Parallel Multi-player Tetris solver

Yen Li Laih, Weiwei Lin
Carnegie Mellon University, PA
{ylaih, weiwei2}@andrew.cmu.edu

I. Summary

We implemented a parallelized Tetris solver for a tetris game using various algorithms and performed a detailed analysis of the speedup and performance of the different algorithms. We parallelized our dfs solver using OpenMP on the GHC and PSC machines, and CilkPlus on the GHC machines. Our final result reached near perfect speedup and can clear 1000 lines per second using 128 threads on the PSC machine.

II. Background

Tetris (1) is a puzzle game that involves randomly generated pieces of various shapes that descend onto the board. The player has to complete horizontal lines with the pieces so that the lines would disappear. If the stacked pieces reach the top of the board, the player loses. There are many search algorithms such as the depth-first search, breadth-first search, minimax algorithm 6, and genetic algorithms 5 that can be applied to solve this game. These search algorithms would benefit from parallel processing and speedup the search for the optimal solution of a single step.

The workload of Tetris is mostly computation-intensive since we would need to search through a huge search space to find the optimal solution under a time constraint. Specifically, we chose a game setting with 7 different shapes of tetrominoes, 4 different orientations, and a game board of 22 rows x 10 columns. Therefore, there are 7 x 4 x 10 = 280 possibilities for a single step in the game. Our search algorithm takes the next two to four blocks into account and calculates the best optimal step. Thus, for a single step (or a level) of the search space, there are actually 1 x 4 x 10 = 40 possibilities since the next tetromino block is known. For four levels, there would be 40 x 40 x 40 x 40 possible states. Since searching through the 40 possible states on the same level are not data-dependent on each other, it would benefit greatly from parallelization. However, searching and computing the scores of block placement on subsequent levels would depend on the game state of previous levels. Therefore, we can only perform parallelization across states on the same level. That still amounts to 40 states for the first level, 40 x 40 on the second level, and so on. Moreover, we found that there are some duplicate states when simply conducting naive parallelization across states on the same level. One such example is that the four orientations of a square tetromino is calculated as four different states when they are essentially the same. Pruning these states would cause workload imbalance, so we came up with various optimizations to alleviate the problem which we would introduce in the following sections.

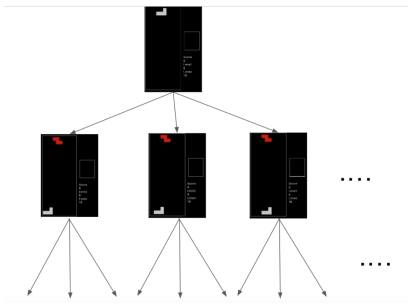


Fig. 1. Tetris game search tree.

To evaluate a state for the search algorithm, we defined some evaluation metrics related to the tetris game such as the maximum height of the stacked blocks, holes in the game board not filled by blocks and lines cleared. The evaluation score is a weighted sum of these metrics, and is used to help the algorithm select the optimal position and orientation of the next move. We conducted experiments to obtain a set of weights that performed the best on achieving the highest score, which is used in guiding our search algorithms. Since this is not related to the parallelization of the search algorithms, we omit the details in our report.

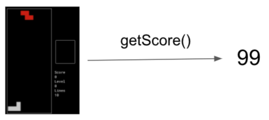


Fig. 2. Score of tetris game state.

Our implementation is based on a tetris game repository (2) implemented in C on Github. We adopted its code for the game-play and game settings. Given the next four blocks to be dropped, we implemented solvers that calculate the next best orientation and position to place a tetris block, and supply it to the game.

III. Approach

In this section, we describe the implementation and parallelization of the tetris game solvers. First, we implemented sequential versions of depth-first search and breadth-first search solvers from scratch in C to be incorporated into the tetris game code 2. Then, we parallelized both methods with OpenMP on the GHC machines and later conducted our experiments on the PSC bridges2 machines. We also built CilkPlus (3) on the GHC machines and parallelized our DFS search algorithm with CilkPlus to alleviate workload imbalance.

The sequential version of our DFS and BFS solvers take the current game state as the input, which includes the game board as a char array and the next four tetris blocks to be dropped, and calculate 4 steps ahead for the best orientation and position to place the next tetris block. The best orientation and position is returned to the game code, where necessary game functions are executed to place the falling block to the best position and advance the game.

