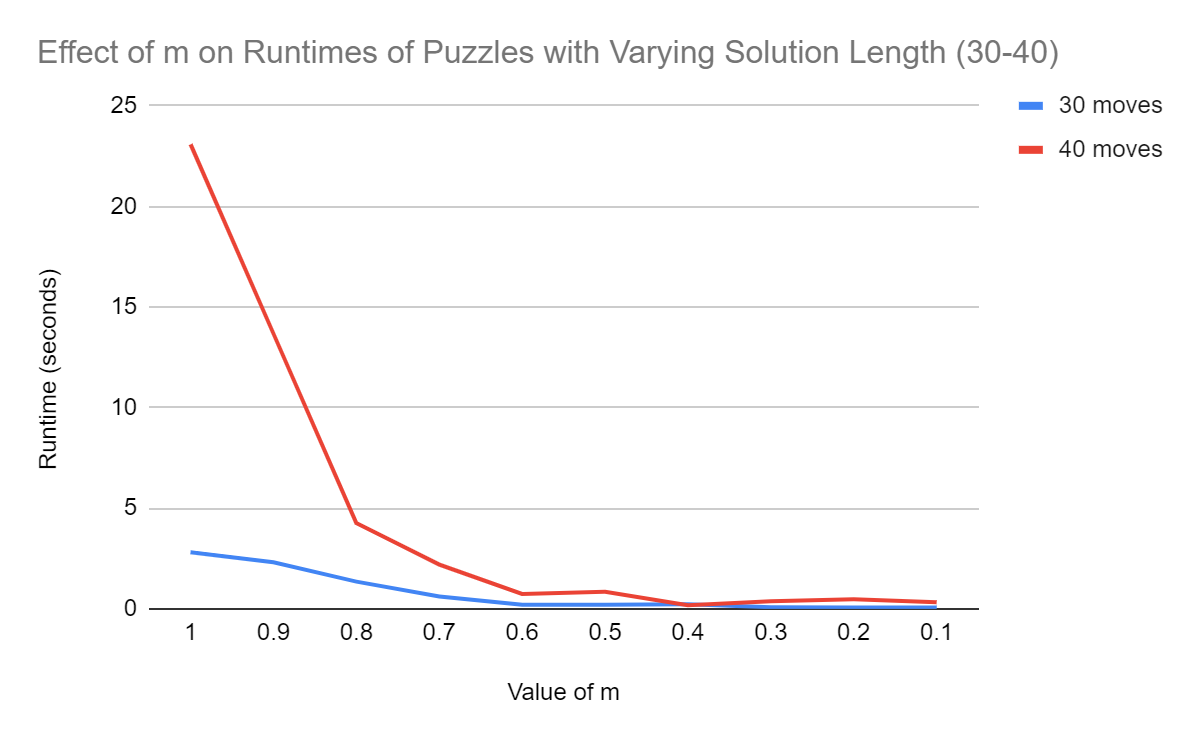
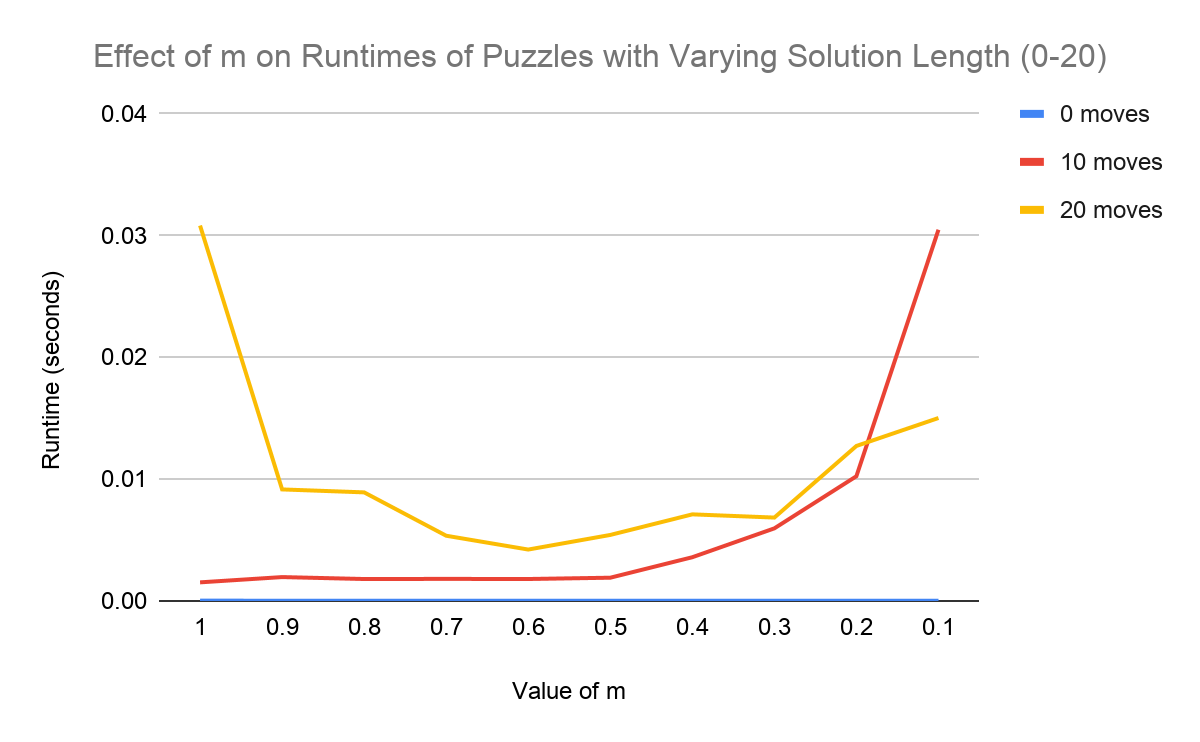
