2048 Project Report

Introduction

In this report, I will detail my process of creating a program designed to play 2048 automatically using depth search. 2048 is a puzzle game where tiles appear randomly on a four by four grid. A player can slide all of the tiles on the grid at once in a cardinal direction of their choosing. Tiles of identical value can merge this way, freeing a space and adding points to a cumulative tally. The game continues until moves are no longer possible, and though many consider obtaining a 2048 tile to be the objective of the game, the game will still continue afterward. This means the effective victory condition is to achieve as high of a score as possible. The machine learning model that I have created consistently achieves point totals in the tens of thousands.

Related Work

Few machine learning researchers have taken an approach like this one, but there are multiple examples of them using more sophisticated models to play 2048 in a way that is highly accurate to the rules, using techniques such as reinforcement learning to play as well as possible without any user input. Two examples of this are detailed in this dissertation by Hung Guei: https://arxiv.org/abs/2212.11087, and this paper by Shilun Li and Veronica Peng: https://arxiv.org/abs/2110.10374. Though they helped with inspiration, these works were not referenced for this project.

Proposed Method

My implementation of depth search would see the 2048 grid as a tree of potential outcomes. Each possible move available to the player is simulated in a copy of the grid. From those potential outcomes, further moves are simulated, increasing the total computation by a power of four, relative to the user specified depth of the tree. The ideal outcome is selected by comparing the amount of points that each branch has acquired and picking the one which obtained the most.

Acquiring data in advance was not required to make this project possible beyond an understanding of the computational logic present in the game 2048. Before a turn begins, one tile is created to exist within a random legal location within the four by four game grid. A legal location would be any spot that contains no tile already. The created tile will have a value of two 90% of the time or a value of 4 10% of the time. This is not specified in any game rules that I could find, but found based on data I collected by playing a web browser version of 2048 and counting the quantity of each tile created.

Once the tile has spawned, the player must make a legal move. A legal move would be north, south, east or west. This move initiates the process of iterating between each of the tiles in the grid to see whether their position would be altered by the selected cardinal direction. For example, let us say north was selected and the bottom row consists of tiles with the quantity two, and the row above it also consists of similarly valued tiles. The program would iterate through each column, comparing values row by row within those columns. In each of these columns, values will slide, merge then slide once more.

The act of sliding the tiles in this case is achieved by selecting each tile in the column, checking whether it has a value of zero, indicating an empty space, then checking further rows

for values that are not zero. If a value that is not zero is found, the first tile will be changed to that value, and the found tile will change to have the value zero. This continues until the end of the column. Once sliding in all columns has been applied, tiles are then merged. Merging tiles in this case is achieved by checking each tile in the column and comparing it to the next tile in sequence. If the next tile contains the same value, the first tile has its value doubled, and the tile it was being compared to will change to have the value zero. An amount of points equal to the value of the newly merged tile will then be added to the game's cumulative point total. Once merging has completed, sliding is applied to the grid once again to ensure all tiles cling to each other while aligned with the specified direction.

This process of sliding, merging and sliding again will trigger with each user input, and at least one tile must be altered by this process or the move is not registered as legal and the game will not advance, since no tile will spawn.

An additional function helps to check whether the game is over that runs every turn. This function first checks whether a tile of value zero is present in the grid, as this always means a move is possible. When this condition is fulfilled, it then iterates through the grid, comparing each tile with its adjacent tiles. If two tiles of equal value are found this way, then a merge is possible and the game is not over. If the whole grid is iterated through without finding a possible point of merging, the game is over, and the amount of points earned is displayed.

Integration of depth search within this program is best accomplished with recursive techniques. The program will perform each of the four possible user inputs on copies of the current grid, returning points accrued in each of these copies. This process will iterate upon itself according to the user-specified depth, such that the selected outcome is the one which generated the most points over several turns.

The program will display an average result and a best result from a user-specified amount of random games to serve as a base reference for improved solutions. This is achieved by replacing user input with a random input that always selects one cardinal direction. Additionally, the amount of specified games will be played once more, this time using the depth search method.

Experiments

A number of experiments were conducted to validate the strength of the devised methods. Within the program, calculations are performed using the game's systems both randomly and according to my depth search methods. At depths below two, it is reasonable to be able to expect hundreds of simulated games in a reasonably short length of time. Past a depth of two, it can take several seconds or even minutes to simulate one game. A depth of five is the last reasonable depth to simulate, requiring almost a full minute to fully execute on average. Once the execution is complete, six statistics are provided. Three of these statistics are the average score, best score, and the best board according to the results from the depth search algorithm. The other three statistics are the average score, best score, and best board according to the results from games which experienced entirely random inputs for each of their moves. In order to test the program with different amounts of games simulating up to different depths, the program must be launched again. I have tested the effectiveness of my methods with what I believe to be the highest number of games that can be simulated within a reasonable time frame for each depth, up to a maximum depth of five. The results will be explained in the next section.

Results and Discussion

The baseline perspective for interpreting the results of my methods will be the portion of the program's output that appears in every instance of the program's execution, the random output. For 100 games, the average score was 1080, the best score was 4316, and the best board contained a tile of value 512. The outputs of the depth search up to any depth will exceed each of these statistics, though not every depth can execute 100 games within a reasonable time frame.

