Optimizing Tetris AI with an Asynchronous Particle Swarm

Todd O. Gaunt
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
toddgaunt@protonmail.ch
May 12, 2018

Abstract

Tetris is a classic computer game with indefinite playtime and limited piece lookahead. This requires a real-time algorithm with the ability to evaluate the value of a given game-state. Real time depth-limited DFS works well on the limited state-space of Tetris, since all nodes can be visited and evaluated accordingly. Beyond a depth of two, however, the amount of states generated becomes excessive and slows down the search noticeably. Since there is only limited lookahead with a depth of two, the static evaluation function has little room for error to allow for indefinite play. To discover the optimal parameters to use for the static evaluation function, an Asynchronous Particle Swarm Optimization search was performed on the game of tetris. A description of the particle swarm algorithmn used, as well as the techniques used during the depth-first search to allow for fast state-space searches in real-time, are described in this paper.

Introduction

In real-time search problems where there is not enough time to compute the path to the goal state, a state evaluation function must be used over a goal specification function in order to be able to return paths to states that will potentially lead to the goal. State evaluation functions compute a score for the state it's evaluating utilizing problem-specific heuristics, and assign a weight to each heuristic according to its importance. Sometimes appropriate weights can be found through trial and error by hand. Nevertheless, when the set of heuristics grows this can become a tedious and time consuming that may not find weights even close to optimal.

A way to automate this task is to use an algorithm that can explore a function to discover the set of inputs that results in a minimum or maximum value of the function. Particle Swarm Optimization is one algorithm that can do such a task, and is the one chosen for this paper. The main idea of the algorithm is to have n particles be uniformly randomly distributed over an initial range in the function input space, these positions being heuristic weights for a static evaluation function in this case. Then the algorithm iterates over every particle, changing its velocity based on its own past position, its own best discovered position, and the best discovered position of the entire swarm until the maximum number of iterations given has been achieved.

The game of Tetris is a classic computer game that can utilize a real-time search algorithm that can be optimized via Particle Swarm Optimization. A Particle Swarm was suitible for Tetris as the algorithm is fairly straightforward to implement asynchronously, which is important since the fitness function used by the Tetris AI can take a long time to evaluate, seeing as the function produces higher scores the longer the AI is able to play Tetris.

Domain: Hidamari, a Tetris Implementation

Tetris is a classic digital game originally made in the 1980s as a single player puzzle game. The game is played on a 10x20 grid with seven possible pieces for the player to control. Each piece comes from the top of the field one at a time, constantly falling at a fixed rate. The goal is to stack the pieces in such a way that an entire row of 10 filled in tiles is made. This row is then cleared, score is rewarded to the player, and finally any blocks above the cleared row are shifted down by one block.

In order to aid the player, there is usually at least one preview piece shown to the player. This preview piece is the next piece that will be spawned by the game for the player to control once the current falling piece locks into place. Given this information, a player can move the current falling piece into place with plans for the next piece in mind. This allows for more optimal moves to be made, which in turn gives the player an opportunity to plan for either achieving higher scores or to arrange the board in such a way that promotes long-term survival. This is what motivated the development of an AI that can play a game of indefinite length with an opportunity to plan within real-time.

The goal of the game has not always been standardized, but getting as high a score as possible while playing for as long as possible is a common one. For the purposes of this project, an AI that can clear 100,000 lines in three consecutive games is considered a success. The number 100,000 lines is chosen as that would mean at the very

least 100,000,000 points would be scored under the standard Tetris scoring system, where even pro players have difficulty getting past 1,000,000. The AI must clear 100,000 lines three games in a row to provide evidence that it can play to the maximum score of 100,000 consistently.

Hidamari, the Tetris implementation written for this project, attempts to follow the Standard Tetris Guidelines as closely as possible, with the goal of the player to clear as many lines and get as high a score as possible before losing. The rules here are similar to any modern Tetris implementation, but the implementation is very lightweight. The playfield is represented as a bit board, an array of 23 16-bit integers. Each integer represents a single row of the board, and in turn each bit represents a single block on the playfield. This method of representing the game reduces most manipulations of the game board into simple bit-wise operations and shifts. This speed allowed for the particle swarm and the AI to simulate the game in-memory very quickly, speeding up the optimization and not causing any pausing in-game respectively.

The goal of this project was to create an AI that would be able to play the game of Tetris without consuming to many computational resources. While the depth-first search used very little processing power, and appeared to work well even with heuristic weights derived from manual guesswork, it couldn't play for much longer than 30 minutes consistently. To rectify this, different methods of function optimization were researched, and particle swarm optimization was chosen. The simplicity of the algorithm, and the ease of making the algorithm perform in parallel were the main considerations for choosing it. Parallelism was essential, as simulating long-running Tetris games in memory requires a lot of computational power, and the server the particle swarm ran in had four cores to be utilized.

