depth=6, search_rotations=True, temperature=0),
39 }
40
41
42 class OutOfTime(Exception):
43     """The time limit passed."""
44
45
46 def evaluate(cells, me):
47     """Sum of line scores. At most 270 in size, so below any win."""
48     opponent = engine.other(me)
49     total = 0
50     for line in engine.LINES:
51         values = [cells[i] for i in line]
52         total += LINE_SCORE.get((values.count(me), values.count(opponent)), 0)
53     return total
54
55
56 def final_value(result, me, moves_played):
57     """Faster wins and slower losses score higher."""
58     if result == engine.DRAW:
59         return 0
60     if result == me:
61         return WIN - moves_played
62     return -WIN + moves_played
63
64
65 class Search:
```

```

66
67 def __init__(self, me, search_rotations, deadline=None):
68     self.me = me
69     self.search_rotations = search_rotations
70     self.deadline = deadline # time.monotonic() value or None
71     self.nodes = 0
72
73 def moves(self, state, first=None):
74     """Legal moves in search order, `first` at the front."""
75     legal = engine.legal_moves(state)
76     if not self.search_rotations:
77         legal = [move for move in legal if move not in engine.ROTATIONS]
78     ordered = [move for move in MOVE_ORDER if move in legal]
79     if first in ordered:
80         ordered.remove(first)
81         ordered.insert(0, first)
82     return ordered
83
84 def leaf_or_none(self, state, depth, moves_played):
85     """Value if the search stops here, else None."""
86     self.nodes += 1
87     if self.deadline is not None and time.monotonic() > self.deadline:
88         raise OutOfTime
89     result = engine.winner(state)
90     if result is not None:
91         return final_value(result, self.me, moves_played)
92     if depth == 0:
93         return evaluate(state[0], self.me)
94     return None
95
96 def alphabeta(self, state, depth, alpha, beta, moves_played):
97     """Minimax value. Stops a node once alpha >= beta."""
98     leaf = self.leaf_or_none(state, depth, moves_played)
99     if leaf is not None:
100         return leaf
101
102     agent_to_move = state[1] == self.me
103     for move in self.moves(state):
104         value = self.alphabeta(engine.apply(state, move), depth - 1, alpha, beta, moves_played
+ 1)
105         if agent_to_move:
106             alpha = max(alpha, value)
107         else:
108             beta = min(beta, value)
109             if alpha >= beta:
110                 break
111     return alpha if agent_to_move else beta
112
113 def minimax(self, state, depth, moves_played):
114     """Minimax without pruning, for the tests."""
115     leaf = self.leaf_or_none(state, depth, moves_played)
116     if leaf is not None:
117         return leaf
118
119     values = [self.minimax(engine.apply(state, move), depth - 1, moves_played + 1) for move in
self.moves(state)]
120     return max(values) if state[1] == self.me else min(values)
121
122 def root_values(self, state, depth, exact, first=None):
123     """Value of each move. exact=False gives upper bounds for all but the best."""
124     values = {}
125     best = -INFINITY
126     for move in self.moves(state, first):
127         alpha = -INFINITY if exact else best
128         value = self.alphabeta(engine.apply(state, move), depth - 1, alpha, INFINITY, 1)
129         values[move] = value
130         best = max(best, value)
131     return values
132

```

```

133
134 @dataclass
135 class Decision:
136     move: object
137     values: dict
138     depth: int
139     nodes: int
140
141
142 def best_move(values):
143     """Highest value. Ties go to the move searched first."""
144     return max(values, key=values.get)
145
146
147 def softmax_choice(values, temperature, rng):
148     """P(move) proportional to exp((value - best) / temperature)."""
149     top = max(values.values())
150     moves = list(values)
151     weights = [math.exp((values[move] - top) / temperature) for move in moves]
152     return rng.choices(moves, weights=weights)[0]
153
154
155 def decide(state, level_name, time_limit=TIME_LIMIT, rng=random):
156     """Depth 1, 2, 3, ... until the level's depth or the time limit. Depth 1 always finishes."""
157     level = LEVELS[level_name]
158     deadline = None if time_limit is None else time.monotonic() + time_limit
159     search = Search(state[1], level.search_rotations)
160     exact = level.temperature > 0
161
162     values, depth_done = None, 0
163     for depth in range(1, level.depth + 1):
164         search.deadline = None if depth == 1 else deadline
165         first = best_move(values) if values else None
166         try:
167             values = search.root_values(state, depth, exact, first)
168         except OutOfTime:
169             break
170         depth_done = depth
171
172     if level.temperature > 0:
173         move = softmax_choice(values, level.temperature, rng)
174     else:
175         move = best_move(values)
176     return Decision(move, values, depth_done, search.nodes)
177
178
179 def choose(state, level_name):
180     return decide(state, level_name).move

```

## main.py

