2048 Implementation FAQs

Q. How should I approach this project to get the best grade in the minimum amount of time?

Reaching 2048 consistently can easily take a lot of time to debug and improve, so write and test your program in small segments. **Most important is to correctly implement expectiminimax and alpha-beta pruning.** With correct alpha-beta pruning and a simple heuristic like the number of free cells, you already get to at least 256 in your top five runs, which is 38 of the 100 points! It might be helpful to start with standard minimax, since expectiminimax simply adds a layer of prediction complexity.

Make sure expectiminimax and alpha-beta pruning work correctly before moving onto heuristic tuning. We recommend saving your correct version of expectiminimax in case you need to revert to this version later for a clean start. With these two correct, you can tweak heuristics and reach 512 tiles for 63/100 points in the next couple hours.

This website (http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/) might be useful in understanding standard minimax search and alpha-beta pruning.

Q. How do we make sure our PlayerAI doesn't time out during a move?

In the `GameManager` class, we enforce the following time limit: if `getMove()` takes longer than 0.2 seconds to return a move, the game automatically terminates and assesses the maximum tile value at that point.

How can you ensure your player logic terminates before `GameManager` times out? **DO NOT rely on a depth-limit as the basis for ensuring you fit within this runtime constraint.** Some past submissions set a max depth such as `if depth => 4: return` to cut off search time. While this might work on your own computer, it's not a machine invariant way of accomplishing the task in the given time limit. Your assignments will be executed on one grading platform to ensure a fair grading environment, so if your program fails to meet the time limit on the grading machine your grade will suffer. **Submissions using a depth constraint typically halted before the first `PlayerAI` move**, since this first move has a near maximum branching factor with the run trying to get the `maximumTile` of 2 or 4.

There are many Python and shell tools to help you track program time. Make sure you **rely on CPU time instead of wall time** for your diagnoses. If you look at `GameManager_3.py`, you'll see we use `time.clock()` to enforce the time limit. If you want to be safe, use `time.clock()` for any time limits you implement in your `PlayerAI_3` logic.

Q. Are there more grading examples?

The following are examples of the top five maximum tile values and their corresponding grades. The maximum score is 100/100 points.

Top 5	Median	Min	Max	Score
[1024, 1024, **2048**, **2048**, 4096]	2048	1024	4096	100
[1024, 1028, **2048**, **2048**, **2048**]	2048	1024	2048	100
[1024, 1024, 1024, **2048**, 4096]	1024	1024	4096	100
[1024, 1024, 1024, **2048**, **2048**]	1024	1024	2048	98
[1024, 1024, 1024, 1024, **2048**]	1024	1024	2048	93
[1024, 1024, 1024, 1024, 1024]	1024	1024	1024	88
[512, 1024, 1024, 1024, **2048**]	1024	512	2048	88
[512, 1024, 1024, 1024, 1024]	1024	512	1024	83
[512, 512, 512, 512, 512]	512	512	512	63
[2, 2, 4, 4, 4] (e.g. depth limit)	4	2	4	0

* As the minimum of the top five runs receives one-level smaller/higher values, the credit will decrease/increase by 5 points.

Q. What is the highest score among past submissions?

While we don't track the highest scores, one submission recorded [2048, 2048, 2048, 4096, 4096].

Q. How can we come up with good heuristics?

Your heuristics are key to your PlayerAI's consistent success. Watch some pro 2048 players' videos. Observe patterns you might try enforcing via heuristics, and compare these players' performances to that of your PlayerAI as a potential benchmark of whether your AI is performing well or not.

Take into consideration both quantitative and qualitative measures, such as:

- the absolute value of tiles,
- the difference in value between adjacent tiles,
- the potential for merging of similar tiles,
- the ordering of tiles across rows, columns, and diagonals,

We also encourage you to research 2048-specific heuristics.

This repo (http://ovolve.github.io/2048-AI/) is a basic 2048 AI written in JavaScript. Your PlayerAI should do much better. This particular StackOverflow thread (https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048) provides the concepts of "Monotonicity" and "Smoothness" which you may choose to implement. **Whatever ideas you use, remember to use expectiminimax and alpha-beta pruning as instructed to receive full credit.**

Q. How can we combine multiple heuristics?

As mentioned in the instructions, a simple but strong approach is to use a weighted combination of your heuristics, and tune these weights. Suppose for example we use the two heuristics `average_tile_number` and `median_tile_number`. The two would combine into something like:

```
return w1 * average_tile_number + w2 * median_tile_number
```

You can now adjust these weights ("hyperparameter tuning") manually or automatically. We recommend you **start with manual tuning to understand the individual and combined effects of your heuristics**, then go to automated tuning if this interests you. Many systematic ways exist for tuning hyperparameters, which you may feel free to implement for quicker optimization until you reach a resulting set of weights you're happy with. For even speedier testing, consider starting your tuning from a set of intermediate board states, e.g. if you consistently get to 512, perhaps start your boards from this state during tuning. The grading script will overwrite all read-only files before grading, so any changes you make in these files will be removed.

Q. How can we combine multiple heuristics with different units?

Imagine combining the heuristics `available_cell_count` and `average_tile_value`. If we naively combine these two with their raw values, the output depends almost entirely on `average_tile_value` because this value will likely be much greater than the 16 cells of the board.

A quick fix is to apply `math.log2(average_tile_value)` before adding `available_cell_count`. Think about what scaling and normalization you need to get comparable heuristic values, **before applying weights.**

Q. How many heuristics should be combined?

2 - 5 heuristics are enough to reach 2048.

Q. Is skeleton code optimized?

No. In particular, `move()`, `getAvailableCells()`, `insertTile()`, and `clone()` are not the most efficient implementations of their tasks. However, optimizing these methods with low level operations and bit-array representation is not the purpose of this assignment and by no means necessary.

Students have written perfectly scoreing PlayerAIs using only the methods given. You're free to write your own helper methods and modules (keeping in mind that Grid_3.py is read-only), but focus your efforts on correct algorithm implementation and smart heuristics. After that, as an extra boost [or last resort] you can write your own optimized versions of these methods.