The depth search algorithm executing at a depth of zero will simply pick the next move based on which one will generate the most points, defaulting to north if gaining points is not currently possible. For 100 games, the average score was 2923, the best score was 7720, and the best board contained tiles of value 512 and 256.

Past depths of zero, the depth search algorithm will begin testing moves branching from the four currently possible moves. This means moves such as north then north, north then east, north then south, etc., will be tested. The path that gained the highest amount of points overall will be selected and executed. A depth of one will simulate one extra turn in the future, a depth of two will simulate two extra turns in the future, and so on. For a depth of one, over 100 games, the average score was 3311, the best score was 7944, and the best board contained tiles of value 512, 256 and two of value 128. For a depth of two, over 100 games, the average score was 10306, the best score was 23400, and the best board contained a tile of value 2048. For a depth of three, over 20 games, the average score was 24171, the best score was 43636, and the best board contained a tile of value 4096. For a depth of four, over four games, the average score was 36214, the best score was 60272, and the best board contained tiles of value 4096, 2048 and 1024. For a depth of five, it is time consuming to test even just one game, so only one game will

be shown as an example. It achieved a score of 58704 and its board contained at least one tile representing each power of two, up to 4096.

It is plain that the presence of depth and increases to it both provide a noticeable increase in average and best scores. Achieving the presence of 2048 as a tile in the grid is possible with a depth as small as two, and a depth of five can almost achieve 8192 as a tile. This clears my original goal of consistently achieving 2048 using the autoplay method, as that seems true with depths of three and above.

There are two anomalies, however. Depths of zero and one both have very similar results despite my prediction that any amount of depth would be a clear improvement. This could be the result of a programming error or proof that seeing the game one turn in advance does not provide a noticeable improvement. The other anomaly is that the improvement depth search provides appears to plateau at a depth of four. This could be a result of the inability to test a large amount of games, subjecting these results to high variance, or it could be a symptom of the quickly increasing complexity of the game state requiring such highly involved thought that plans must be made more than five turns in advance. It is likely a combination of the two. Aside from these, however, depth search appears to function as predicted.

Conclusions

As each experiment heightened in complexity, there was a clear increase in both effectiveness and time required for execution. Playing games with a depth search parameter set to zero could complete execution within seconds while a depth parameter of five could take a full minute to complete. It is clear that there is room for improvement in that regard.

There is also an issue with the way that the model obtains its conclusions. Since each turn begins with a randomly spawned tile, future simulated turns effectively force a single outcome for these spawned tiles. This means that the program can make decisions based on randomness that occurs multiple turns in advance. This would not be possible in a normal game of 2048. Since the current ideal machine learning model lies outside my current skill set, I had begun to consider alternative models that might apply better to a real-time environment, such as the adversarial search model minimax, but I ultimately decided to stick with the work I had initially planned on performing, because I consider the depth search model to be at least a good proof of concept for a potentially more sophisticated model that is totally true to the rules of 2048.

Within the definition of my version of 2048, the model I created is still able to play remarkably well. My own abilities can only achieve points in the thousands. The depth search model exceeds that standard very early on and quickly earns great scores even at low depths. I believe that fact as well as the many things learned in the pursuit of this goal makes this a worthwhile accomplishment.

References

Cirulli, G. (2014). 2048. https://play2048.co/

Appendix

Results as obtained by program execution:

Random:

```
Average random score was:
1080.2
Best random score was:
4316
Best random board was:
4     2     8     16

2     32    4     2

4     512   16    8

16    8     4     2
```

Depth 0:

```
Average bot score was:
2923.44
Best bot score was:
7720
Best bot board was:
2     64    128   512

4     32    64    256

16    8     16    64

2     4     8     16
```

Depth 1:

```
Average bot score was:
3311.8
Best bot score was:
7944
Best bot board was:
2     4     128   512

16    32    64    256

4     8     16    128

2     4     8     32
```

Depth 2:

```
Average bot score was:
10306.04
Best bot score was:
23400
Best bot board was:
4     2     32    2

2048  256   128   4

4     64    512   8

2     8     16    2
```

Depth 3:

```
Average bot score was:
24171.8
Best bot score was:
43636
Best bot board was:
2     4096  8     4

8     16    32    128

256   512   64    2

2     16    32    4
```

Depth 4:

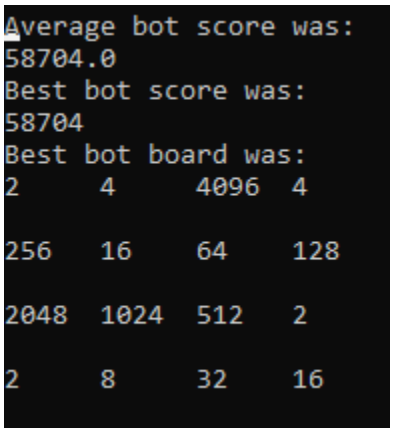
```
Average bot score was:
36214.0
Best bot score was:
60272
Best bot board was:
2     8     4     4096

4     2048  128   1024

16    32    16    2

2     512   4     64
```

Depth 5:


Average bot score was:
58704.0
Best bot score was:
58704
Best bot board was:
2 4 4096 4
256 16 64 128
2048 1024 512 2
2 8 32 16