```
1  """Web interface for Gravity Shift Connect Four. Run: python main.py"""
2
3  import asyncio
4  import math
5  import time
6  from pathlib import Path
7
8  from nicegui import app, run, ui
9
10 import ai
11 import engine
12 from engine import AGENT, COLUMNS, DRAW, HUMAN, SIZE, index
13
14 PORT = 3117
15 HUMAN_TIME = 120 # seconds per game
16 LOW_TIME = 15 # clock turns red
17
18 ROTATE_MS = 380
19 FALL_MS_ONE_ROW = 110
20 MIN_THINK_SECONDS = 0.35 # so the agent's move is visible
21
22 RESULT_TEXT = {HUMAN: "Kamu menang", AGENT: "AI menang", DRAW: "Seri", "timeout": "Waktu habis"}
23 KEY_MOVES = {"1": 0, "2": 1, "3": 2, "4": 3, "5": 4, "6": 5, "q": "L", "e": "R"}
24
25 STYLE = Path(__file__).with_name("style.css")
26
27
28 def fall_ms(rows):
29     """Free fall: time grows with the square root of distance."""
30     return round(FALL_MS_ONE_ROW * math.sqrt(rows))
31
32
33 def fall_rows_after_rotation(turned):
34     """{landing cell: rows fallen} for the board right after a turn."""
35     rows_fallen = {}
36     for col in COLUMNS:
37         filled_rows = [row for row in range(SIZE) if turned[index(row, col)] != engine.EMPTY]
38         for offset, row in enumerate(reversed(filled_rows)):
39             landing_row = SIZE - 1 - offset
40             rows_fallen[index(landing_row, col)] = landing_row - row
41     return rows_fallen
42
43
44 def add_style():
45     ui.add_css(STYLE)
46     ui.add_css(f":root {{ --rotate-ms: {ROTATE_MS}ms; }}")
47     ui.add_head_html(
48         '<meta name="theme-color" content="#fafafa" media="(prefers-color-scheme: light)">'
49         '<meta name="theme-color" content="#0e1013" media="(prefers-color-scheme: dark)">'
50     )
51
52
53 def link(text, href, classes=""):
54     with ui.element("a").classes(f"btn {classes}").props(f'href="{href}"') as element:
55         ui.label(text)
56     return element
57
58
59 def icon_link(icon, label, href, classes=""):
60     with ui.element("a").classes(f"btn icon {classes}").props(f'href="{href}" aria-label="{label}"') as element:
61         ui.icon(icon).props('aria-hidden="true"')
62     return element
63
64
```

```

65 def icon_button(icon, label, on_click, classes=""):
66     with ui.element("button").classes(f"btn icon {classes}").props(f'aria-label="{
{label}"').on("click", on_click) as element:
67         ui.icon(icon).props('aria-hidden="true"')
68         return element
69
70
71 # Level menu
72
73 @ui.page("/")
74 def menu_page():
75     add_style()
76     ui.page_title("Gravity Shift")
77     with ui.element("main").classes("screen menu"):
78         with ui.element("h1").classes("wordmark").props('translate="no"'):
79             ui.label("Gravity Shift")
80             ui.label("Connect Four").classes("wordmark-sub")
81         with ui.element("nav").classes("levels"):
82             for name in ai.LEVELS:
83                 link(name, f"/play/{name}", "level")
84
85
86 # Game
87
88 class PlayerChip:
89     """Name, rotations left and clock of one player."""
90
91     def __init__(self, name, player, with_clock=False):
92         self.player = player
93         with ui.element("div").classes(f"player p{player}").as self.element:
94             ui.element("span").classes("dot")
95             ui.label(name).classes("player-name")
96             self.clock = ui.label("").classes("clock") if with_clock else None
97             self.pips = ui.element("span").classes("pips").props(f'aria-label="Sisa putaran
{name}"')
98
99     def update(self, state, active):
100         if active:
101             self.element.classes(add="active")
102         else:
103             self.element.classes(remove="active")
104         left = state[2][self.player - 1]
105         self.pips.clear()
106         with self.pips:
107             for i in range(engine.ROTATIONS_PER_PLAYER):
108                 ui.element("span").classes("pip" if i < left else "pip used")
109
110     def show_clock(self, text, low):
111         self.clock.set_text(text)
112         if low:
113             self.clock.classes(add="low")
114         else:
115             self.clock.classes(remove="low")
116
117
118 class Game:
119     """One game in one browser tab."""
120
121     def __init__(self, level_name):
122         self.level_name = level_name
123         self.state = engine.new_game()
124         self.result = None # HUMAN, AGENT, DRAW or "timeout"
125         self.waiting_for_human = False
126         self.time_left = HUMAN_TIME
127         self.turn_started = None # time.monotonic()
128
129         with ui.element("main").classes("screen game"):
130             with ui.element("header").classes("topbar"):
131                 icon_link("arrow_back", "Menu", "/", "back")

```