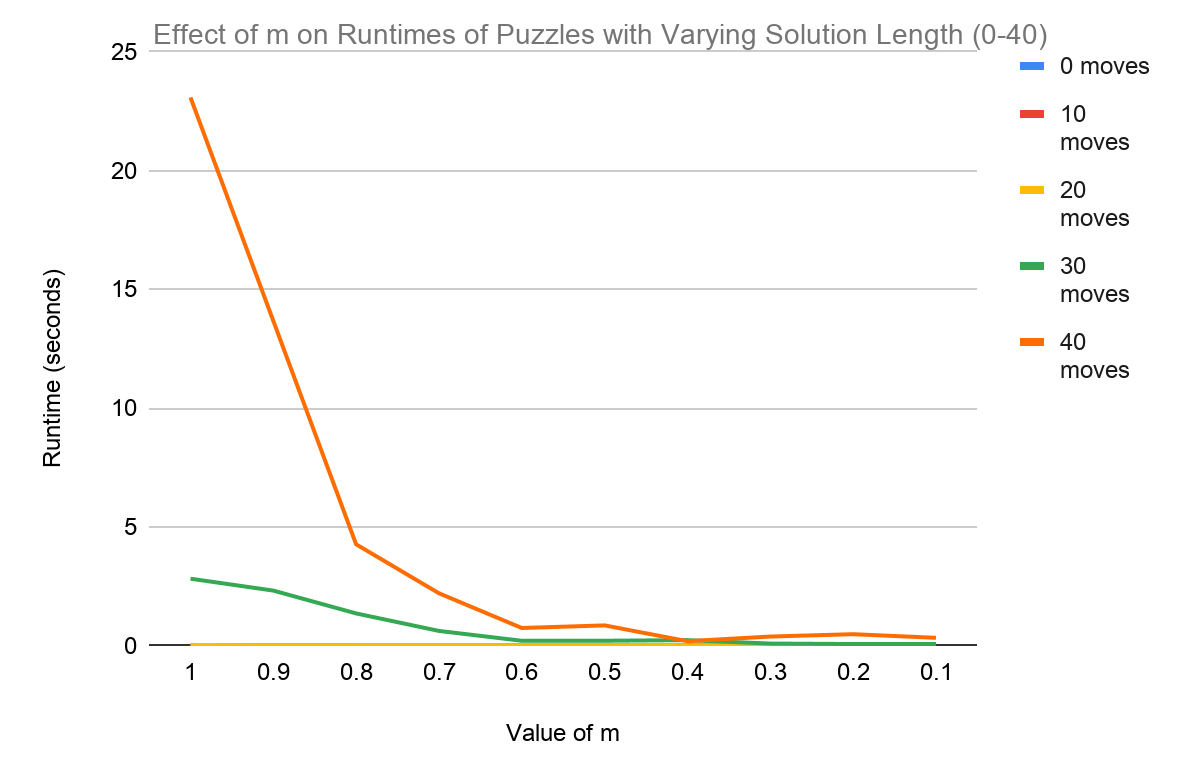
**Sliding Puzzles Part 3 Writeup: 4th Exploration**