Specifically, we make a copy of the current tetris game object, which includes the current and next four falling blocks and the current game board as a 22x10 char array. The copy is passed to the solvers. Then, we perform game functions such as moving and dropping blocks on the tetris game object copy, which updates the game board. We calculate the resulting scores and other performance metrics and return the best orientation and position for the block back to the game.

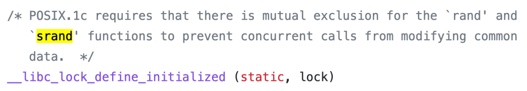
For BFS, we need to store all states on a level of the search space in a queue, to be applied to the game board when we go deeper into the next level of the search space. This takes up a lot of memory space when there are four levels of the search tree, which makes it difficult to scale. In comparison, DFS is much better suited for this task in that it does not have to remember all states on a level. We now describe the parallelization approaches we took to parallelize both search algorithms.

A. *Approach 1*

Our first approach involves parallelizing both our DFS and BFS solvers with OpenMP on the GHC and PSC machines.

For the BFS approach, we created separate queues for the different threads and distributed the first layer nodes to different queues to run BFS. However, this approach suffers from memory problems because we need to store all possible states for BFS, as mentioned above.

For the DFS version, we parallelized the first layer nodes of the DFS search space. Initially, this produced sub-linear speedup with regard to the number of threads. However, we found out that the bottleneck was the C random function, because it requires mutual exclusion according to the source code, as shown in Fig 3.



```
/* POSIX.1c requires that there is mutual exclusion for the `rand' and
   `srand' functions to prevent concurrent calls from modifying common
   data.  */
__libc_lock_define_initialized (static, lock)
```

Fig. 3. Random function bottleneck

As a result, we removed it when running the solver. This produced linear speedup up to 40 threads (on the psc machine), which is the number of first layer nodes. The graphs for search time and speedup with the increase of threads is shown in Fig **??** and Fig **??**, respectively. A problem with this approach is that it is difficult to parallelize with more threads using the current method because the number of first layer nodes is limited to 40. This is a major shortcoming since we should be able to perform 128-thread parallelization and much better speedup on the psc machines. To solve this issue, we decided on two alternative ways to perform parallelization: 1) rewrite the DFS parallel solver to map the workload onto the available threads (128 on the psc machines) and 2) perform the parallelization with Cilk instead of OpenMP. This brings us to our next approaches.

B. *Approach 2*

In this approach, instead of using DFS, we first calculate the number of possible leaf nodes, in the most naive case, it would be 40^4 nodes. We then index this 40^4 nodes and assign partition of the index to the worker threads where each worker thread is responsible for their assigned partition. For example, if there are 2560000 (40^4) nodes, each thread would be assigned 20000 nodes, so threads 0 would be responsible for node 0-19999, thread 1 responsible for node 20000-39999, and so on. Now the next problem is how do we map the node to the orientation and column position that should be dropped? We solve this by coming up with a one by one mapping function from the orientations and drop column position to the node id. We first consider there are four next blocks T_1, T_2, T_3 and T_4. Since we might prune redundant position, a tetris block might have less than 40 possible position. We let O_i and C_i be the number of possible orientation and drop column of tetromino block T_i, then the number of possible position of T_i would be $O_i \cdot C_i$ which we annotate as P_i. Now the number of possible placement of all fours blocks would be $P_1 \cdot P_2 \cdot P_3 \cdot P_4$. The next thing we do is we map the orientation o_i and drop column c_i of T_i to a integer tid_i where we define it as the tetris block id. We can achieve this easily by defining

$$tid_i = o_i \cdot C_i + c_i$$

where both o_i, c_i are integer and $o_i \in [0, O_i - 1]$, $c_i \in [0, C_i - 1]$. Now consider the DFS case, every leaf node actually represent a combination of tid_i, hence we can map a leaf node to a set $\{tid_1, tid_2, tid_3, tid_4\}$. Finally, we can map this set to the node id we defined previously by the formula

$$id_{node} = tid_1 \cdot P_2 \cdot P_3 \cdot P_4 + tid_2 \cdot P_3 \cdot P_4 + tid_3 \cdot P_4 + tid_4$$

Using the formula, workers threads can easily map their given node id to the corresponding tetromino block placements using division and modulus. The synchronization overhead in this approach is neglectable since all threads first store the optimal solution of their partition to a local variable. The local optimal solution is then compared in a critical section to get the global optimal, which is very fast since it needs only the number of threads amount of comparison. This approach is much more flexible than approach one since we are now partitioning the node ids which is trivial to do, and allows us to do various optimization on the search space without changing our algorithm. It is also more efficient implementation-wise since we only need to recover the board state every 4 steps, unlike in the DFS case, where we need to do this for every new node. In the final version, we applied various optimization to push the perforamce to the limit, we would discuss them in the following section.