```

132         ui.label(level_name).classes("level-name")
133         with ui.element("div").classes("players"):
134             self.human_chip = PlayerChip("Kamu", HUMAN, with_clock=True)
135             self.agent_chip = PlayerChip("AI", AGENT)
136             self.board = ui.element("div").classes("board")
137             self.controls = ui.element("div").classes("controls").props('aria-live="polite"')
138
139         ui.keyboard(on_key=self.on_key, ignore=[])
140         ui.timer(0.2, self.update_clock)
141         self.render_board()
142         self.start_human_turn()
143
144     def render_players(self):
145         for chip in (self.human_chip, self.agent_chip):
146             chip.update(self.state, active=self.result is None and self.state[1] == chip.player)
147
148     def render_board(self, falls=None, winning=None):
149         """`falls` maps cell to rows fallen, for discs that animate in."""
150         falls = falls or {}
151         cells = self.state[0]
152         self.board.clear()
153         with self.board:
154             for col in COLUMNS:
155                 column = ui.element("button").classes("column").props(f'aria-label="Kolom {col +
156 }"')
157                 column.on("click", lambda col=col: self.on_human_move(col))
158                 with column:
159                     for row in range(SIZE):
160                         with ui.element("div").classes("hole"):
161                             player = cells[index(row, col)]
162                             if player == engine.EMPTY:
163                                 continue
164                             disc = ui.element("div").classes(f"disc p{player}")
165                             rows = falls.get(index(row, col), 0)
166                             if rows > 0:
167                                 disc.classes("falling").style(f"--rows: {rows}; --fall-ms:
168 {fall_ms(rows)}ms")
169                             if winning is not None and index(row, col) not in winning:
170                                 disc.classes("faded")
171
172     def render_controls(self):
173         self.controls.clear()
174         with self.controls:
175             if self.result is None:
176                 can_rotate = self.waiting_for_human and self.state[2][HUMAN - 1] > 0
177                 for direction, icon, label in (("L", "rotate_left", "Putar kiri"), ("R",
178 "rotate_right", "Putar kanan")):
179                     rotate = icon_button(icon, label, lambda d=direction: self.on_human_move(d),
180 "rotate")
181                     if not can_rotate:
182                         rotate.props("disabled")
183             else:
184                 ui.label(RESULT_TEXT[self.result]).classes("result")
185                 with ui.element("div").classes("result-actions"):
186                     link("Main lagi", f"/play/{self.level_name}", "primary")
187                     link("Level", "/")
188
189     def seconds_left(self):
190         if self.turn_started is None:
191             return self.time_left
192         return self.time_left - (time.monotonic() - self.turn_started)
193
194     def start_clock(self):
195         self.turn_started = time.monotonic()
196
197     def stop_clock(self):
198         self.time_left = self.seconds_left()
199         self.turn_started = None

```

