Luke Schipper
Dr. Antonio Bolufe-Rohler
CS4110
18 Oct. 2019

Introduction

Creating an efficient heuristic for solving 2048 is no trivial matter. To solve 2048 perfectly, for example, the minimum number of moves would be $2048/2 = 1024$ moves (when no two's are wasted). Of course, one would expect that this is very difficult to achieve, even with a computer. With a grid size of 4 and a goal value of 2048, the state space for an algorithm to search is in the order of $16 \times 11^{15} + 11^{16}$ (the number of board permutations with 2048 summed with the number of permutations without 2048). The original heuristic presented in assignment #2, which counted the number of tiles, would result in an impossibly long search.

Therefore, a new approach must be found. This assignment will explore heuristics that promote greedily searching for the state with the highest maximum tile value. Progressively larger weights will be assigned to tiles with smaller values to force A* to search along the path of the highest maximum tile. The heuristic will then perform various calculations on the state of the board to offset the value of the weight, allowing A* to make choices that result in fast searches.

The system used to perform these searches is equipped with an Intel Core i7-4790 CPU (8 threads, 4 cores @ 3.60 GHz) and 16GB of RAM. All tests are run in the Windows 10 command line.

Max Tile Weight

The Max Tile Weight heuristic is simplistic in nature. It does not perform calculations on the state of the board besides finding the tile with the highest value. When that value is found, it is assigned an arbitrary weight. It is always better to assign a larger weight to smaller tile values, since A* will then greedily search for the board state with the highest maximum tile. The following Python code shows how weights are calculated:

```python
def generate_tile_weights(goal_value, fn = None):
    weights = dict()
    tile_value = goal_value
    i = 0
    if fn is None:
        fn = lambda t, i, g: g - t
    while tile_value >= 2:
        weights[tile_value] = fn(tile_value, i, goal_value)
        tile_value //= 2
        i += 1
    return weights
```

If the goal value is 64, weights will be assigned to 64, 32, 16, 8, 4, and 2 in that order. Weights are simply a function of the goal value, the value of the tile being assigned a weight, and the counter. In this assignment, all heuristics will be using the following function:

$$f(t,\ i,\ g)\ =\ 2^{floor((g-t)/2)}\ -\ 1$$

The weights for goal value 32 would be:

Tile Value	32	16	8	4	2
Weight	0	255	4095	16383	32767

The Max Tile Weight heuristic simply finds the tile with the highest value for each board state and returns its weight.

```
def maxTileWeight_heuristic(game):
    return tile_weights[max(map(max, game.board))]
```

With such large values for the weights, clearly this heuristic is inadmissible. If, for example, one were to take $2^{1023} - 1$ moves to reach 2048, they would be playing for the rest of their life. If more optimistic weights were used, this heuristic would be admissible. For example,

$$f(t, i, g) = i$$

would assign a weight of 0 to tile value 32 and a weight of 4 to tile value 2. With these weights, the assumption is made that for any maximum tile, a tile of the same value exists and can be combined in one move. This is about as optimistic as one can get. Unfortunately, with these weights the heuristic performs very poorly, since the number of states greatly exceeds the proposed heuristic value. With greedy weights from $f(t, i, g) = 2^{floor((g-t)/2)} - 1$, the heuristic performs as follows:

	Goal: 16 steps/sec	Goal: 32 steps/sec	Goal: 64 steps/sec
[4x4]	11/0.04	20/1.16	--
[8x8]	11/0.10	20/3.19	--

Clearly, the performance of the heuristic is not good enough to solve 2048. It is better than counting the number of tiles, but not by much.