C. Approach 3

The other alternative for increasing the number of parallelization threads is to use Cilk. Cilk is the ideal parallelization tool given the recursive nature of our DFS solver. One advantage is the simplicity to convert C code to utilize Cilk, and the second advantage has to do with the workload imbalance due to pruning the redundant states mentioned in the background section.

First, we built CilkPlus on the GHC machine by building our own llvm with cmake via miniconda. The modifications we made to convert the code to Cilk includes using $cilk\_for$ to parallelize across different states of a level of search space (different positions and orientations). For each level, the DFS algorithm has to compare the scores obtained across different threads to find the optimal position and orientation. At first, we collected the scores from all threads using an array, and synchronized the threads with the $cilk\_sync$ barrier before performing a serialized comparison for all scores to find the best one. Then, we found that the cilk reducer 7, which is available only in C++, could be used to perform the same function without using synchronization and serialized comparison. Switching to the Cilk reducer speeds up the search time and reduces the amount of memory used since we do not need to keep track of scores form all threads.

Since there are redundant states as mentioned above, pruning them for optimization would cause workload imbalance. This would not be a problem with Cilk, since it has a work-stealing scheduler that would automatically balance the workload. A small problem we encountered is that we could not manually install CilkPlus on the PSC machines as we did on the GHC machines, because the gcc versions on the PSC machines are too recent. Therefore, we did not conduct our experiments of this approach on the PSC machines.

IV. Challenges & Optimization

In this section, we bring up some challenges and performance bottleneck we encounter that limited our speedup and how we optimized these bottlenecks. There are in total three optimization we performed.

A. Redundant search space

The first bottleneck that limits our speedup and performance is that there were a lot of redundant search state we are searching over. Our naive approach iterated over 4 orientation of each block, however, some tetromino blocks has less than 4 orientations. For example consider the I space tetromino block in Fig. 4. It only has two unique orientations 0 and 1, the rest are just duplicate orientations. Secondly, we also assume that each block has 10 possible columm placement, which is obvious also not true since the shape of the block would limit its placement. For example, a I space block with orientation 0, would only have 6 possible column placement, otherwise the piece would go out of bound (under our game setting with column width as 10). Therefore, we were able to prune a lot of search space out, and each

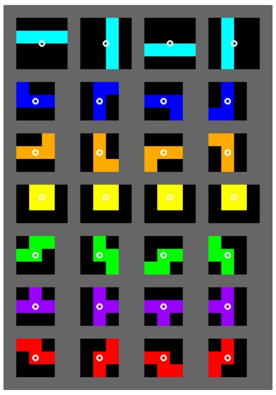


Fig. 4. Orientation of Tetromino Blocks. This figure shows orientation 0 to orientation 3 of all blocks from left to right

tetromino block has their own branching factor shown in Fig 5.

shape	I	J	L	O	S	T	Z
orientations	2	4	4	1	2	4	2
columns	10	10	10	10	10	10	10
branching factor	20	40	40	10	20	40	20

Fig. 5. Branching Factor of Tetromino Blocks. Image reference: 4

This optimization, however, can only be used in approach 2, otherwise it would introduce severe workload imbalance in the DFS tree, resulting in significant decrease in the speedup. In approach 2, since we are partitioning the combination space, it is easy for us to implement this but only on the orientations. We still iterate over 10 column for every blocks for coding simplicity.

B. Workload imbalance

Here we focus on the workload imbalance of Approach 2. As mentioned previously, approach 2 can evenly partition the search space among worker threads. However, we still observed workload imbalance in the scoring phase of each worker where we take the tetris board state and output a score that indicates the wellness of the board. The imbalance is subtle but became a bottleneck when we optimized our code to a certain extent. We discovered that it was due to the function that calculates the height of each board. Since the way we find the height is checking from top to bottom of whether each row is empty and return if we found one non-empty row, this would result in a imbalance when the height of the board is different. Considering how we partitioned our search space, each threads would be responsible for a continuous range of node id thus the final board state of a thread would be similar. If one threads tends to always have a higher board state than another thread, its workload on the scoring phase would be higher resulting in a imbalance (see Fig 6.) We solve this issue by simply storing the board height in the game object and maintaining it every time we place a tetromino on the board. This way we can get the height in constant time for any board state.

