PEDAGOGICAL POSSIBILITIES FOR THE 2048 PUZZLE GAME

Todd W. Neller
Department of Computer Science
Gettysburg College
Gettysburg, PA 17325-1400
717-337-6643
tneller@gettysburg.edu

ABSTRACT

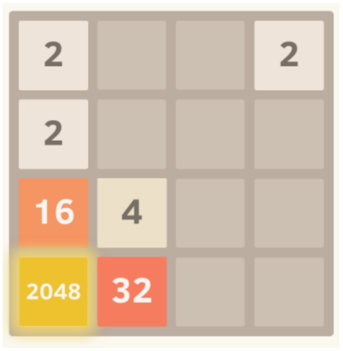
In this paper, we describe an engaging puzzle game called 2048 and outline a variety of exercises that can leverage the game's popularity to engage student interest, reinforce core CS concepts, and excite student curiosity towards undergraduate research. Exercises range in difficulty from CS1-level exercises suitable for exercising and assessing 1D and 2D array skills to empirical undergraduate research in Monte Carlo Tree Search methods and skilled heuristic evaluation design.

INTRODUCTION

Released in March 2014, the online and mobile puzzle game 2048[1] began its surge of popularity. In this paper, we will describe this engaging game and point out ways that it may be used to advantage in the Computer Science curriculum.

The outline of the paper is as follows: We begin by describing the 2048 game, noting work already in progress to leverage the popularity of this game in CS education, and describing the scope of our unique contribution. Next, we discuss the modeling of the game state itself from simple fragments of the specification suitable for CS1 to a full game state model suitable for CS2. We then turn our attention to the creative challenge of heuristic game state evaluation. Finally, we describe a progression of Monte Carlo simulation and search methods providing modest, achievable goals for introductory CS1 through advanced AI students.

Game Description



The 2048 game is played on a 4-by-4 square grid that is partially filled with tiles labeled with powers of 2. The primary goal of the game is to merge randomly generated tiles in order to create a tile labeled 2048 (2^{11}) as shown above. Beyond

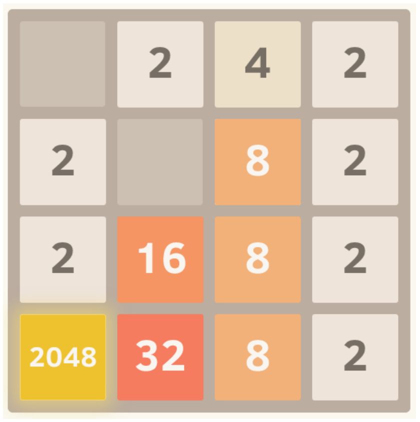
[1] http://gabrielecirulli.github.io/2048/

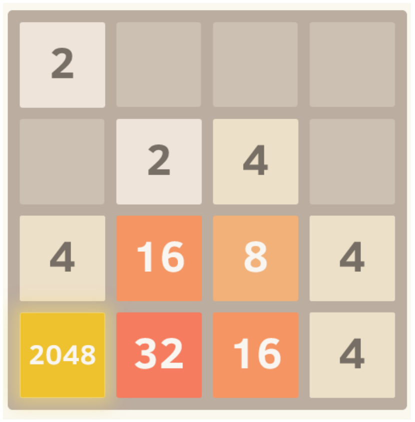
this primary goal, however, players may continue play so as to achieve higher scores and tiles with higher powers as well. The highest possible tile would be 131,072 (2^{17}), although this would be improbable to achieve.

Initially, the board has two randomly generated tiles. Randomly generated tiles are distributed uniformly over all empty cells, but such tile values are either 2 (with probability .9) or 4.

In each turn, a player may choose to move left, right, up, or down. A move in a given direction is legal if it results in a change to the grid. When no legal move exists, the game is over. Change occurs through tiles merging and/or sliding into different cells. Thus, the grid must contain at least one empty cell or at least one pair of identical, successive tiles along a row or column for the game to continue.

When the player chooses a direction, the tiles in all rows or columns aligned with that direction merge and slide in that direction if possible. Let us refer to a single row/column as a line, with the front of the line being the end farthest in the direction of motion. From the front to the back of the line, successive identical tile pairs merge into a new tile with their sum. (Merged tiles cannot merge again in the same movement.) Then, all tiles (merged or not) slide as far as possible in the direction of motion. Consider the following grids before (left) and after (right) a downward move:





In the rightmost column, both bottom and top pairs of vertically adjacent 2s merge into two 4s that slide downward. Note that these merged 4s do not then merge with each other in the same movement. In the next column to the left, note the 3 vertically adjacent 8s. The two bottommost 8s merge into a 16, being farthest in the direction of motion. In the next column to the left, there are no possible merges; the 2 simply slides down. In the leftmost column, two 2s merge into a 4.

