ChatGPT

Heuristic Tetris-Playing Agent: Features, Weights, and Search Strategy

Overview and Context

Heuristic-based Tetris agents rely on a hand-crafted evaluation function to choose moves, rather than any neural network or reinforcement learning. A recent study by Bairaktaris & Johannssen (2025) compared such a heuristic agent to modern RL algorithms and found the heuristic approach can **outperform deep RL** in both score and efficiency (the heuristic agent achieved higher scores in Atari Tetris while using far less computation) [1]. This underscores that a well-designed heuristic strategy – using carefully chosen features and weights – remains highly effective in Tetris. Below, we detail the key components of a successful pure heuristic Tetris agent, including its evaluation function, feature set, weight tuning, model structure, and move search method, with pseudocode and examples for clarity.

Heuristic Evaluation Function and Features

At the heart of the agent is a **linear evaluation function** that assigns a score to a given board state. The function is a weighted sum of **hand-crafted features** that capture undesirable or desirable properties of the Tetris grid. For each possible move (placement of a piece), the agent simulates the resulting board and computes this score; the move with the best (e.g. highest) score is chosen. In a canonical heuristic Tetris AI, the evaluation $f(\mathbf{x})$ looks like:

$$f(\mathbf{x}) \;=\; a\cdot(\mathrm{AggregateHeight}) \;+\; b\cdot(\mathrm{CompleteLines}) \;+\; c\cdot(\mathrm{Holes}) \;+\; d\cdot(\mathrm{Bumpiness}) \;+\; \cdots$$

Here $\mathbf{x}$ represents the feature vector of the resulting board state [2] [3]. The four primary features (with their intuitive meaning) are:

- **Aggregate Height** – the sum of the heights of all columns on the board (height = number of cells from the bottom up to the highest filled cell in that column). This measures how "high" the pile of pieces is in total [4]. Lower aggregate height is better since a lower stack means more space available before reaching the top. The agent will try to **minimize** this value.
- **Complete Lines** – the number of full lines completed (and ready to clear) in the board. Higher is better, as clearing lines is the goal to keep the board empty. The agent will try to **maximize** line clears [5]. (Often the immediate *reward* for a move can include the number of lines cleared by that piece.)
- **Holes** – the count of empty cells that have at least one filled cell above them in the same column. These "holes" are problematic because they cannot be filled unless the lines above them are cleared [6]. Fewer holes is better, so the agent minimizes this count.
- **Bumpiness** – a measure of how uneven or "jagged" the surface of the board is. Typically defined as the sum of absolute height differences between adjacent columns [7]. A flatter top (low bumpiness) is preferable because tall isolated columns create deep wells that are hard to fill. The agent minimizes bumpiness.

These four features – **aggregate height, completed lines, holes, and bumpiness** – form a common basis for many Tetris heuristic bots [8]. Intuitively, a good move is one that keeps the pile low and even, creates or retains cleared lines, and avoids burying holes. Each feature corresponds to a simple count derived from the board, making them fast to compute. The evaluation function combines them linearly with coefficients (a, b, c, d) that reflect the relative importance of each factor.

Additional features: Some advanced heuristic agents include more features beyond the four above. A classic example is the *Dellacherie* strategy, which uses **six features** [9]:

- *Landing Height* (the row where the placed piece lands – lower is better),
- *Rows with Holes* or *Row Transitions* (count of transitions from empty to filled in each row, penalizing gaps in rows),
- *Column Transitions* (transitions from empty to filled in each column, penalizing rugged vertical profile),
- *Cumulative Wells* (the total "well depth" – deep vertical gaps – across the board),
- *Holes* (as defined above), and
- *Eroded Cells* (the number of cells cleared by the current line clears, i.e. cells removed in completed lines).

Dellacherie's evaluation function was famously given as:

$$\text{Score} = -4 \times (\text{Holes}) \; - \; (\text{CumulativeWells}) \; - \; (\text{RowTrans}) \; - \; (\text{ColTrans}) \; - \; (\text{LandingHeight}) \; + \; (\text{ErodedCell}$$

All terms are to be *minimized* except eroded cells which is rewarded [10]. In other words, this scheme heavily penalizes holes (4× weight) and also penalizes any kind of unevenness or height, while slightly rewarding line-clear cells. Despite using more features, the concept is the same: a linear combination scoring function to evaluate board goodness.