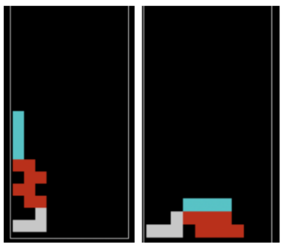


Fig. 6. Low and High Tetris Board of Same Block Combination

C. State caching

Another optimization we did is that we cached the board state to avoid duplicate drops. Consider one worker thread reponsible for node 0 to node 20000. Using the formula we derived earlier we can see that consecutive nodes would have the same prefix for their tid set. For example, if node 0 corresponds to $\{tid_1, tid_2, tid_3, tid_4\}$ then node 1 would correspond to $\{tid_1, tid_2, tid_3, tid_4 + 1\}$. In this case, we could cache the drop result of $\{tid_1, tid_2, tid_3\}$, and reuse it. In the best case, we can save up to 40 x 3 drops, where 40 is the branching factor of the fourth block and 3 corresponds to the previous 3 drops. Currently, we are only caching depth - 1 layer of drops, more aggressive caching is possible but are harder to implement, the current implementation already can significantly lower our drop time.

V. Results

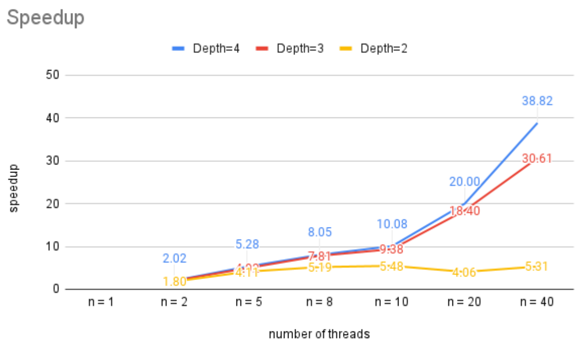


Fig. 7. Approach 1 Speedup vs. Number of threads for different search depths. This figure shows that the speedup increases linearly with the number of threads.

In this section, we present the results for the experiments we conducted on the GHC and PSC machines. We focused our experiments on approach2 since it has the best performance and also the most flexible design.

A. Approach 1 Naive DFS

Our first experiment measures the speedup with different number of threads for the approach 1 parallelized OpenMP DFS search algorithm for different search depths. A graph of the experiment is shown in Fig. 7. Here we only show

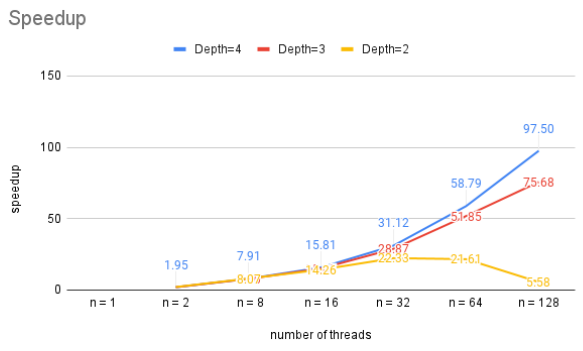


Fig. 8. Approach 2 Naive Speedup vs. Number of threads for different search depths.

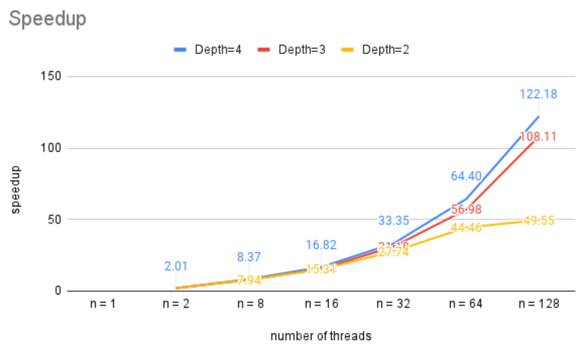


Fig. 9. Approach 2 Optimized Speedup vs. Number of threads for different search depths.

the speedup up to 40 threads since the DFS algorithm can't support threads more than 40. We can observe from the graph that as the search depth increases, the speedup approaches linear speedup to the number of threads used. We think the reason is that shallower depth results in smaller search space and the overhead of searching for the optimal position and orientation outweighs the speedup benefits. However, later we discover that the main reason for the sub-optimal speedup is because of a lock shared between the threads which we would discuss in later sections. Since this approach limited the number of thread we can use, we didn't further optimize this approach.

