COMP20003 Assignment 2: Teaching a robot to procrastinate

Benjamin Metha, 681260
methab@student.unimelb.edu.au

October 18, 2016

1 Introduction

In 2014, the game 2048 was published on iOS. With its simple gameplay and calming graphics, the app quickly went viral, becoming one of the most popular free-to-play games on the app store.

While puzzle-goers may have spent hours (or even years)[1] trying to find the best way to beat this game, this kind of problem stretches back to long before developer Gabriele Cirulli was even born. We can model this game as a **graph**, with each board state connected to up to 4 other board states by a subset of the four possible moves (up, down, left and right), and we can find the sequence that leads to the best (highest-scoring) board state using a variant of an algorithm designed in 1956 by Edsger W. Dijkstra.

This paper compares two different possible implementations of Dijkstra's algorithm in the context of this problem. The first chooses each move by looking for the path that finds the node with the maximum score, whereas the second searches for the path with the greatest average score along all of its nodes. These two algorithms were tested against each other several times varying the maximum depth of the paths examined, and the scores achieved by each algorithm were compared.

The program was then improved by implementing some heuristics that I use when playing 2048. These include knowing to keep your best tile in the bottom corner and fill the bottom row with other high numbers, keeping a lot of empty space, and keeping homogeneously increasing sections in order to allow for smoother merges.

2 Experimentation

2.1 Blind search

The program was run 10 times at depths ranging from 0 to 6. For each run, the maximum tile reached, the total score achieved, and the time taken to run the program were all recorded. The average and standard deviation of each of these numbers was calculated. Graphs of the results are shown over the page.