```

197 def update_clock(self):
198     left = max(0.0, self.seconds_left())
199     shown = math.ceil(left)
200     self.human_chip.show_clock(f"{shown // 60}:{shown % 60:02d}", low=left <= LOW_TIME)
201     if left == 0 and self.waiting_for_human and self.result is None:
202         self.stop_clock()
203         self.finish("timeout")
204
205 def start_human_turn(self):
206     self.waiting_for_human = True
207     self.start_clock()
208     self.render_players()
209     self.render_controls()
210
211 def on_key(self, event):
212     if event.action.keydown and not event.action.repeat:
213         move = KEY_MOVES.get(str(event.key).lower())
214         if move is not None:
215             asyncio.create_task(self.on_human_move(move))
216
217 async def on_human_move(self, move):
218     if not self.waiting_for_human or move not in engine.legal_moves(self.state):
219         return
220     self.waiting_for_human = False
221     self.stop_clock()
222     await self.play(move)
223     if self.result is None:
224         await self.agent_turn()
225
226 async def agent_turn(self):
227     self.agent_chip.element.classes("thinking")
228     started = time.monotonic()
229     move = await run.cpu_bound(ai.choose, self.state, self.level_name)
230     await asyncio.sleep(max(0.0, MIN_THINK_SECONDS - (time.monotonic() - started)))
231     self.agent_chip.element.classes(remove="thinking")
232     if move is None: # server shutting down
233         return
234     await self.play(move)
235     if self.result is None:
236         self.start_human_turn()
237
238 async def play(self, move):
239     """Apply, animate, and end the game if it is over."""
240     before = self.state[0]
241     self.state = engine.apply(self.state, move)
242     self.render_players()
243     self.render_controls()
244
245     if move in engine.ROTATIONS:
246         self.board.classes(f"turn-{move}")
247         await asyncio.sleep(ROTATE_MS / 1000)
248         falls = fall_rows_after_rotation(engine.rotate(before, move))
249         self.board.classes(remove=f"turn-{move}")
250     else:
251         landing_row = engine.lowest_empty_row(before, move)
252         falls = {index(landing_row, move): landing_row + 1} # enters from above the board
253     self.render_board(falls)
254     if falls:
255         await asyncio.sleep(fall_ms(max(falls.values())) / 1000)
256
257     result = engine.winner(self.state)
258     if result is not None:
259         self.finish(result)
260
261 def finish(self, result):
262     self.result = result
263     self.waiting_for_human = False
264     if result in (HUMAN, AGENT, DRAW):
265         winning = engine.winning_cells(self.state[0])

```

```
266         self.render_board(winning=winning or None)
267         self.render_players()
268         self.render_controls()
269
270
271 @ui.page("/play/{level_name}")
272 def game_page(level_name: str):
273     if level_name not in ai.LEVELS:
274         ui.navigate.to("/")
275         return
276     add_style()
277     ui.page_title(f"Gravity Shift · {level_name}")
278     Game(level_name)
279
280
281 async def start_agent_worker():
282     """Warm up the worker so the first move is not slow."""
283     await run.cpu_bound(ai.choose, engine.new_game(), "Beginner")
284
285
286 app.on_startup(start_agent_worker)
287
288 if __name__ in {"__main__", "__mp_main__"}:
289     run.process_pool_start_method = "spawn"
290     ui.run(host="127.0.0.1", port=PORT, title="Gravity Shift", favicon="🔴", reload=False,
show=False, dark=None)
```

## style.css