Weight Coefficients and Tuning

The coefficients a, b, c, d (and any additional feature weights) are critical to the agent's performance. They determine how strongly the AI prefers clearing lines versus avoiding holes, etc. **Exact weight values** have been found in literature either by manual tuning or by automated search:

- **Manual tuning:** In the early era, weights were often set by informed guesswork. For example, Pierre Dellacherie (as reported by Fahey 2003) chose the six-feature weights by **trial and error**, adjusting them until the agent performed well [9]. This yielded the Dellacherie weights implicitly shown above (e.g. holes had a weight of -4, indicating a hole is four times worse than most other feature increments). Despite being manually set, these weights proved remarkably effective – Dellacherie's hand-crafted agent was, for a time, the best Tetris player (clearing on the order of **hundreds of thousands of lines** per game in the standard 20×10 grid).

- **Genetic algorithms (GA) and evolutionary search:** More recently, researchers and hobbyists have used GAs to *learn* optimal weights. A GA treats the weight vector as a "chromosome" and evolves a population of Tetris players by repeatedly mutating and recombining the weights, keeping those that achieve higher line clears in simulation [11] [12]. This approach was popularized because it requires no domain knowledge beyond defining the features and a fitness metric (e.g. total lines

cleared). One well-known result from such a GA optimization is the weight set:
$$a = -0.510066,\quad b = +0.760666,\quad c = -0.35663,\quad d = -0.184483,$$
for the four features [AggregateHeight, CompleteLines, Holes, Bumpiness] respectively [13]. These *GA-discovered* weights have become a de facto standard because they allow an agent to play **extremely well** – in fact, near *indefinitely*. Using these weights in the linear evaluation, the AI strongly prioritizes clearing lines and reducing height, while giving moderate penalties to holes and bumpiness. This particular weight combination was evolved by Y. Lee (2013) using a GA and has been widely cited and re-used [14] [15]. With this weighted evaluation, the agent can clear millions of lines without topping out, essentially playing forever under ideal conditions. For instance, Lee's AI could run for weeks, only ending due to external stops, not game over [16]. Academic studies have similarly found high-performing weights via evolutionary algorithms; *Böhm et al.* (2005) evolved a controller that achieved **≈480 million lines cleared** (on a 20×10 board with two-piece lookahead) using a linear feature weight approach [17] – an astronomical score indicating the game can be prolonged almost endlessly with a good heuristic policy.

Other optimization methods have also been applied to tune heuristic weights (e.g. *cross-entropy method*, *particle swarm*, *harmony search*, etc.), all aiming to maximize lines cleared by adjusting weight values. In some cases, weight optimization is treated as a reinforcement learning problem too (seeking weights that maximize long-term reward), but importantly the **final agent is still a fixed weighted heuristic**, not a complex network. In summary, whether by human intuition or genetic evolution, the weights are set to specific constants before play. There is **no online learning** happening during gameplay – the agent always evaluates states with the same fixed weighted formula.

Example: To concretize, using the GA-optimal weights above, the evaluation function would be:

$$f = -0.510066 \cdot (\text{AggregateHeight}) \; + \; 0.760666 \cdot (\text{CompleteLines}) \; - \; 0.35663 \cdot (\text{Holes}) \; - \; 0.184483 \cdot (\text{Bumpiness}$$

If a potential move results in a board with (say) aggregate height = 10, complete lines = 1, holes = 2, bumpiness = 3, the score would be:

$$f = -0.510066(10) + 0.760666(1) - 0.35663(2) - 0.184483(3) \approx -5.1007 + 0.7607 - 0.7133 - 0.55345 = -5.6068.$$

Lower (more negative) scores here indicate a worse outcome; a different move yielding a higher score (less negative or positive) would be preferred. Because **lines cleared** has a relatively large positive weight, a move that clears a line often outweighs moderate increases in height or bumpiness. Meanwhile, even a single **hole** (negative weight -0.35663) can significantly drag down the score, which reflects the heuristic's strategy to avoid holes almost at all costs.

Model Structure (Heuristic Policy Architecture)

The structure of this agent is exceedingly simple compared to neural network models – there are **no layers or neurons**. The policy is essentially a linear function (one could think of it as a single-layer perceptron with fixed weights, but without any hidden units). The input to the function is a game state (typically represented by the heights of columns or occupancy of cells), which is then mapped to a set of feature values. These feature values are multiplied by the predetermined weights and summed to produce a single scalar *score*. This score represents the heuristic's estimate of the "goodness" of that state.