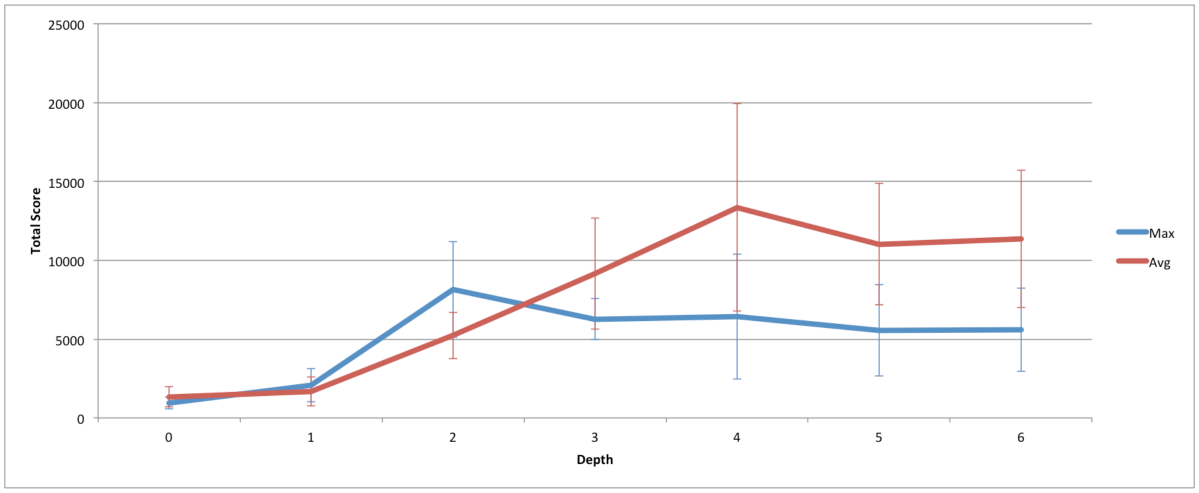

Figure 1: Maximum score v depth

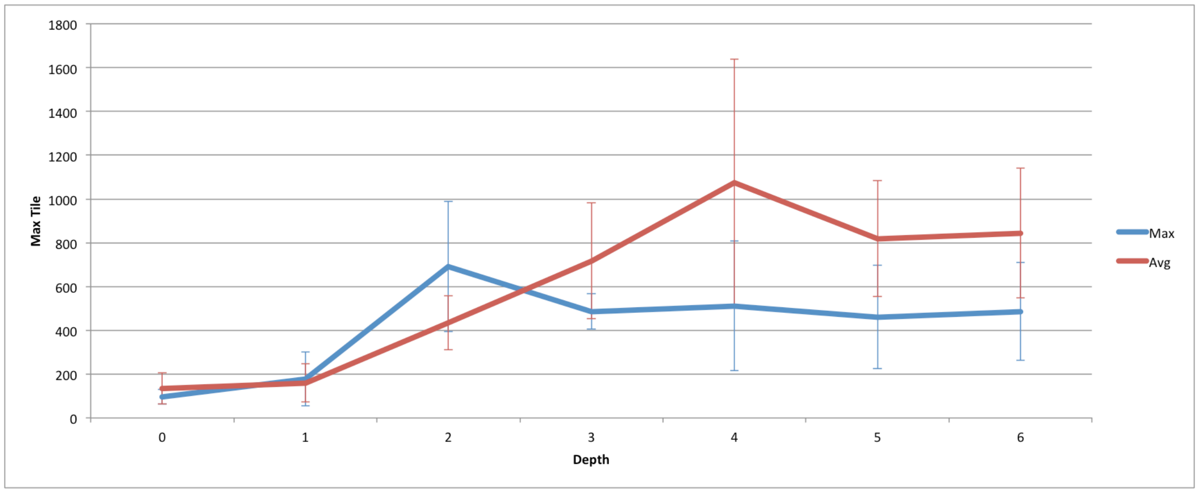

Figure 2: Largest tile achieved v depth

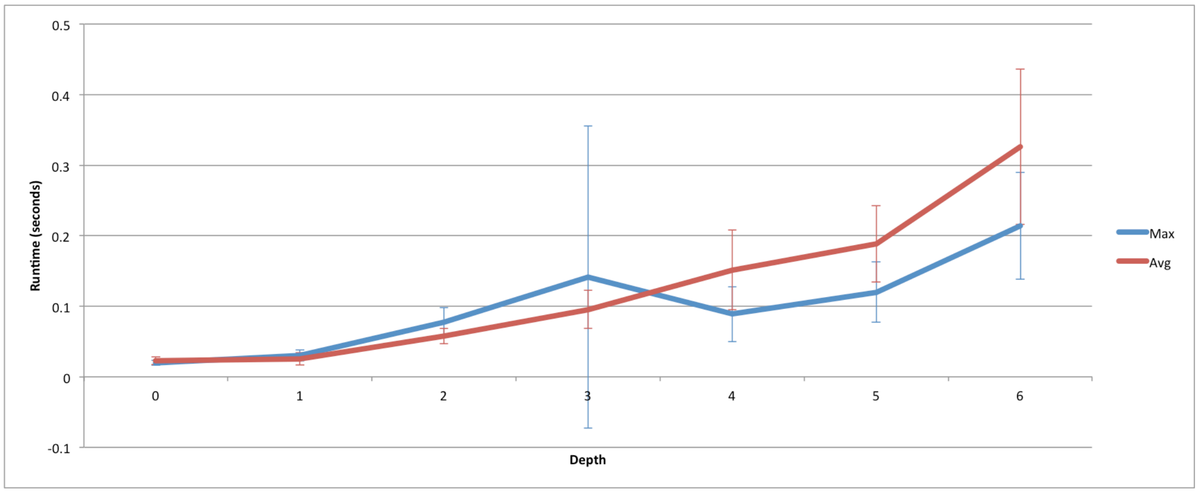

Figure 3: Total runtime v depth

Out of all 140 games played, only 2 of them achieved a victory, creating a 2048 tile before the board was filled. These both occurred when the *Average* propagation setting was used with a maximum depth of 4.

Some trends can be clearly seen by looking at these graphs. Increasing the maximum depth of the paths examined increased the score up to a depth of around 4, after which changing the depth of the path does not seem to really effect the final score. It is also clear that the *Average* propagation mode far outplayed the *Max* propagation node at higher depths.

Another trend that could be very easily seen was how the runtime of this algorithm increased as the depth of the path was increased. There are two reasons for this. Firstly, as the depth increases the program becomes better at playing the game. This makes the games last longer as more moves are taken. Secondly, each time the depth is increased by one the number of nodes examined and generated for every move is also increased, so each individual move also takes longer.

2.2 With Heuristics

To improve this algorithm, I began by teaching it the heuristics that I already knew to use when playing 2048. The first of these was also the simplest: *keep as many empty squares as you can*. Originally I did this by adding a static points bonus (bonuses of 5 points, 25 points and 45 points were all tested), but I always found that this ended up being ignored when the numbers on the board became larger than some threshold value. Instead, I made the size of the points bonus equal to the size of the largest tile on the board. This created a big incentive for the program to keep a lot of empty space at all stages in the game, and also to try to make the largest max tile that it could.

The second trick was one that I had learned from my brother, and also one that can be found on the 2048 tips and tricks website[2]: *keep your highest value tile in the bottom right corner and all your other biggest tiles on the bottom row*. To implement this, I added a bonus to the score equal to $8\times$ the value of each tile on the bottom row and $16\times$ the value of the tile on the bottom right. These coefficients were chosen empirically and not theoretically, as they were the smallest numbers that still allowed for this heuristic to be strictly followed.