```
1  /* --rotate-ms, --rows and --fall-ms come from main.py. */
2
3  :root {
4      --bg: oklch(0.985 0 0);
5      --ink: oklch(0.22 0.012 255);
6      --muted: oklch(0.47 0.012 255);
7      --line: oklch(0.89 0.004 255);
8      --surface-hover: oklch(0.955 0.004 255);
9      --board: oklch(0.33 0.035 255);
10     --board-hover: oklch(0.38 0.04 255);
11     --hole: var(--bg);
12     --red: oklch(0.6 0.19 27);
13     --red-edge: oklch(0.52 0.18 27);
14     --yellow: oklch(0.84 0.15 86);
15     --yellow-edge: oklch(0.75 0.14 80);
16     --danger: oklch(0.55 0.2 27);
17
18     --gap: 6px;
19     --ease-turn: cubic-bezier(0.77, 0, 0.175, 1);
20     --ease-gravity: cubic-bezier(0.11, 0, 0.5, 0); /* ease-in-quad: free fall */
21     --ease-out: cubic-bezier(0.23, 1, 0.32, 1);
22
23     color-scheme: light dark;
24 }
25
26 @media (prefers-color-scheme: dark) {
27     :root {
28         --bg: oklch(0.17 0.008 255);
29         --ink: oklch(0.95 0.004 255);
30         --muted: oklch(0.72 0.01 255);
31         --line: oklch(0.3 0.01 255);
32         --surface-hover: oklch(0.22 0.01 255);
33         --board: oklch(0.3 0.03 255);
34         --board-hover: oklch(0.34 0.035 255);
35         --hole: oklch(0.15 0.008 255);
36         --red: oklch(0.64 0.19 27);
37         --red-edge: oklch(0.55 0.18 27);
38         --yellow: oklch(0.85 0.15 86);
39         --yellow-edge: oklch(0.76 0.14 80);
40         --danger: oklch(0.68 0.19 27);
41     }
42 }
43
44 /* Page */
45
46 body {
47     background: var(--bg);
48     color: var(--ink);
49     font-family: system-ui, -apple-system, "Segoe UI", Roboto, sans-serif;
50     -webkit-tap-highlight-color: transparent;
51     touch-action: manipulation;
52 }
53
54 .nicegui-content {
55     padding: 0;
56     align-items: center;
57 }
58
59 .screen {
60     width: 100%;
61     max-width: 440px;
62     min-height: 100dvh;
63     padding: calc(24px + env(safe-area-inset-top)) 16px calc(32px + env(safe-area-inset-bottom));
64     display: flex;
65     flex-direction: column;
```

```

66 }
67
68 /* Buttons */
69
70 .btn {
71     appearance: none;
72     border: 1px solid var(--line);
73     background: transparent;
74     color: var(--ink);
75     font: inherit;
76     font-size: 16px;
77     font-weight: 550;
78     height: 52px;
79     padding: 0 20px;
80     border-radius: 12px;
81     cursor: pointer;
82     display: inline-flex;
83     align-items: center;
84     text-decoration: none;
85     transition: transform 140ms var(--ease-out), background-color 140ms ease;
86 }
87
88 .btn:active:not(:disabled) {
89     transform: scale(0.97);
90 }
91
92 .btn:disabled {
93     opacity: 0.35;
94     cursor: default;
95 }
96
97 .btn:focus-visible,
98 .column:focus-visible {
99     outline: 2px solid var(--ink);
100     outline-offset: 2px;
101 }
102
103 .btn.primary {
104     background: var(--ink);
105     border-color: var(--ink);
106     color: var(--bg);
107 }
108
109 .btn.icon {
110     width: 52px;
111     padding: 0;
112     display: grid;
113     place-items: center;
114 }
115
116 .btn.icon .q-icon {
117     font-size: 26px;
118 }
119
120 @media (hover: hover) and (pointer: fine) {
121     .btn:hover:not(:disabled):not(.primary) {
122         background: var(--surface-hover);
123     }
124     .btn.primary:hover {
125         opacity: 0.9;
126     }
127 }
128
129 /* Level menu */
130
131 .menu {
132     justify-content: center;
133     gap: 40px;
134 }

```

```
135
136 .wordmark {
137     font-size: 34px;
138     font-weight: 700;
139     letter-spacing: -0.02em;
140     line-height: 1.1;
141     margin: 0;
142 }
143
144 .wordmark-sub {
145     color: var(--muted);
146     font-weight: 500;
147 }
148
149 .levels {
150     display: flex;
151     flex-direction: column;
152     gap: 10px;
153 }
154
155 /* Game: top bar and players */
156
157 .game {
158     gap: 20px;
159 }
160
161 .topbar {
162     display: flex;
163     align-items: center;
164     gap: 12px;
165 }
166
167 .topbar .back {
168     width: 40px;
169     height: 40px;
170     border-color: transparent;
171 }
172
173 .level-name {
174     font-weight: 600;
175     color: var(--muted);
176 }
177
178 .players {
179     display: grid;
180     grid-template-columns: 1fr 1fr;
181     gap: 10px;
182 }
183
184 .player {
185     display: flex;
186     align-items: center;
187     gap: 8px;
188     height: 52px;
189     padding: 0 14px;
190     border: 1px solid var(--line);
191     border-radius: 12px;
192     transition: border-color 160ms ease;
193 }
194
195 .player.active {
196     border-color: var(--ink);
197 }
198
199 .player .dot {
200     width: 14px;
201     height: 14px;
202     border-radius: 50%;
203     flex: none;
```