Because it's a linear combination, the model is **interpretable**: each feature's contribution is additive. For example, you can directly see how much penalty a hole adds or how much reward a line clear gives in the score. There is no learned non-linear interaction between features as you'd have in a neural network. This lack of complexity means the model **does not generalize beyond the features provided** – it can't "invent" new considerations – but the chosen features are usually sufficient to capture what makes a Tetris position promising or dangerous.

In summary, the agent consists of **a fixed scoring function** with a small number of inputs. There is no internal state apart from the current board features, and no adaptive learning happening during play. The simplicity of this structure makes it extremely fast to evaluate (just a few multiplications and additions per move) and easy to debug.

Move Selection and Search Method

While the evaluation function tells us how good a single board configuration is, the agent still needs to **search for the best move** (i.e. how to place the current falling tetromino). At each turn, the agent considers all possible placements of the current piece and uses the heuristic to decide which placement yields the highest score. The search process works as follows:

1. **Enumerate possible moves:** For the current tetromino, generate all legal final positions and orientations it can be dropped in. This involves trying each orientation (rotation) and each feasible horizontal position (column) where the piece would come to rest on the stack or floor. For a standard Tetris piece set, this could be on the order of 10–40 possible placements to consider (depending on piece shape and board width). The agent essentially "simulates" or imagines each move outcome. [18]

2. **Evaluate each outcome:** For each possible placement, the agent computes the resulting board configuration (how the grid will look after the piece locks down and any full lines are cleared). It then calculates the heuristic score f for that resulting board using the weighted features described earlier. This yields a score for each move option [8].

3. **Select the best move:** The move with the best score (typically the **highest** score, if higher means better in the chosen formulation) is selected as the action to take. That piece is then dropped in the chosen position.

4. **Repeat for the next piece:** After the move, the next tetromino becomes the current piece and the cycle repeats (back to step 1).

This one-ply search (lookahead of 0, or just evaluating immediate results) constitutes a **greedy strategy** – the agent optimizes the immediate next board state according to the heuristic. In many implementations, however, the agent takes advantage of the known **next piece preview** (Tetris typically shows the next piece in advance). This allows a modest lookahead of depth 1: the agent can effectively evaluate a sequence of two moves (place current piece, then place the next piece). In practice, this is done by *folding the next piece's outcome into the evaluation*: for each candidate placement of the current piece, the agent can also simulate the drop of the next piece (assuming optimal placement of that next piece) and evaluate the resulting board after two moves. The score of the sequence can be used to choose the current move. This procedure

amounts to a depth-2 search (current and next piece), often implemented efficiently by simply taking the max over next-piece placements internally. Many heuristic Tetris AIs include this one-piece lookahead because it significantly improves performance by avoiding shortsighted moves [18].

Pseudocode: Greedy move selection with one-piece lookahead (using our heuristic function `Score(board)`):

```
function choose_move(current_piece, next_piece, board_state):
    best_score = -∞
    best_move = None
    for each orientation O of current_piece:
        for each column X where piece can drop without collision:
            # 1. Simulate dropping current_piece at orientation O in column X
            new_board = simulate_drop(board_state, current_piece, O, X)
            clear_full_lines(new_board)  # apply line clears if any
            if next_piece is given:
                # Lookahead: consider best placement of the next piece as well
                best_next_score = -∞
                for each orientation O2 of next_piece:
                    for each column Y for next_piece:
                        board_after_next = simulate_drop(new_board, next_piece,
O2, Y)
                        clear_full_lines(board_after_next)
                        score = Score(board_after_next)  # evaluate after two
moves
                        best_next_score = max(best_next_score, score)
                total_score = best_next_score  # use the best achievable score
with optimal next move
            else:
                # No lookahead, just evaluate current move result
                total_score = Score(new_board)
            if total_score > best_score:
                best_score = total_score
                best_move = (O, X)
    return best_move
```

In the above pseudocode, the agent exhaustively tries every possible drop for the current piece (orientation `O` and column `X`). If a next piece is known, it then tries every placement of that next piece on the resulting board (`Y` and `O2`) to find the best possible follow-up score. The current move is then scored according to that best possible future outcome (this is effectively a one-step lookahead or a depth-2 mini search). The move leading to the highest score is chosen. Without a next-piece preview, the inner loop over the next piece is omitted and the agent simply uses `Score(new_board)` for the current move outcome.