The upper-left corner 2 is a randomly generated tile. After each legal move, a single tile is randomly generated as described above. Scoring is simple: Each time a merged tile is created, the value of that tile is added to the score. Thus, as the board is seeded with 90% 2s and 10% 4s after each move, one seeks to move so as to maximize one's expected future score.

Previous CS Teaching and Learning with 2048

Already, a free, short Udacity course "Make Your Own 2048"[2] is being offered to "teach you the basics of HTML and CSS and how they interact with Javascript". Having no prerequisites, the course invites the student to "make small changes to HTML and CSS files" that significantly change the look of the game.

Being free and open source, 2048 clones abound these days. Many have been motivated to dig into the 2048 source code via Github[3], learning enough Javascript to make not only trivial and cosmetic changes[4], but also mathematical variations[5] and Artificial Intelligence (AI) hint systems and demonstration bots[6].

Scope

Given the rapidly growing interest in do-it-yourself 2048 clones and ample coverage of such cloning skills elsewhere, we instead turn our attention in this paper to the space of 2048 assignment ideas ranging in application from CS1 to Introductory AI courses. We do not presume Javascript as a programming language. Indeed, we put aside both language and graphical user interface commitments, hoping to explicate the rich opportunities the game provides for teaching core CS concepts in a fun and entertaining way. Our simple, text-based Java experimentation with these ideas has proven as interesting as the game itself.

MODELING

One of the first problems we can present students is this: Given a 4-by-4 puzzle grid, compute whether or not the game is over. Since the game is over if and only if there are no empty cells and no orthogonally adjacent identical tiles, this can be approach via two steps: (1) a search of a 2D array for the value 0 (encoding an empty cell), and (2) a search of each row and column for adjacent pairs. This exercise would be suitable in a unit on multidimensional arrays in a CS1 course, or as a relatively easy programming contest problem. With this and other problems that follow, many variations are possible, as there are many precursors and subsequent variations of this game, e.g. Threes[7] and 2584 (the Fibonacci variant), respectively.

One can also parameterize this problem with a direction: Can the player legally move in a given direction? Note that, while reducing our adjacency consideration to one dimension, one aspect of the computation requires care: Not every empty cell aids movement in a specific direction. If there are empty cells(s) in a line that is already compacted in the given direction, such cells do not make movement in that direction legal.

Using this parameterization, one can then compute the legal move directions for the grid.

[2] https://www.udacity.com/course/ud248
[3] https://github.com/gabrielecirulli/2048
[4] http://doge2048.com/
[5] http://mathmunch.org/2014/03/24/2048-2584-and-variations-on-a-theme/
[6] http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048
[7] http://asherv.com/threes/

Another simple problem concerns the modeling of a single line movement in 2048. Let the first index of an array be the farthest in the direction of motion, and let the value 0 again represent an empty cell. Have students first devise an algorithm that will slide and compact all nonzero values to the beginning of the array. Then students can augment this algorithm with tile-merges. Finally, the tile-merging code can be modified to calculate accumulated score gain.

Alternatively, one may approach the processing of a line by enqueuing nonzero values into a queue and dequeuing values into a new/cleared array. In this approach, a merge takes place when a just-dequeued value matches the new value at the front of the queue, in which case the second value is dequeued as well and the sum is placed into the array. This would work nicely as a simple CS2 queue exercise.

From such underlying computation, one can then compute an entire grid movement and associated score gain one line (i.e. row/column) at a time.

When working with an object-oriented (O-O) language, it is useful to consider the design of a 2048 state object with a well-designed interface for user interaction and play modeling (e.g. isGameOver(), makeMove(Direction), generateRandomTile(), getScore(), etc.), heuristic evaluation (e.g. getGrid()), and search/simulation (e.g. clone() or copy constructor). Of course, one's best design would depend on one's possible follow-on projects described in the following sections.

HEURISTIC EVALUATION

One of the more intriguing questions the player faces is how one prefers one game state to another. With experience, a player notes that it is often advantageous to prefer a monotonic snake-like pattern where the largest tile is in a corner and tiles monotonically decrease back-and-forth along rows or back-and-forth along columns. A particularly delightful game experience is then performing successive merges along this snake-like path to form a new maximum tile in the corner.

However, one's ideal configuration cannot generally be maintained. The question then again is this: "What is to be preferred in the space of all configurations?" In AI, this scoring of states is referred to as *heuristic evaluation*. By defining a heuristic scoring function (or heuristic function), we formalize what we value in a given configuration of tiles. Consider how much one should value…

- … having our maximum value tile in a corner?
- … monotonicity along a snake-like path?
- … monotonicity along any line?
- … the maximum tile of a line along an edge?
- … adjacent identical tiles?
- … empty cells?