```
204 }
205
206 .player.p1 .dot { background: var(--red); }
207 .player.p2 .dot { background: var(--yellow); }
208
209 .player-name {
210     font-weight: 600;
211 }
212
213 .clock {
214     margin-left: auto;
215     font-weight: 650;
216     font-variant-numeric: tabular-nums;
217 }
218
219 .clock.low {
220     color: var(--danger);
221 }
222
223 .pips {
224     display: flex;
225     gap: 4px;
226     margin-left: auto;
227 }
228
229 .clock + .pips {
230     margin-left: 4px;
231 }
232
233 .pip {
234     width: 6px;
235     height: 6px;
236     border-radius: 50%;
237     background: var(--ink);
238 }
239
240 .pip.used {
241     background: var(--line);
242 }
243
244 .player.thinking .dot {
245     animation: pulse 900ms ease-in-out infinite;
246 }
247
248 @keyframes pulse {
249     50% { opacity: 0.3; }
250 }
251
252 /* Board */
253
254 .board {
255     display: grid;
256     grid-template-columns: repeat(6, 1fr);
257     gap: var(--gap);
258     padding: var(--gap);
259     background: var(--board);
260     border-radius: 16px;
261     aspect-ratio: 1;
262     width: 100%;
263 }
264
265 .column {
266     appearance: none;
267     border: 0;
268     padding: 0;
269     background: transparent;
270     display: grid;
271     grid-template-rows: repeat(6, 1fr);
272     gap: var(--gap);
```

```

273     border-radius: 999px;
274     cursor: pointer;
275     transition: background-color 120ms ease;
276 }
277
278 @media (hover: hover) and (pointer: fine) {
279     .column:hover {
280         background: var(--board-hover);
281     }
282 }
283
284 .hole {
285     position: relative;
286     border-radius: 50%;
287     background: var(--hole);
288     aspect-ratio: 1;
289 }
290
291 .disc {
292     position: absolute;
293     inset: 0;
294     border-radius: 50%;
295 }
296
297 .disc.p1 { background: var(--red); box-shadow: inset 0 0 0 3px var(--red-edge); }
298 .disc.p2 { background: var(--yellow); box-shadow: inset 0 0 0 3px var(--yellow-edge); }
299
300 /* One row = hole height + gap. */
301 .disc.falling {
302     animation: fall var(--fall-ms) var(--ease-gravity) both;
303 }
304
305 @keyframes fall {
306     from { transform: translateY(calc(-1 * var(--rows) * (100% + var(--gap)))); }
307 }
308
309 /* main.py redraws the turned board when this ends. */
310 .board.turn-R { animation: turn-right var(--rotate-ms) var(--ease-turn) forwards; }
311 .board.turn-L { animation: turn-left var(--rotate-ms) var(--ease-turn) forwards; }
312
313 @keyframes turn-right { to { transform: rotate(90deg); } }
314 @keyframes turn-left { to { transform: rotate(-90deg); } }
315
316 .disc.faded {
317     animation: fade 240ms ease forwards;
318 }
319
320 @keyframes fade {
321     to { opacity: 0.25; }
322 }
323
324 /* Controls and result */
325
326 .controls {
327     display: flex;
328     justify-content: center;
329     align-items: center;
330     flex-direction: column;
331     gap: 16px;
332     min-height: 52px;
333 }
334
335 .controls:has(.rotate) {
336     flex-direction: row;
337     gap: 24px;
338 }
339
340 .result {
341     font-size: 24px;

```

```
342     font-weight: 700;
343     letter-spacing: -0.01em;
344 }
345
346 .result-actions .btn {
347     justify-content: center;
348 }
349
350 .result-actions {
351     display: grid;
352     grid-template-columns: 1fr 1fr;
353     gap: 10px;
354     width: 100%;
355 }
356
357 /* Reduced motion */
358
359 @media (prefers-reduced-motion: reduce) {
360     .disc.falling,
361     .board.turn-R,
362     .board.turn-L,
363     .player.thinking .dot {
364         animation: none;
365     }
366     .btn,
367     .column {
368         transition: none;
369     }
370 }
```

## test\_engine.py