Depth First Search with Static Evaluator

The backbone of the AI written for Hidamari is a simple depth-first search through the state-space of the game. A few simplifications were made for the state-space to conserve memory, and a depth limit of two is used for the search. The simplifications that were made to the game were to only consider the actions of moving and rotating the piece before locking it in the place at the bottom of the playfield. This way, only a single state is needed to represent all movements required to lock a given piece into the playfield, rather than one state per move. With the state-space simplified to be so few nodes, and with a branching factor of 36, the depth-first search is able to exhaustively evaluate all 1296 nodes of the depth-two before even a single frame of the game is rendered to find the best possible state.

The static evaluation function used by the depth-first search utilizes three distinct heuristics, as shown in figure 1, to compute a score for any given state. The higher the score the worse the state is, with the ideal score being exactly 0. Each heuristic is given a weight multiplier, which decides the relative importance of each heuristic. This is what determines how well the AI plays.

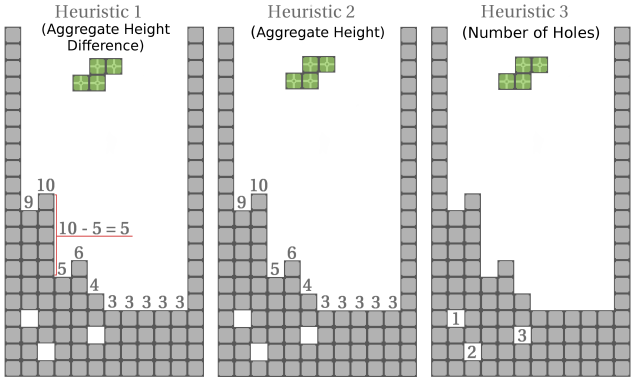

Figure 1: Visualization of heuristics

The first heuristic computes the aggregate height difference between all columns. The reason being that difference in column height, or playfield bumpiness, is a negative attribute as most pieces used in the game do not slot into bumpy boards very well. The second heuristic used is to compute the aggregate height of all columns on the playfield. This negative attribute should be avoided to prevent a game over, encouraging overall low column height. The third and final heuristic is to count the number of "holes" in the playfield. A "hole" is defined as any empty tile with a filled tile above it. Empty tiles with an opening to the left or right are still counted as holes in this way, since the AI cannot move pieces down and then to the side, only to the side and then down. Holes are an unattractive feature, as they require clearing all lines above them in order to remove them from the board. Minimizing holes is important for optimal play.

Asynchronous Particle Swarm Optimization

Particle Swarm Optimization is a function optimization algorithm that is used to find either the maxima or minima of a function. The inputs of a function are represented as spatial dimensions of a particle that has both a position and a velocity. Every iteration of the algorithm, the particles' positions are updated along with their velocities. The velocity update is done with respect to the rest of the swarm and the particles' own knowledge of past positions. This allows the swarm to eventually converge towards a maximum or minimum value of the function, while also engaging in individual exploration to influence the swarm.

To parallelize the algorithm, an atomically accessed priority queue containing the swarm's particles prioritized by the number of iterations performed is shared among the threads used to process the particles. The job of each thread con-

sists of popping a particle off the atomic queue, updating its velocity and position, potentially updating the particle's local best position, and then potentially atomically updating the swarm's global best position. After processing that particle, if the particle still had iterations left to complete, the thread puts the updated particle back into the queue with an incremented iteration count. Otherwise, the thread does not put the particle back. This process is repeated until there are no more particles to be popped off the queue.

Pseudo Code and Explanation

```
0  define apso()
1  {
2    let n be the number of threads
3    let s be the swarm of particles
4    let pq be the atomic priority queue
5    let bu be the upper bound of the search
6    let bl be the lower bound of the search
7
8    for each particle in s {
9      // Note that v, x, and p are
10     // vectors of the same dimensionality
11     // v is the particle velocity
12     // x is the particle position
13     // p is the best particle position
14     // g is the best swarm position
15
16     let v := U(-|bl - bu|, |bl - bu|)
17     let x := U(bl, bu)
18     let p := x
19
20     if it is the first particle
21       s.best_position := p
22     push(pq, particle(1, v, x, p))
23   }
24
25   for i in 0..(n - 1)
26     thread_spawn(lambda () {work(s, pq);})
27
28   work(s, pq)
29
30   for i in 0..(n - 1)
31     thread_join()
32
33   return s.best_position
34 }
```