Together these rules increased the score of the algorithm, but not to the point where it was consistently winning. After these two were implemented, I tried a lot of different things to try to get the computer to think the same way that I did, including adding bonuses for keeping the tiles on the bottom row in increasing order from left to right, keeping the tiles on the second-to-bottom row in increasing order from right to left, and keeping the tiles in each column in increasing order from top to bottom, but none of these tactics improved the efficiency that much, and some of them even ended up working against me. Rather than continue to try increasingly complicated things, I decided to go online. In a web forum[3] I discovered that all the tricks I had tried to implement were just different, weaker versions of the heuristic called *monotonicity* - simply making sure that every row or column is only increasing or decreasing. The first version of the function I wrote to give bonus points for monotonic rows was a bit faulty - instead, it gave bonuses to any rows or columns with monotonic sections near the edges - but this "faulty" function actually performed better than a functional monotonicity function.

With these three heuristics, the algorithm was run again ten more times on each depth using the more successful average propagation method. These results are compared with the previous results of the algorithm with in the graphs below:

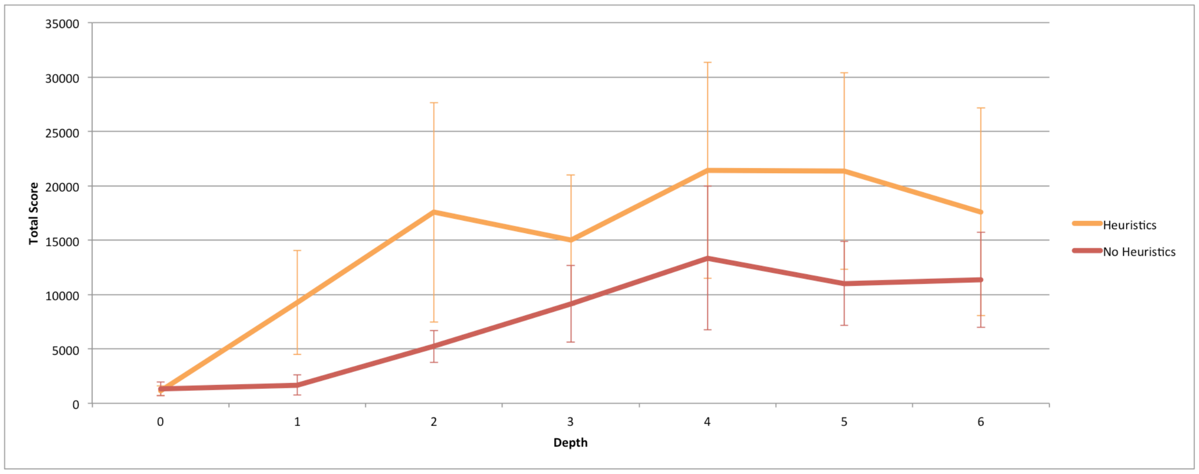

Figure 4: Maximum score v depth

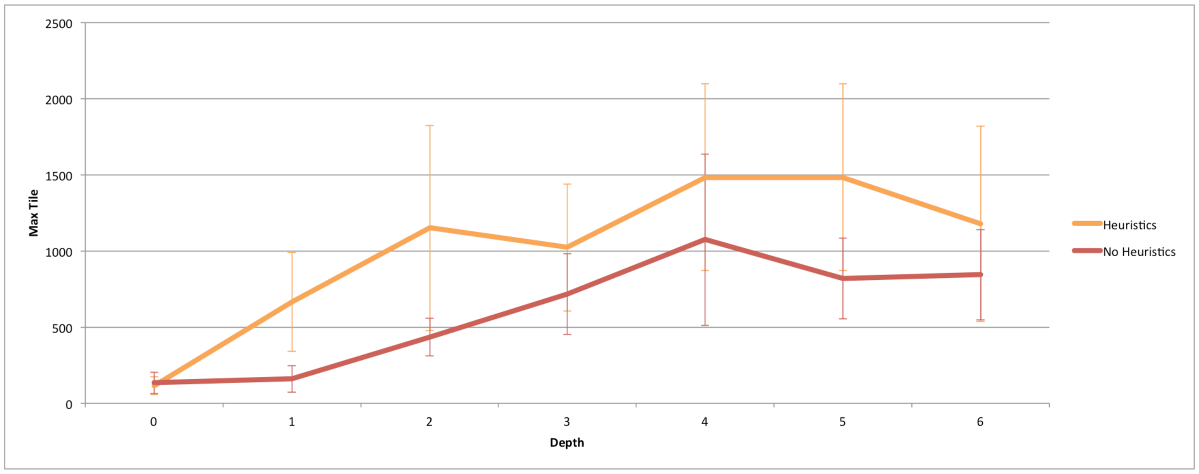

Figure 5: Largest tile achieved v depth

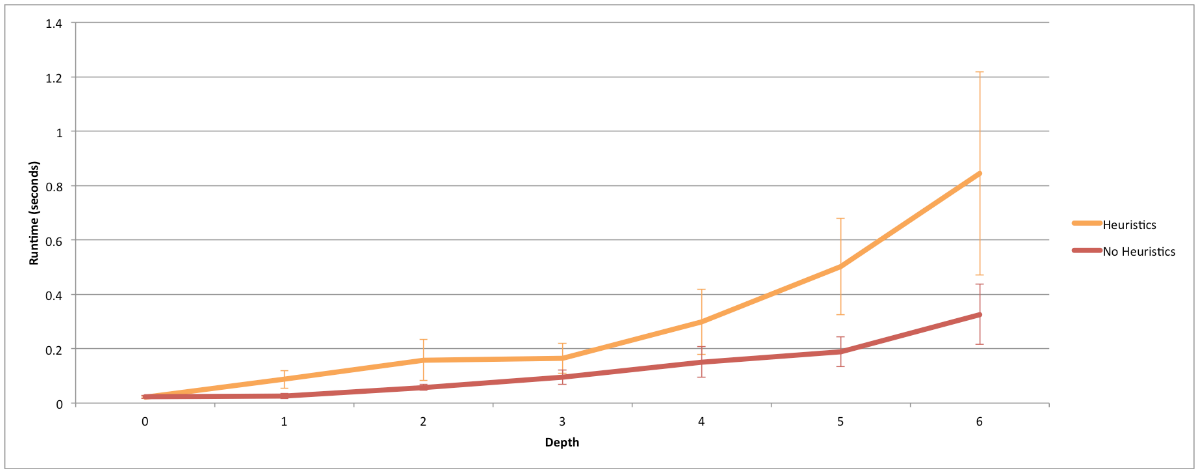

Figure 6: Total runtime v depth

This implementation managed to score the 2048 tile 17 times over the 70 runs, making it 8 times more successful than the algorithm with no heuristics. At depths of 4 or greater, the 2048 tile was achieved around 40-45% of the time. The highest score achieved in the testing was 35,252 points - almost double my personal high score of 20,000.

Though this algorithm did a lot better than the previous one, this success was not very consistent, as can be seen by the large error bars on this graph. Often if a trial got started well, the computer would play in a very similar manner to how I would, but if it got a bad start it would continue making poor decisions, and usually fail after scoring around 7,000 points. Even still, the average score of this implementation was around 10,000 points higher than the blind search algorithm.