```
1  """Tests for engine.py and ai.py. Run: python test_engine.py"""
2
3  import random
4
5  import ai
6  import engine
7  from engine import AGENT, DRAW, EMPTY, HUMAN, index
8
9  SYMBOLS = {".": EMPTY, "X": HUMAN, "O": AGENT}
10
11
12 def board(*rows):
13     """6 strings, top row first. "." empty, "X" human, "O" agent."""
14     return tuple(SYMBOLS[ch] for row in rows for ch in row)
15
16
17 def state(rows, to_move=HUMAN, rotations_left=(3, 3)):
18     return (board(*rows), to_move, rotations_left)
19
20
21 def random_position(rng, moves):
22     """An unfinished game after up to `moves` random moves."""
23     position = engine.new_game()
24     for _ in range(moves):
25         next_position = engine.apply(position, rng.choice(engine.legal_moves(position)))
26         if engine.winner(next_position) is not None:
27             break
28         position = next_position
29     return position
30
31
32 # Rules
33
34 def test_new_game():
35     cells, to_move, rotations_left = engine.new_game()
36     assert cells == (EMPTY,) * 36
37     assert to_move == HUMAN
38     assert rotations_left == (3, 3)
39     assert engine.legal_moves(engine.new_game()) == [0, 1, 2, 3, 4, 5, "L", "R"]
40
41
42 def test_there_are_54_lines():
43     assert len(engine.LINES) == 54
44
45
46 def test_drop_lands_on_the_lowest_empty_cell():
47     position = engine.apply(engine.new_game(), 2)
48     assert position[0][index(5, 2)] == HUMAN
49     position = engine.apply(position, 2)
50     assert position[0][index(4, 2)] == AGENT
51     assert position[1] == HUMAN
52
53
54 def test_full_column_is_not_legal():
55     position = engine.new_game()
56     for _ in range(6):
57         position = engine.apply(position, 0)
58     assert 0 not in engine.legal_moves(position)
59
60
61 def test_rotate_right_then_left_gives_back_the_board():
62     rng = random.Random(1)
63     cells = tuple(rng.choice([EMPTY, HUMAN, AGENT]) for _ in range(36))
64     assert engine.rotate(engine.rotate(cells, "R"), "L") == cells
65
```

```

66
67 def test_rotate_right_turns_the_bottom_row_into_the_left_column():
68     position = state([".....", ".....", ".....", ".....", ".....", "XOX..."])
69     cells = engine.apply(position, "R")[0]
70     # Bottom row becomes column 0.
71     assert [cells[index(row, 0)] for row in (3, 4, 5)] == [HUMAN, AGENT, HUMAN]
72     assert sum(1 for c in cells if c != EMPTY) == 3
73
74
75 def test_rotate_left_turns_the_bottom_row_into_the_right_column():
76     position = state([".....", ".....", ".....", ".....", ".....", "XOX..."])
77     cells = engine.apply(position, "L")[0]
78     assert [cells[index(row, 5)] for row in (3, 4, 5)] == [HUMAN, AGENT, HUMAN]
79
80
81 def test_discs_fall_after_a_rotation():
82     position = state([".....", ".....", ".....", ".....", "X.....", "XX...."])
83     cells = engine.apply(position, "R")[0]
84     for col in range(6):
85         column = [cells[index(row, col)] for row in range(6)]
86         first_disc = next((row for row, c in enumerate(column) if c != EMPTY), 6)
87         assert all(c != EMPTY for c in column[first_disc:])
88
89
90 def test_rotation_uses_up_the_players_rotations():
91     position = engine.new_game()
92     for _ in range(3):
93         position = engine.apply(position, "R") # human rotates
94         position = engine.apply(position, 0) # agent drops
95     assert position[2] == (0, 3)
96     assert "L" not in engine.legal_moves(position)
97     assert "R" not in engine.legal_moves(position)
98
99
100 def test_four_in_a_row_in_every_direction():
101     horizontal = state([".....", ".....", ".....", ".....", ".....", "XXXX.."])
102     vertical = state([".....", ".....", "X.....", "X.....", "X.....", "X....."])
103     diagonal_down = state([".....", ".....", "X.....", ".X.....", "..X...", "...X.."])
104     diagonal_up = state([".....", ".....", "...X..", ".X...", ".X.....", "X....."])
105     for position in (horizontal, vertical, diagonal_down, diagonal_up):
106         assert engine.winner(position) == HUMAN
107
108
109 def test_rotation_can_make_the_other_player_win():
110     position = state([
111         ".....",
112         ".X....",
113         ".X....",
114         ".0.000",
115         ".0.XXX",
116         ".XX00X",
117     ])
118     assert engine.winner(position) is None
119     assert engine.winner(engine.apply(position, "L")) == AGENT
120
121
122 def test_rotation_that_gives_both_players_four_is_a_draw():
123     position = state([
124         ".....X",
125         ".....0",
126         ".X...0",
127         ".X0..X",
128         "X00..0",
129         "0X0XXX",
130     ])
131     assert engine.winner(position) is None
132     after = engine.apply(position, "R")
133     assert engine.four_in_a_row(after[0]) == {HUMAN, AGENT}
134     assert engine.winner(after) == DRAW

```