Next Max

The Next Max heuristic attempts to build off the Max Tile Weight heuristic by filtering out undesired board states. Max Tile Weight has a major flaw: two states with the same maximum tile value have the same heuristic value. What if the next maximum tile value in that state is 2? Ideally the search would choose the state that where the next maximum is the highest.

```
def nextMax_heuristic(game):
    temp = [tile for tile_list in game.board for tile in tile_list]
    temp.sort()
    max_tile = temp.pop()
    next_max_tile = temp.pop()
    if next_max_tile == 0:
        return tile_weights[max_tile] + 1
    return tile_weights[max_tile] + math.log(max_tile//next_max_tile, 2)
```

The board is flattened into a one-dimensional list. It is then sorted, and two maxima are taken off. The sum of the weight of the first maximum and the logarithmic difference between the two maxima is returned. Therefore, if the difference between the two maxima is high, the heuristic will punish the board state more. This heuristic could be made admissible with the highly optimistic weights mentioned before. However, it too performs very poorly. With greedy weights, the heuristic performs as follows:

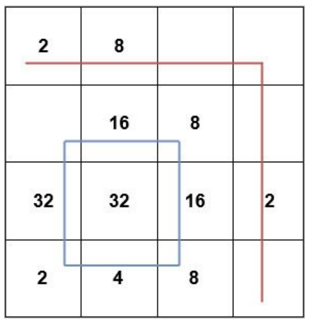
	Goal: 32 steps/sec	Goal: 512 steps/sec	Goal: 2048 steps/sec
[4x4]	20/0.50	315/0.10	1228/5.41
[8x8]	20/1.08	391/0.31	2097/2.07

This heuristic performs very well. In fact, it is the simplest but fastest heuristic presented in this assignment. Why does it perform so well? Firstly, it differentiates heuristic values in the search so that paths with a higher second maximum will be followed. No doubt, board states where the next maximum is low will result in a large subtree of actions that could have been avoided. Secondly, the heuristic is very lightweight, since the most costly operation is sorting. Thus, it is $O(nlogn)$, where n is the size of the board. Interestingly, it is quicker for 512 than it is for 32. This is likely due the logarithmic difference being more punishing, since the range of (2, 512) is much larger than (2, 32).

Adjacent Propagation Order

Adjacent Propagation Order looks to reward good strategy. A good strategy for 2048 is to ensure the largest tile is not blocking the values around it from combining. Thus, it is best for the largest tile to be in a corner, or at least on the side of the board. Furthermore, the tiles that surround the largest tile should be in order from greatest to least so that combinations between two smaller tiles are not blocked by a larger tile.

This heuristic follows the same greedy behaviour as the others. However, it looks at the values surrounding each maximum tile value and punishes the board state if they are not a desired value. Consider the example on the next page:



The heuristic will first locate the indexes of the maximum tile values. In this case, 32 is the maximum tile value and its index is (2, 1). On the first iteration of the heuristic, it will propagate outwards from the maximum and analyze a square of adjacent tiles (blue on the diagram). The max is 32, so the desired value on adjacent tiles is a any tile greater than or equal to 16. The heuristic visits each tile on the square and, if the tile's value is less than 16 and greater than 0, the heuristic computes the logarithmic difference between the tile's value and 16. Thus, the intermediate calculated value is, starting at the empty space and going clockwise: $0 + 0 + 1 + 0 + 1 + 2 + 3 + 0 = 7$. That intermediate value is than added to the weight of the maximum tile to produce a heuristic value of

$7 + weight(32) = 7 + 2^{16} - 1$, if 64 is the goal value. On the second iteration (red on the diagram), the desired value is 8. Starting from 2 and going clockwise, an intermediate value of 2 + 0 + 0 + 0 + 0 + 2 + 0 = 4. Therefore, the final heuristic value of the example board is $4 + 7 + 2^{16} - 1 = 10 + 2^{16}$. See the next page for Python code.