For example, one heuristic evaluation advocated in an online discussion of optimal 2048 play[8] is to equally reward (1) each line where the maximum value is at one end, and (2) each empty cell of each line. This strategy, when coupled with optimized expectimax search of up to depth 8, was reported to achieve a 16384 tile

[8] http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048; https://github.com/nneonneo/2048-ai

in 13 out of 100 trials. Thus, heuristic evaluation need not be complex to be effective.

Arguing the merits of various strategies may have some fascination, but here we are provided a challenge to *define* our strategies precisely and *test* them rigorously with play that is guided by such heuristic evaluation. The 2048 game provides the Computer Scientist a wonderful opportunity for both problem-solving creativity and the experience of empirical research.

From an O-O perspective, one may implement the heuristic evaluation as a tightly integrated function of the state object. This would have the advantage of inviting students to extend class definitions, overriding the parent heuristic evaluation, yet inheriting and possibly using various supporting computations.

One might also take the approach of creating a separate evaluator object that is allowed to visit and evaluate the state object. Thus, heuristic evaluation invites interesting discussion of O-O design patterns.

MONTE CARLO AND SEARCH TECHNIQUES

Finally, we consider ways we can use both elements above, a game model and a game state heuristic evaluation, to inform play.[9] We begin simply with algorithms suitable for CS1 exercises and then elaborate to projects suitable for an introductory or advanced AI course. In all of these techniques, we assume either (1) an ability to clone a state object so as to preserve a parent state, or (2) an ability to both make and undo alternating moves and random tile generations. In short, we require an ability to simulate the effects of different play decisions.

Throughout all of these examples, we will make use of the simple heuristic described in the previous section that, for each row and each column line, scores a point for a maximum value at a line end plus the number of empty cells in that line.

Simple Greedy Play

The simplest use of a heuristic evaluation function is to make each legal move without subsequent random tile generation, evaluate each resulting state, and choose the move leading to the state that maximizes our heuristic function. Let us call this "simple greedy" play.

Simple greedy play is weaker than many human players because it is so short-sighted, not even considering the ramifications of the next random tile generation. For introductory CS students, it would be desirable to have a pre-recursion technique that does not require consideration of game tree traversal and evaluation. For this, we have developed and tested a simple approach based on Monte Carlo simulation.

Simple Monte Carlo Play

One way to gain weak insight beyond the next move is to make a few random moves to get a very rough feel for the heuristic values further along that line of play.

[9] An online demonstration of AI added to provide a player hints or play an automated game can be found at http://ov3y.github.io/2048-AI/

In general, *Monte Carlo simulation* is a fast way to gain probabilistic insight in systems where events of interest are not too infrequent [1].

Suppose we want to get a rough evaluation for states beyond a move in a given legal move direction. We copy the current state, make the move, and then make a few more random moves. For this example, we'll make one deliberate move and two random moves. After all three moves and subsequent random tile generations (which we'll assume henceforth), we evaluate the resulting state with our heuristic function.

We repeat this process many times and average the evaluation. For our experiments, we'll perform 1000 Monte Carlo simulations for each legal move. Once we've averaged the evaluations for each legal move, we choose the move that maximizes our expected state 2 random moves beyond our chosen move at depth 3.

Note that this technique relies on iteration, and does not require the student to have mastered recursion or game-tree evaluation concepts.

Expectimax

In the context of an introductory AI course, we would naturally show students that the structure of the game tree consists of choice nodes that one maximizes alternating with chance nodes that one averages. The basic relevant algorithm would be (depth-limited) expectimax[10]. Given the branching factor of the problem, a depth-limited expectimax without optimizations in modeling (e.g. fast bit representation and operations) and search (e.g. caching), one would need to terminate at a shallow depth-limit in order to achieve rapid play.

Expectimax with Chance Sampling

We can gain much of the benefit of expectimax reasoning with faster performance by performing chance sampling of random tile generation. In doing so, we can limit our effective branching factor and search faster or deeper.

Sampling such chance events is an interesting exercise. An incorrect naïve approach would uniformly sample all possible events, treating the generation of 2 and 4 tiles as equiprobable. If one randomly generates samples so as to allow duplicates, it slows the process of gaining unique samples in proper proportion. A correct efficient approach would be to create two copies of a list of empty cells. One copy represents unchosen 2 tile generations and the other copy represents unchosen 4 tiles generations. The relative probability of sampling an unchosen 2 tile is .9 times the number of unchosen samples divided by the sum of same product plus .1 times the number of unchosen 4 tile samples. Using this insight, we choose the list/tile to sample with correct probability, and then select a random unchosen cell from the list in a manner similar to the Fisher-Yates shuffle algorithm, building an unbiased sample efficiently.