```

135
136
137 def test_full_board_without_four_is_a_draw():
138     position = state(["XX00XX", "00XX00", "XX00XX", "00XX00", "XX00XX", "00XX00"])
139     assert engine.winner(position) == DRAW
140
141
142 def test_game_in_progress_has_no_winner():
143     assert engine.winner(engine.new_game()) is None
144
145
146 # Agent
147
148 def test_alphabeta_gives_the_same_value_as_minimax():
149     rng = random.Random(7)
150     for _ in range(30):
151         position = random_position(rng, rng.randint(4, 12))
152         for search_rotations in (True, False):
153             search = ai.Search(position[1], search_rotations)
154             expected = search.minimax(position, 3, 0)
155             actual = search.alphabeta(position, 3, -ai.INFINITY, ai.INFINITY, 0)
156             assert actual == expected
157
158
159 def test_evaluation_stays_far_below_a_win():
160     rng = random.Random(3)
161     for _ in range(200):
162         cells = random_position(rng, 20)[0]
163         assert abs(ai.evaluate(cells, AGENT)) <= 270
164
165
166 def test_agent_takes_an_immediate_win():
167     position = state([".....", ".....", ".....", ".....", "XXX...", "000..."], to_move=AGENT)
168     for level in ("Hard", "Expert"):
169         assert ai.decide(position, level, time_limit=None).move == 3
170
171
172 def test_agent_blocks_an_immediate_loss():
173     position = state([".....", ".....", ".....", ".....", ".....", "XXX.0."], to_move=AGENT)
174     after = engine.apply(position, ai.decide(position, "Hard", time_limit=None).move)
175     # A drop or a rotation may block. The human must have no winning reply.
176     for reply in engine.legal_moves(after):
177         assert engine.winner(engine.apply(after, reply)) != HUMAN
178
179
180 def test_softmax_almost_never_picks_a_losing_move():
181     rng = random.Random(0)
182     picks = [ai.softmax_choice({0: 10, 1: -1000}, 20, rng) for _ in range(1000)]
183     assert picks.count(1) == 0
184
185
186 def test_softmax_picks_close_moves_often():
187     rng = random.Random(0)
188     picks = [ai.softmax_choice({0: 10, 1: 0}, 20, rng) for _ in range(2000)]
189     #  $P(1) = e^{-0.5} / (1 + e^{-0.5}) = 0.38$ 
190     assert 0.33 < picks.count(1) / len(picks) < 0.43
191
192
193 def test_low_levels_never_rotate():
194     rng = random.Random(5)
195     for _ in range(20):
196         position = random_position(rng, 6)
197         position = (position[0], AGENT, position[2])
198         for level in ("Beginner", "Easy", "Medium"):
199             assert ai.decide(position, level, time_limit=None, rng=rng).move not in
engine.ROTATIONS
200
201
202 def test_agent_answers_within_the_time_limit():

```

```
203     import time
204     position = random_position(random.Random(2), 10)
205     start = time.monotonic()
206     ai.choose(position, "Expert")
207     assert time.monotonic() - start < ai.TIME_LIMIT + 0.2
208
209
210 if __name__ == "__main__":
211     tests = [(name, f) for name, f in sorted(globals().items()) if name.startswith("test_")]
212     for name, test in tests:
213         test()
214         print("ok ", name)
215     print(f"{len(tests)} tests passed")
```