The "apso" procedure begins by initializing the swarm of particles with uniformly distributed randomized vectors for velocity and position on lines 16 through 18. Each particle is then queued into the priority queue on line 22. After initialization is complete, $n - 1$ work threads are started to begin processing the particle queue. The main thread then also begins to perform work after the other threads have been spawned. Finally once all the particles have been processed a sufficient number of iterations, the main thread joins all of the threads and returns the best position vector found by the swarm. If only one thread is specified, the algorithm behaves similarly to the serial version, albeit slower because of the additional overhead to guarantee no detrimental data races occur.

```
0  define work(s, pq)
1  {
2    s is the swarm of particles
3    pq is the atomic priority queue
4
5    loop {
6      let particle := pop(pq)
7      if NONE = particle
8        break
9
10     let i := particle.iteration
11     let v := particle.velocity
12     let x := particle.position
13     let p := particle.best_position
14     let g := atomic_load(s.best_position)
15
16     // Update the velocity
17     let rp := U(0.0, 1.0)
18     let rg := U(0.0, 1.0)
19     let v2 := PHI * v
20             + ALPHA * rp * |p - x|
21             + BETA * rg + |g - x|
22
23     // Update the particle's position
24     let x2 := x + v
25
26     // Evaluate the fitness
27     if f(x2) > f(p)
28       let p2 := x2
29       if f(p2) > f(g)
30         atomic_store(s.best_position, p2)
31
32     if i < MAXIMUM_ITERATIONS
33       push(pq, particle(i + 1, v2, x2, p2))
34   }
35 }
```

Each thread in the apso algorithm uses the "work" function defined above. An infinite loop begins on line 7 which will only be broken out of once the work queue has no more particles to be processed, terminating the thread. If there is a particle to be processed after atomically popping it out of the queue, its position and velocity are updated according to lines 17 through 24. The constants PHI, ALPHA, and BETA control how much inertia, the past local best position, and the past swarm best position influence the

updated velocity. If the fitness of the updated position is greater than the particle's last known best position's fitness, the position is saved as the particle's last known best position. The same is then done to the swarm's last known best position on lines 27 through 30. At the end of the loop, on lines 32 and 33, the updated particle is pushed back onto the queue with an increased iteration count if and only if its iteration count is less than MAXIMUM_ITERATIONS.

Evaluation and Results

The goal stated for this project was to achieve a Tetris AI that could play at least three games to 100,000 points in a row. The particle swarm was run on Hidamari, the Tetris implementation used for this project, for 4 days with 50 particles over 100 iterations. The parameters to the particle swarm were $\phi = 0.8$, $\alpha = 0.1$, and $\beta = 0.2$, with all particles' positions initialized randomly within the bounds of $[0, 1]$ for each dimension. The values for ϕ, α, and β were chosen to encourage swarm exploration, as the inertia retained with each velocity update was rather high. The bounds for the initial distribution were chosen as between 0 and 1 since the heuristic weights are relative to each-other, and negative weights are mostly worthless. A range between 0 and 100 would've worked just as well. Figure 2 is a visualization of what a 10 particle swarm running for 20 iterations looks like under these same parameters. Lines drawn through the points represent the particles' movement over time.

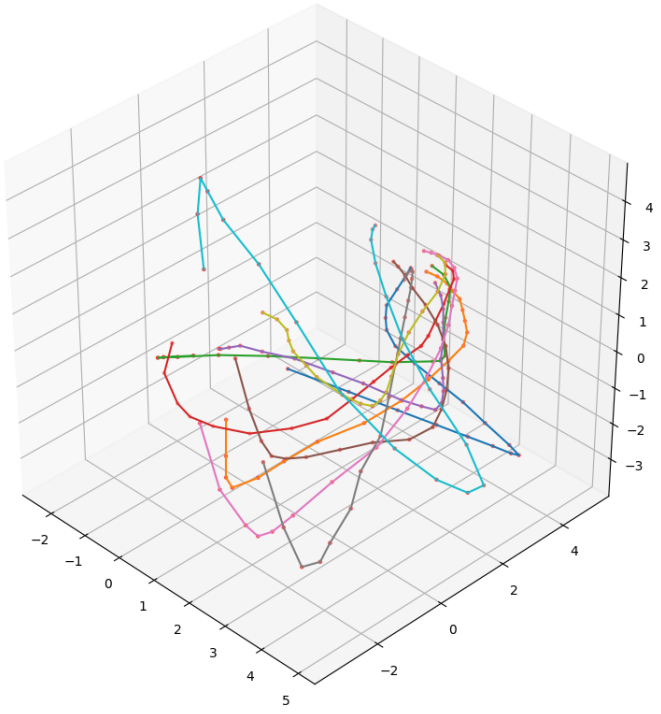

Figure 2: Visualization of 10 particles in a swarm optimizing heuristic weights over 20 iterations of Hidamari each in 3D

The fitness function used for the swarm played three games of Hidamari using the current particle position as heuristic weights for the state evaluation function, returning the least number of lines cleared of all three games played. This was done to avoid any heuristic weights that could not consistently pass 100,000 lines cleared. While there were a few distinct heuristic ratios that were able to achieve this goal, the final weights chosen for the AI were as follows:

Heuristic	Weight
Heuristic 1 (Aggregate height difference)	0.848058
Heuristic 2 (Aggregate height)	2.304684
Heuristic 3 (Number of holes)	1.405450

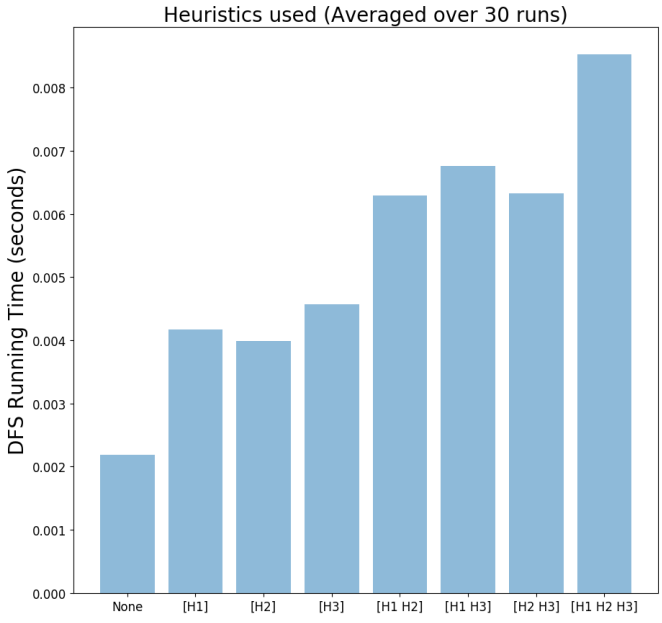

Figure 3: Running time of a single DFS traversal with different heuristics applied.

After deriving these heuristic weights from the particle swarm, the AI was put to the test. After running it for a few hours without any visualization, the AI was able to clear over 500,000 lines before manual termination.

The computation required for each heuristic, as seen in figure 3, was nearly equivalent. The DFS running time was increased by nearly constant amount for each additional heuristic run in the static evaluation function. Heuristic 3, number of holes, could potentially be removed for more performance at the cost of AI longevity since it contributes to the evaluation function the least. Removing it, however, is probably not worth constant computational performance gains since it will noticeably degrade the AI's ability to play the game.

Conclusions

For a performant Tetris AI that can be adjusted for varying difficulties, a simple depth-first search evaluating the first 1296 states proved to be sufficient. Currently the depth-first search uses 180kb, but this amount could easily be reduced as the depth-first search is not optimized to require memory based on only the depth of the tree, but instead has similar memory requirements to a breadth-first search.

The running-time of the algorithm with all three heuristics, as seen in figure 3, can be run twice every frame when the game is rendered at sixty frames per second on a dual-core laptop processor[1]. This is more than fast enough to provide a responsive experience on many devices, considering that the AI will usually have a few seconds to compute its plan while the current piece is falling, not just a sixtieth of a second.

Particle Swarm Optimization proved successful in finding good heuristic weights while reducing the search-space overtime allowing for very refined heuristic weights to be discovered for the AI in a reasonable time. The asynchronous modification of the particle swarm optimization algorithm was very beneficial, as it allows expensive fitness functions used by the algorithm to be run concurrently, which was critical since the fitness function used in this case was essentially running three games of Tetris before returning.

Future Work

There are numerous improvements that could be made to reduce memory usage and take advantage of parallel processing. The depth-first search used for the AI is just a modified breadth-first search using a stack rather than a queue. This could be modified to instead take advantage of how a depth-first search can use memory proportional to its depth rather than breadth, reducing the current 180kb required by the AI to some much smaller amount.

AI planning could also be optimized to run on a separate thread, in case it takes longer to create a plan than it does to render a frame. This would allow for the AI to take more time to perform more complex planning such as performing a depth 3 search in the state-tree, or computing more computationally expensive static evaluation heuristics. Currently parallelization isn't quite necessary, as the computation finished in a timely manner, but any future work on the AI may require it.

Since the AI only requires a representation of the current visible Tetris playfield, current falling piece, and the preview piece, it could potentially be adapted to work with any Tetris implementation rather than just Hidamari. The AI doesn't care about gravity, wall-kicks, scoring, or other elements that are often slightly different between Tetris implementations. As long as the core properties of line-clearing, piece rotations, and piece locking are present, the AI can be adapted to any implementation with those similar properties.

[1] Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz

References

[1] Helwig, Sabine, et al. Particle Swarm Optimization with Velocity Adaptation. Department of Computer Science, University of Erlangen-Nuremberg, Germany. 2009.

[2] Tetris Wiki Authors, et al. "The Tetris Guideline". http://tetris.wikia.com/wiki/Tetris_Guideline 2018.

[3] Lee, Yiyuan. "Tetris AI - The (Near) Perfect Bot". https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player 2018.