[10] Expectimax is the expectiminimax algorithm [5] without the minimizing adversarial player.

Performance Results

Using each of the previous algorithms, 1000 games were played. Performance results are summarized in the following table:

Algorithm	Median Score	Mean Score	Std. Dev.	Max. Tile	Median Tile
random	1028	1075	512	256	128
simple greedy	3326	3620	1708	1024	256
simple Monte Carlo (depth 3)	14586	14295	6592	2048	1024
expectimax (depth 2)	14398	14241	6717	2048	1024
expectimax (depth 3)	27132	25270	10429	4096	2048
10-sample expectimax (depth 2)	14496	14623	7096	4096	1024
10-sample expectimax (depth 3)	27052	24883	10106	4096	2048

Of greatest significance to introductory students is the fact that the Simple Monte Carlo Play to depth 3 described above requires no recursion or game-tree evaluation, yet performs at a similar level to depth 2 expectimax.

To have median performance that achieves the winning 2048 tile, depth 3 expectimax is necessary, but this can be achieved more efficiently with sampling. Switching to a 10-sample expectimax for depth 2 and depth 3 took 55% and 36% of the computational time, respectively, without significant degradation of performance.

The maximum score achieved in any of our initial experimental runs was 69624 using a 10-sample expectimax to depth 3. 100 runs of non-sampling expectimax to a maximum depth of 10 reportedly achieved the 2048, 4096, 8192, and 16384 tiles in 100%, 97%, 76%, and 13% of runs, respectively. The median score was 157652, and the maximum score was 377792.[11]

Finally, we note that 1000 games is a small sample size for these initial exploratory experiments. The purpose here is to provide a rough feel for the relative merits of these algorithms in the order of their programming complexity. We thus provide reasonable approximate expectations for performance with a simple heuristic and a wide range of levels of algorithmic difficulty for the student.

We hope the reader is encouraged by the performance that is achievable with relatively simple methods and is thus emboldened to have students explore this fascinating design space. Indeed, this general topic area suggests excellent undergraduate research projects. For example, an introductory student might experiment with different heuristic evaluations using simple Monte Carlo play, whereas an advanced student might do the same using expectimax play.

As many online are sharing their own 2048 AI algorithmic approaches, this appears to be an excellent, accessible playground for formative experiences in empirical AI research.

[11] http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

Next Steps

An advanced undergraduate- or master's-level research project might consist of reading a survey of Monte Carlo Tree Search (MCTS) algorithms [2] and creating a MCTS player for 2048. We have not yet had opportunity to apply MCTS to 2048, but searching online, we observe that others are thinking and working along the same lines.

CONCLUSIONS

Computer Science education, as with many areas of education, has benefitted from an increasingly contextualized approach[12] to teaching its abstract concepts. Board games [3] and video games [4] have long provided good contexts of engaging experiences for teaching many CS concepts.

In this article, we have described a range of feasible exercises and projects for the new, popular 2048 puzzle game. From the CS1 student testing game-ending conditions to the advanced AI student applying expectimax with chance sampling, we hope the reader appreciates the many interesting pedagogical possibilities offered by 2048.

We have experimented with these ideas, yet given that 2048 burst into popular consciousness one month ago, there has not been time to assess such assignments. However, we have considerable experience with game exercises ranging from CS1 through advanced AI, and serve here to primarily report on the engaging quality of this game, and the many pedagogical possibilities that strike us as worthwhile. Just as the 2048 game is engaging, we expect that relevant exercises will also engage CS students and fruitfully exercise core CS concepts. We foresee this as a ripe area for empirical research as evidenced by many 2048 blog articles and open-source project spin-offs. As CS educators, may we catch these teaching opportunities early and aid each other in rapidly developing curricular applications.

REFERENCES

[1] P. J. Nahin, Digital Dice: Computational Solutions to Practical Probability Problems, Princeton, New Jersey: Princeton University Press, 2013.

[2] Cameron Browne, et al, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 4, no. 1, pp. 1-43, March 2012.

[3] Peter Drake, Kelvin Sung, "Teaching Introductory Programming with Popular Board Games," in *SIGCSE '11 Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, Dallas, Texas, USA, 2011.

[4] K. Sung, "Computer games and traditional CS courses," *Communications of the ACM,* vol. 52, no. 12, pp. 74-78, December 2009.

[5] Stuart Russell, Peter Norvig, Artificial Intelligence: A Modern Approach, 3rd ed., Upper Saddle River, New Jersey: Prentice Hall, 2010, p. 178.

[12] Mark Guzdial's talk on Contextualized Computing Ed.: http://home.cc.gatech.edu/guzdial/169 .