Another point to make when comparing this algorithm to the previous one is that this algorithm takes a lot more **time**. A lot of this was due to the games lasting longer, forcing the computer to calculate more moves, but even without that, by comparing games with a similar final score and the same settings, it was clear to see that the games played by this algorithm took approximately twice as long as games played by the other one. For games with depth in this range, however, this is still not an issue, as every game generated in this test was completed in under 1.5 seconds.

3 Discussion

These trials show that using the *Average* propagation method is far more successful than using the *Maximum* propagation method. This makes sense, as making a first move that maximises a board position 3 or 4 steps in advance with no concern for the immediate future is probably not a great tactic.

These tests also show that a few well-chosen heuristics can improve an algorithm greatly. The addition of just three simple heuristics improved the algorithm far more than increasing the search depth. When solving a problem, it is always a good idea to investigate the tactics that are already under use, as this can help to produce better solutions and save a lot of computation time.

If nothing else, this assignment demonstrates the breadth of ways in which graph theory applies to all kinds of problems, and the power that a single algorithm has to solve many seemingly unrelated problems. It also shows that by drawing parallels between two seemingly unrelated problems, a solution to one can be generalised to solve many others. I have a feeling that this technique will come into play many times in my future.

Finally, this algorithm is not perfect. One way in which it could be improved is by accounting for the random appearance of a tile every turn - something the present algorithm currently ignores. There are also a vast number of other, more complex heuristics available on the internet that could also be applied to create a better AI.

Notes

[1]http://www.theepochtimes.com/n3/585996-2048-game-addiction-people-have-already-played-the-equivalent-of-521-years/
[2]http://2048game.com/tips-and-tricks/
[3]http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048