B. Approach 2 Before Optmization

The next experiment measures the speedup with different number of threads for the Approach 2 parallelized method for different search depths. The graph of this experiment is shown in Fig. 8. This experiment is conducted on n=1 to n=128 threads. We can observe the same trend of speedup approaching linear to the number of threads used as the search depth increases. However, the speedup is not perfect for n = 64 and n = 128. This is due to the workload imbalance we mentioned in the previous section. As for depth = 2, we observed the same performance downgrade in the Approach 1 for high number of threads.

To further investigate the non-linear speedup for shallower search depths, we conduct our third experiment, which measures the duration of each function. The results of this

Depth 4	n = 1		n = 128		ratio
	time(ms)	pct(%)	time(ms)	pct.(%)	
solve	9534.483	74.01%	77.678	71.29%	1.25
drop	2451.974	19.03%	22.209	20.38%	1.61
copy	70.101	0.54%	0.591	0.59%	1.24
create	0	0.00%	0.388	0.36%	5687.5
other	826.24	6.41%	8.1	7.42%	–
total	12882.80	100.00%	108.953	100.00%	1.27
Depth 3	n = 1		n = 128		ratio
	time(ms)	pct(%)	time(ms)	pct.(%)	
solve	232.794	77.54%	1.892	63.51%	1.2
drop	45.863	15.28%	0.469	15.74%	2
copy	1.752	0.58%	0.013	0.59%	6.36
create	0	0.00%	0.396	13.29%	1045.8
other	19.81	6.60%	0.2	7.02%	–
total	300.22	100.00%	2.979	100.00%	1.68
Depth 2	n = 1		n = 128		ratio
	time(ms)	pct(%)	time(ms)	pct.(%)	
solve	5.803	81.72%	0.051	13.32%	94.5
drop	0.783	11.03%	0.007	1.83%	17.8
copy	0.043	0.61%	0.00034	0.59%	2.86
create	0	0.00%	0.316	82.51%	395.4
other	0.47	6.65%	0.0	2.26%	–
total	7.10	100.00%	0.383	100.00%	116.5

TABLE I

FUNCTION DURATION BREAKDOWN OF NAIVE APPROACH 2.

Depth 4	n = 1		n = 128		ratio
	time(ms)	pct(%)	time(ms)	pct.(%)	
solve	8002.533	79.05%	62.937	74.85%	1.1
drop	1218.769	12.04%	12.376	14.72%	1.25
copy	70.227	0.69%	0.592	0.70%	1.48
create	0.001	0.00%	0.0002	0.00%	1.47
other	832.39	8.22%	8.2	9.73%	–
total	10123.93	100.00%	84.083	100.00%	1.09
Depth 3	n = 1		n = 128		ratio
	time(ms)	pct(%)	time(ms)	pct.(%)	
solve	183.701	77.63%	1.543	71.20%	1.09
drop	31.128	13.15%	0.338	15.60%	1.4
copy	1.766	0.75%	0.014	0.65%	5.14
create	0.0001	0.00%	0.00009	0.00%	1.154
other	20.03	8.47%	0.3	12.55%	–
total	236.63	100.00%	2.167	100.00%	1.11
Depth 2	n = 1		n = 128		ratio
	time(ms)	pct(%)	time(ms)	pct.(%)	
solve	4.578	77.88%	0.039	32.50%	75.96
drop	0.778	13.24%	0.009	7.50%	38.58
copy	0.043	0.73%	0.0003	0.25%	2.86
create	0.00006	0.00%	0.0001	0.08%	1.09
other	0.48	8.15%	0.1	59.67%	–
total	5.88	100.00%	0.12	100.00%	1.12

TABLE II

FUNCTION DURATION BREAKDOWN OF OPTIMIZED APPROACH 2.

experiment is displayed in Table. I. For this experiment, we measure the execution time of four main functions that is necessary during our DFS search for different search depths using 1 and 128 threads, respectively. We used C function gettimeofday which has the granularity of microseconds. The overhead of the time measurement is under 1us so wouldn't affect the solver performance. In the table, we measure 4 crucial function including solve, drop, copy and create. The solve function denotes the tg_get_score() function, which is used to calculate the score of a game state. The drop function denotes the tg_drop() function, which drops the falling block into place after the block position (column) has been determined. The copy function denotes the tg_copy() function, which copies the tetris game solver, this is needed because we want to resume the board state. The create function denotes the tg_create() function, which creates a new tetris game solver function. We measured tg_create() because we discovered it was the bottleneck for shallower depth, especially for depth = 2. Additionally, we also measure the maximum and minimum execution time of each function between different threads and define ratio as maximum / minimum, the larger ratio is, the more imbalance the function workload is.