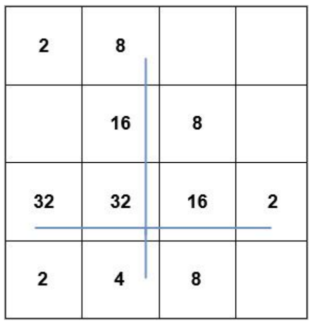
In general, this heuristic promotes larger values in the corner of the board, since it will have less adjacent tiles. It also promotes the greatest to least ordering that was described before. It does not, however, punish empty spaces inside the ordering. The heuristic performs as follows:

	Goal: 32 steps/sec	Goal: 512 steps/sec	Goal: 2048 steps/sec
[4x4]	20/0.24	299/0.20	1112/32.39
[8x8]	20/0.66	417/0.72	2997/5.35

There are some very surprising results here. The heuristic performs very well all around, similar to Next Max. However, it struggles on the 4x4 version of 2048 but not the 8x8 version of 2048. If the board size is increased to 16 with goal 2048, the resulting performance is 1712/9.22. In general, it seems that this heuristic performs better with larger boards. With more rows and columns to process, the propagation might be finding more accurate heuristic values for board states. Otherwise, it is a rather costly heuristic at $O(n^2)$, since it must be run for each maxima.

Adjacent Line Order

The last heuristic presented in this assignment, Adjacent Line Order, attempts to combine some of the concepts from previous heuristics. It is similar to the Adjacent Propagation Order heuristic. Consider the example below:



The Adjacent Line Order heuristics will grab four lists in total: One for all tiles directly north of the maximum value of 32; similarly, it will grab lists for east, west, and south.

North	South	East	West
[32, 16, 8]	[32, 4]	[32, 16, 2]	[32, 32]

Adjacent Propagation Order (previous heuristic)

```
def adjacentPropagationOrder_heuristic(game):
    h = float("inf")
    max_tile_indexes = find_max_tile_indexes(game)
    for index in max_tile_indexes:
        desired_tile = game.board[index["row"]][index["col"]]
        temp_h = tile_weights[desired_tile]
        box_border = {
            "w": index["col"],
            "e": index["col"],
            "s": index["row"],
            "n": index["row"],
        }
        while is_valid(box_border["w"], game) or is_valid(box_border["e"], game) or
is_valid(box_border["s"], game) or is_valid(box_border["n"], game):
            desired_tile //= 2
            box_border["w"] -= 1
            box_border["e"] += 1
            box_border["s"] += 1
            box_border["n"] -= 1
            for i in range(box_border["n"], box_border["s"] + 1):
                if is_valid(i, game) and is_valid(box_border["w"], game):
                    tile = game.board[i][box_border["w"]]
                    if tile < desired_tile and tile > 0:
                        temp_h += math.log(desired_tile//tile, 2)
            for i in range(box_border["n"], box_border["s"] + 1):
                if is_valid(i, game) and is_valid(box_border["e"], game):
                    tile = game.board[i][box_border["e"]]
                    if tile < desired_tile and tile > 0:
                        temp_h += math.log(desired_tile//tile, 2)
            for j in range(box_border["w"], box_border["e"] + 1):
                if is_valid(j, game) and is_valid(box_border["s"], game):
                    tile = game.board[box_border["s"]][j]
                    if tile < desired_tile and tile > 0:
                        temp_h += math.log(desired_tile//tile, 2)
            for j in range(box_border["w"], box_border["e"] + 1):
                if is_valid(j, game) and is_valid(box_border["n"], game):
                    tile = game.board[box_border["n"]][j]
                    if tile < desired_tile and tile > 0:
                        temp_h += math.log(desired_tile//tile, 2)
        if temp_h < h:
            h = temp_h
    return h
```

It then performs two calculations on each list and computes their sum. One of these is Hamming distance. Each list is sorted from greatest to least and the number of tiles that are out of place is returned (i.e. [16, 2, 8] would result in a Hamming distance of 2). The second calculation is "sorted distance", which is similar in concept to the logarithmic difference in Adjacent Propagation Order. See the following code:

```
def determine_sorted_distance(sorted_list):
    distance = 0
    for i in range(len(sorted_list) - 1):
        if sorted_list[i + 1] <= 0:
            return distance
        distance += math.log(sorted_list[i]//sorted_list[i + 1], 2)
    return distance
```