Search complexity: This greedy or one-lookahead search is quite fast. There are at most ~40 placements for the current piece and ~40 for the next, leading to on the order of $40 \times 40 = 1600$ evaluations in the worst case – easily done in a fraction of a second in modern hardware. This is why heuristic agents are

extremely efficient: they can simulate and evaluate thousands of moves per second using the lightweight scoring function. In contrast, a deep RL agent would require a neural network forward-pass for each evaluation, which is slower.

For **even stronger performance**, researchers have experimented with deeper search (looking ahead multiple pieces) using techniques like **beam search** or brute-force search with pruning. In beam search, for example, the agent would keep the top k candidate sequences after evaluating each piece and explore several pieces into the future. Each additional piece considered multiplies the branching factor, so the search space grows exponentially; thus, a limited beam width or depth is used to keep it tractable. Some implementations allow an adjustable lookahead depth N (where $N=1$ is just the next piece, $N=2$ would consider two pieces ahead, etc.) [15]. Increasing lookahead generally improves the quality of decisions (since the agent can foresee bad outcomes further out), at the cost of computation. Notably, the world-record-holding Tetris heuristic controllers have indeed leveraged multi-piece lookahead. For instance, one controller using **two-piece lookahead and an evolved evaluation function** was able to clear hundreds of millions of lines as noted earlier [17]. However, even with just a one-piece lookahead, the heuristic agent described is remarkably strong and far outperforms casual human play or naive algorithms.

Source Code and Implementation

The simplicity of the heuristic approach means implementations are straightforward. Many open-source Tetris AI projects have been built around these ideas. For example, Lee's "(Near) Perfect Tetris AI" provides a JavaScript implementation and an online demo [16], and other developers have published their versions in Python, C++, Haskell, etc., often using the exact feature set and weights described above [15]. The core logic (enumerating moves and evaluating via the linear function) is typically under a few hundred lines of code. The pseudocode provided in the previous section can be used as a reference for implementing the move selection.

In summary, a pure heuristic Tetris-playing agent consists of: *(a)* a **scoring function** with hand-crafted features (e.g. height, lines, holes, bumpiness) and fixed weights (tuned manually or via genetic algorithms), and *(b)* a **search procedure** that simulates possible piece placements and uses the scoring function to greedily pick the best move. This approach has proven highly successful – such agents can play indefinitely under ideal conditions and serve as strong benchmarks. The 2025 study "Outsmarting Algorithms" confirmed that even against modern RL agents, a well-tuned heuristic policy is formidable [1]. By focusing on clear-cut board metrics and optimizing their weights, the heuristic Tetris agent exemplifies how expert knowledge and simple algorithms can "outsmart" more complex learning-based methods in certain structured tasks.

Sources:

- Bairaktaris, J.A., & Johannssen, A. (2025). *Outsmarting algorithms: A comparative battle between Reinforcement Learning and heuristics in Atari Tetris*. Expert Systems with Applications, **277**, 127251. (Heuristic agent vs. RL performance) [1]
- Lee, Y. (2013). *Tetris AI – The (Near) Perfect Player*. *(Blog post & implementation)* – Describes a heuristic Tetris AI with four features and GA-tuned weights [8] [13] [18].
- Fahey, C. (2003). *Tetris AI analysis* – Analysis of Pierre Dellacherie's heuristic strategy (six features with manually tuned weights) [9] [10].

- Böhm, H. et al. (2005). *Evolutionary Algorithm for Tetris* – Demonstrated extremely high line clears using a two-piece lookahead and evolved weights [17].
- Gilliland, T., & Zhang, D. (2020). **Parallel Greedy Tetris Solver** *(Project report)* – Implemented a multi-threaded Tetris AI using Lee's heuristic and confirms weight values [15].

---

[1] Outsmarting algorithms: : A comparative battle between ...
https://dl.acm.org/doi/10.1016/j.eswa.2025.127251

[2] [3] [4] [5] [6] [7] [8] [13] [14] [16] [18] Tetris AI – The (Near) Perfect Bot | Code My Road
https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

[9] [10] [17] The Game of Tetris in Machine Learning
https://arxiv.org/pdf/1905.01652

[11] [12] GitHub - takado8/Tetris: Heuristic AI for playing tetris
https://github.com/takado8/Tetris

[15] cs.columbia.edu
https://www.cs.columbia.edu/~sedwards/classes/2020/4995-fall/reports/Tetris.pdf