We can observe from Table. I for depth 3 and 4, most execution time is on solve and drop accounting for over 85% of the work time, which is what we expected. However, there is a slight workload imbalance of the two function (1.2 and 1.6 for depth = 4) that limits our speedup from reaching perfect speedup. The overhead of multiple threads here are still small with around 7% of total time. As for depth = 2, we can see that the execution time is mostly on tg_create() which is abnormal. The ratio of tg_create() in all depth is also unreasonably large. After some dig in, we find that it was because the same problem we encounter earlier, there was a srand() function call in tg_create() that acquires a lock betweens threads. Therefore, we removed this srand() in the optimized version of approach 2.

C. Approach 2 After Optimization

Here we present the result of our optimized approach 2, we applied the three optimization mentioned in optimization section and also fixed the srand() locking problem. The speedup result is shown in Fig. 9. We can see that depth = 4 reached near perfect speedup (122.18) after we optimized the workload imbalance of the tg_get_score() function. As the depth decrease, the overheads takes in effect and the speedup is lowered, as depth = 2 has only 49.55 speedup, however this is still much higher than the naive version (5.41).

From Table. II there is 4 things worth pointing out. First, the tg_create() is no longer a bottleneck for depth = 2 n = 128, instead other are now the main reason for the performance slowdown. This is expected since we fixed the lock problem. Second, as the depth decrease, the other percentage also increase, especially in depth = 2, where 60% of the time is on other. This is expected since other represents the overhead and the overhead increase as depth decrease since we have less calculation for each thread. Third, the drop function time for depth = 4 decreased from 20% to 15%, indicating our caching is effective. Finally, we can see that nearly all ratio decrease for all depth, which implies that our optimization has successfully solve the workload imbalance, resulting in better speedup.

VI. Conclusion

In this project, we present and analyzed several way to implement a tetris solver, including a approach (approach 2) that partitions the tetris search space evenly making it extremely easy to parallelize our solver. We then analyzed the performance bottleneck and reasons for imperfect speedup and proposed 3 optimizations. After applying our optimizations, we show that we removed nearly all workload imbalance and increased our speedup to near perfect speedup. To prove our result, we measured and analyzed the time spent on each critical functions using gettimeofday(). As a result, we were able to build a tetris solver that can clear up to 1000 lines per second.

REFERENCES

[1] https://en.wikipedia.org/wiki/Tetris

[2] https://github.com/brenns10/tetris

[3] https://cilkplus.github.io

[4] https://tetris.fandom.com/wiki/SRS?file=SRS-pieces.png

[5] Man, Kim-Fung, Kit-Sang Tang, and Sam Kwong. ”Genetic algorithms: concepts and applications [in engineering design].” IEEE transactions on Industrial Electronics 43.5 (1996): 519-534.

[6] Korf, Richard E., and David Maxwell Chickering. ”Best-first minimax search.” Artificial intelligence 84.1-2 (1996): 299-337.

[7] Frigo, Matteo, et al. ”Reducers and other Cilk++ hyperobjects.” Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. 2009.

[8] Demaine, Erik D., Susan Hohenberger, and David Liben-Nowell. ”Tetris is hard, even to approximate.” International Computing and Combinatorics Conference. Springer, Berlin, Heidelberg, 2003.

[9] Galván-López, Edgar, et al. ”Heuristic-based multi-agent monte carlo tree search.” IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications. IEEE, 2014.

[10] Singhal, Shubhendra Pal, and M. Sridevi. ”Comparative study of performance of parallel Alpha Beta Pruning for different architectures.” 2019 IEEE 9th International Conference on Advanced Computing (IACC). IEEE, 2019.

[11] Burgiel, Heidi. ”How to lose at Tetris.” The Mathematical Gazette 81.491 (1997): 194-200.

VII. LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT

We implemented nearly every function together.
Yen Li Laih (ylaih): 50% Wei Wei Lin (weiweil2): 50%