The calculation is:

$$\sum_{i=0}^{length(sorted\_list)-2} log_2(floor(sorted\_list(i)/sorted\_list(i+1)))$$

Therefore, the final calculations for the above example will look like this:

	North: [32, 16, 8]	South: [32, 4]	East: [32, 16, 2]	West: [32, 32]
Hamming:	0	0	0	0
Distance:	1 + 1 = 2	3	1 + 3 = 4	0

A goal of 64 results in: $Total = 2 + 3 + 4 + weight(32) = 2^{16} + 8$. The Python code is as follows:

```
def adjacentLineOrder_heuristic(game):
    h = float("inf")
    max_tile_indexes = find_max_tile_indexes(game)
    for index in max_tile_indexes:
        north_list = [game.board[i][index["col"]] for i in range(index["row"], -1, -1)]
        south_list = [game.board[i][index["col"]] for i in range(index["row"], game.dimension)]
        east_list = [game.board[index["row"]][j] for j in range(index["col"], game.dimension)]
        west_list = [game.board[index["row"]][j] for j in range(index["col"], -1, -1)]
        w_h = determine_hamming_distance(west_list) + determine_sorted_distance(west_list)
        e_h = determine_hamming_distance(east_list) + determine_sorted_distance(east_list)
        s_h = determine_hamming_distance(south_list) + determine_sorted_distance(south_list)
        n_h = determine_hamming_distance(north_list) + determine_sorted_distance(north_list)
        temp_h = w_h + e_h + s_h + n_h + tile_weights[game.board[index["row"]][index["col"]]]
        if temp_h < h:
            h = temp_h
    return h
```

The differences between this heuristic and Adjacent Propagation Order are as follows:

- Adjacent Line Order punishes empty spaces (with Hamming distance)
- Adjacent Line Order is harsher on inorder lists.
  (i.e. For [32, 16, 8, 4], APO = 0 while ALO = 1 + 1+ 1 = 3)
- Adjacent Line Order does not consider tiles that are diagonally adjacent to the maximum tile

In general, Adjacent Line Order will promote the strategy of having the maximum tile in the corner. However, it will more aggressively pursue lines of tiles with higher values. This results in a performance improvement:

	Goal: 32 steps/sec	Goal: 512 steps/sec	Goal: 2048 steps/sec
[4x4]	21/0.34	315/0.14	1228/6.08
[8x8]	21/1.16	391/0.43	2097/2.40

Clearly, Adjacent Line Order performs consistently well, similar to Next Max. It is better than Adjacent Propagation Order for the reasons it is different. It promotes the same strategies while being slightly less costly. The diagonal adjacent tiles are not as important to visit, since the board can only move North, East, West, and South. Even with a board size of 32 and a goal of 2048, the heuristic finds a solution in 10 seconds.

Summary

Heuristic [4x4]	Goal: 32 steps/sec	Goal: 256 steps/sec	Goal: 512 steps/sec	Goal: 1024 steps/sec	Goal: 2048 steps/sec
Max Tile Weight	20/1.16	--	--	--	--
Next Max	20/0.50	229/0.08	315/0.10	690/1.00	1228/5.41
Adjacent Propagation Order	20/0.24	271/0.16	299/0.20	673/0.73	1112/32.39
Adjacent Line Order	21/0.34	229/0.09	315/0.14	690/1.25	1228/6.08

In conclusion, it seems simpler is better in regard to 2048 heuristics (at least for this assignment). Next Max consistently performs best, with Adjacent Line Order a close second. Max Tile Weight performs rather poorly, but its greedy concept is used in the next three heuristics. Again, Next Max seems to perform better since its lightweight and greedy towards high-valued boards. While Adjacent Propagation Order and Adjacent Line Order are intricate and smart with their calculations, their overhead might be too much to be efficient. It might be worth adjusting how much either algorithm punishes the board state for undesirable qualities. This may result in improved performance overall.