Tetris Bot Project Report

Heuristic-based Search for Optimal Tetris Piece Placement

Anthony Medico

Introduction

The aim of this project was to experiment with heuristic functions by creating a bot that plays the game of Tetris. The main challenge with Tetris is that knowing what an optimal placement looks like is challenging, and with the randomness of the pieces that drop, and the many different possible snapshots of a board state, it presents an interesting opportunity to experiment with different heuristics. As part of the project, I created a Tetris API game that will handle all game logic, providing the bot with the current game state for each move, await the bot's choice, and display the game graphically, including statistics like score, lines cleared, and average points per line cleared.

The bot receives the game board as a matrix along with the current and next piece for each move. With this information it uses both pieces to determine the best placement for the current piece. This is achieved with a weighted heuristic function with dynamically changing weights based on the board state. The bot has two modes: scoring and recovery, which changes based on how full the board currently is. This allows the bot to work towards high scoring by setting itself up for Tetris's when it's safe to do so, and if the board starts to fill up, it prioritizes clearing lines as fast as possible.

Project Setup

The project uses Pycharm for both the game and the bot. There are a total of three python files:

- `tetris.py`
    - The Tetris game API made using Pygame
    - To install package, use command `pip install pygame`
    - Controls all game logic, GUI display, interactivity
    - Creates a new bot and sends it game information
- `piece.py`
    - Custom Piece class used to create piece objects
    - Attributes of a piece are:

            - `type`
                - The letter representing the type (shape) piece
                - Possible representations: I, O, T, L, J, S, Z
            - `shape`
                - A rectangular matrix representing the actual shape
                - A cell with value 1 is a block, 0 is no block
            - `colour`
                - Each piece type has its own colour which is a tuple containing the respective RGB value
            - `col`
                - Defines which column the piece is placed on the board
                - For both row and col, this is where the top left cell of the shape is placed on the board
            - `row`
                - Defines which row the piece is placed on the board
                - For both row and col, this is where the top left cell of the shape is placed on the board
            - `orientation`
                - The current rotation of the piece
                - A value between 0 and 3 representing each 90-degree rotation of a piece
    - Functions of the piece class:
        - `rotate()`
            - Rotates the piece clockwise 90 degrees
        - `set_orientation(new_orientation)`
            - Directly sets the piece to a specified rotation
- `bot.py`
    - The bot class to handle choosing a move
    - Uses a dynamically weighted heuristic function to calculate a score for each piece, and returns the move with the highest score
    - Considers the `next` piece in its decision
        - First finds all valid moves using an efficient recursive algorithm to check all possible paths a piece can take
            - Marks paths already checked to avoid any repetition
        - For each valid move, it will then calculate all valid moves for the `next` piece, and evaluate the board with each of those by calculating all

        parameters needed for the heuristic function, followed by the function itself to get a score
        - Keeps track of the highest scoring move and returns it after exhausting the search space
        - Search space was trivial, so no pruning was added

Running the Game

- To run the game, open a terminal in the directory containing the three Python files
- Install Pygame (if needed): `pip install pygame`
- Run tetris.py: `python3 tetris.py`

Game Controls and Display

Upon launching the game you have the following options:

- "Next move" button:
    - Manually trigger the next move (disabled if autoplay is on)
    - Good for taking as much time as you need between moves to evaluate the bot's moves
- Autoplay
    - Toggled with the spacebar
    - Allows the game to run continuously
    - Delay is set at 500ms for the bot move
    - Additional 300ms delay before line clears
    - Can change these values in the code (lines 246 and 247 in `tetris.py`)
- Quit Game
    - Press Q at any time to quit the game
- Restart Game
    - Press R at any time to reset the game (even on "Game Over" screen)

The game also displays various information:

- Autoplay status: ON or OFF
- Bot mode: "Scoring" or "Recovery"
- Stats:

    - Total score
    - Total lines cleared
    - Average score per line cleared
        - See Scoring section under Game Rules

Game Rules

Known information for each move:

- Full game board
- Current piece
- Next piece

Piece generation:

- Completely random

Piece placement:

- Pieces can be nudged right or left to ‘fit’ under other pieces if there is room
- Needs to be a fully valid path to any piece placement

Game over:

- If a new piece enters the board and immediately causes a collision, it’s game over

Scoring:

- Score simply given depending how many lines cleared
- No levels or successive clears bonuses
- There is a bonus for points per line cleared which grows with more lines cleared
- Points:
    - 1 line: 40 points (40 points per line)
    - 2 lines: 100 points (50 points per line)
    - 3 lines: 300 points (100 points per line)
    - 4 lines: 1200 points (300 points per line)
- Encourages player (bot) to aim for Tetris’s (four-line clears)
- The closer the average points per line is to 300, the more four-line clears are happening

Heuristic Function and Parameters

The following parameters were used when evaluating the board:

- Number of holes
    - A hole is defined as any empty cell which has a block in any cell above it (in the same column)
    - Calculated by counting the number of holes in total on the board
    - Generally needs a high weight value since different piece placements typically only differ by 1 or 2 new holes (unless really bad move)
    - Lower is better
    - Influences bot to not create new holes
- Board evenness
    - A measure of how flat or bumpy the board is
    - Calculated by summing the height differences between adjacent columns
    - Lower is better
    - Influences bot to keep board even and make use of empty space lower on the board, instead of building up certain columns higher than others
- Board fill
    - A general measure of how full the board is
    - Calculated by summing the heights of each column
    - Lower is better
    - Influences bot to nudge pieces under other pieces if possible and to use a flatter orientation
- Lines cleared
    - Number of lines cleared
    - Always a value between zero and four so needs high weight values
    - Separated into four different parameters for 1, 2, 3 or 4 lines cleared so each can have its own weight
        - This is to facilitate scoring mode, where you would want to avoid 1, 2, or 3 line clears to allow building up to a four-line clear
        - These weights can be changed in recovery mode to give higher scores to lower line clears again
    - Higher is better
- Last column
    - The way scoring mode is handled is to influence the bot to leave the last column empty, which leaves space for dropping in an “I” piece for a Tetris
    - Calculated by checking how many blocks are in the last column

    - Uses a very high weight in scoring mode, and a weight of zero in recovery mode
    - Lower is better
- Danger zone
    - I high value is given for a move placed near the top center of the board, which is where the following new piece appears
    - This is the danger zone and should be avoided at all costs
    - If a piece is in the danger zone, a Boolean called danger is set to True, otherwise it is False, and the weight is multiplied by this Boolean, which Python interprets as 0 or 1 in the equation
    - Uses a very high weight

Weights:

- Each parameter above is given a weight to affect how much influence it has on the overall heuristic function
- Some parameters naturally produced higher values and generally didn't need a high weight, while others produced small values and needed high weights to first boost their influence, and then tweaked from there

How were weights calculated?

- For the purposes of this project, weights were chosen empirically, observing the bot behaviour and adjusting weights accordingly

Final Heuristic Function:

- Scoring Mode:
    - 1 line cleared: -100
    - 2 lines cleared: -80
    - 3 lines cleared: -50
    - 4 lines cleared: +10000
    - Holes: -(50 * number of holes)
    - Evenness: -(sum of absolute height differences of adjacent columns)
    - Board fill: -(sum of all column heights)
    - Last column: -(1000 * number of blocks in last column)
    - Danger: -1000 if in danger zone
- Recovery Mode:
    - 1 line cleared: +100
    - 2 lines cleared: +400
    - 3 lines cleared: +900

- 4 lines cleared: +10000
- Holes: -(50 * number of holes)
- Evenness: -(sum of absolute height differences of adjacent columns)
- Board fill: -(sum of all column heights)
- Last column: -(0 * number of blocks in last column)
- Danger: -1000 if in danger zone

Results

Results for 10 runs:

Run	Score	Lines Cleared	Average Points
1	27200	288	94
2	5680	60	94
3	6400	59	108
4	6440	56	115
5	12120	98	123
6	38620	330	117
7	2780	24	115
8	1580	12	131
9	5480	60	91
10	36740	315	116

From the table above, results vary quite a bit which shows that the bot doesn't play well in general, and it depends on the luck of the pieces. Improvements can certainly be made by either further adjust weights or add new and potentially more complex parameters to the function. It seems "scoring" mode can get the bot in trouble which then depends on the luck of the following series of pieces to recover.

Another interesting observation is the visual proof of the bot taking the `next` piece into account. Some piece placements look like a bad placement or an error at first, but then the next piece gets tucked in with it nicely.

I will certainly continue to tinker with this project and build on it in the future.