Note: All data was generated from puzzles in the 15\_puzzles.txt file.

Extension A:



The effect of decreasing *m* on A\* runtime depended on puzzle solution length. As illustrated above by Graphs 1 and 3, puzzles with solution lengths greater than or equal to 30 showed fairly consistent decreases in runtime as *m* dropped. The magnitudes of the runtime decreases appear to best fit an exponential decay curve, first rapidly decreasing with *m* and later leveling off. Among the 0 to 20 move puzzles, however, the effect of *m* on runtime is more unpredictable (see Graph 2). The 20-move puzzle runtimes most closely follow the pattern established by those of the 30- and 40-move puzzles. They conform to an exponential decay curve until they begin gradually increasing after reaching an *m* of 0.6. Diametrically opposed to all other puzzle runtimes, the 10-move puzzle runtimes best fit an exponential growth curve, gradually then rapidly increasing after an *m* of 0.5. The 0-move puzzle runtimes, with an overall range of only 1.3400001 \* 10-5 seconds, did not significantly change with decreases in *m*.

|  |  |
| --- | --- |
| Puzzle Solution Length (moves) | Lowest Accurate *m* |
| 0 | 0.1 |
| 10 | 0.1 |
| 20 | 0.2 |
| 30 | 0.5 |
| 40 | 0.6 |

Similar to A\* runtime, solution length accuracy was influenced by both puzzle solution length and *m*. Decreasing *m* had more drastic effects on the accuracy of longer length puzzles. As shown in the table above, varying *m* appeared to have no effect on puzzles of solution lengths 0 and 10 moves. Starting with the 20-move puzzle, however, accuracy began to be compromised at higher *m* values as puzzle solution length increased.

Extension B:

|  |  |  |  |
| --- | --- | --- | --- |
| Effect of *m* on Accuracy of a 55-move Puzzle with Random Tie-Breaking | | | |
|  | Value of *m* | | |
| 0.6 | 0.5 | 0.4 |
| Average Deviation from 55 (moves) | 4 | 10 | 18.267 |
| Standard Deviation (SD) | 0 | 1.63 | 3.751 |

|  |  |  |  |
| --- | --- | --- | --- |
| Effect of *m* on Accuracy of a 55-move Puzzle without Random Tie-Breaking | | | |
|  | Value of *m* | | |
| 0.6 | 0.5 | 0.4 |
| Average Deviation from 55 (moves) | 4 | 8 | 22 |
| Standard Deviation (SD) | 0 | 0 | 0 |

The accuracy advantage of random tie-breaking becomes apparent as path lengths grow and *m* decreases. Weighted A\* inflates path lengths because multiplying depth by a value of *m* less than 1 causes it to undervalue the relative “cost” of depth and eventually give inaccurate “shortest” paths. The inaccuracy of a weighted A\* without a random tie-breaking feature is consistent from run to run because the algorithm will always find the same incorrectly long path. Random tie-breaking, however, allows weighted A\* to find different paths from run to run and subsequently has the potential to give more accurate results over multiple runs. The likelihood of random tie-breaking improving accuracy grows with path length. As path length increases, weighted A\* will process more nodes. The more nodes processed, the more ties and thus opportunities to randomly choose more accurate paths are likely to be encountered.

The tables above suggest the path length-dependent nature of random tie-breaking’s accuracy advantage. Lowering *m* artificially increases path length. Accordingly, random tie-breaking gained an accuracy advantage only when weighted A\* was run with the lowest value of *m* tested. At higher values of *m*, random tie-breaking either “tied” or performed less accurately than straightforward weighted A\*.

Extension D:

|  |  |  |
| --- | --- | --- |
| Algorithm | Nodes Processed per Second | Lowest Puzzle Solution Length over 10 Seconds (moves) |
| ID-DFS | 91954.16533675838 | 17 |
| BFS | 65256.5435967227 | 19 |
| A\* | 6757.346142654169 | 35 |

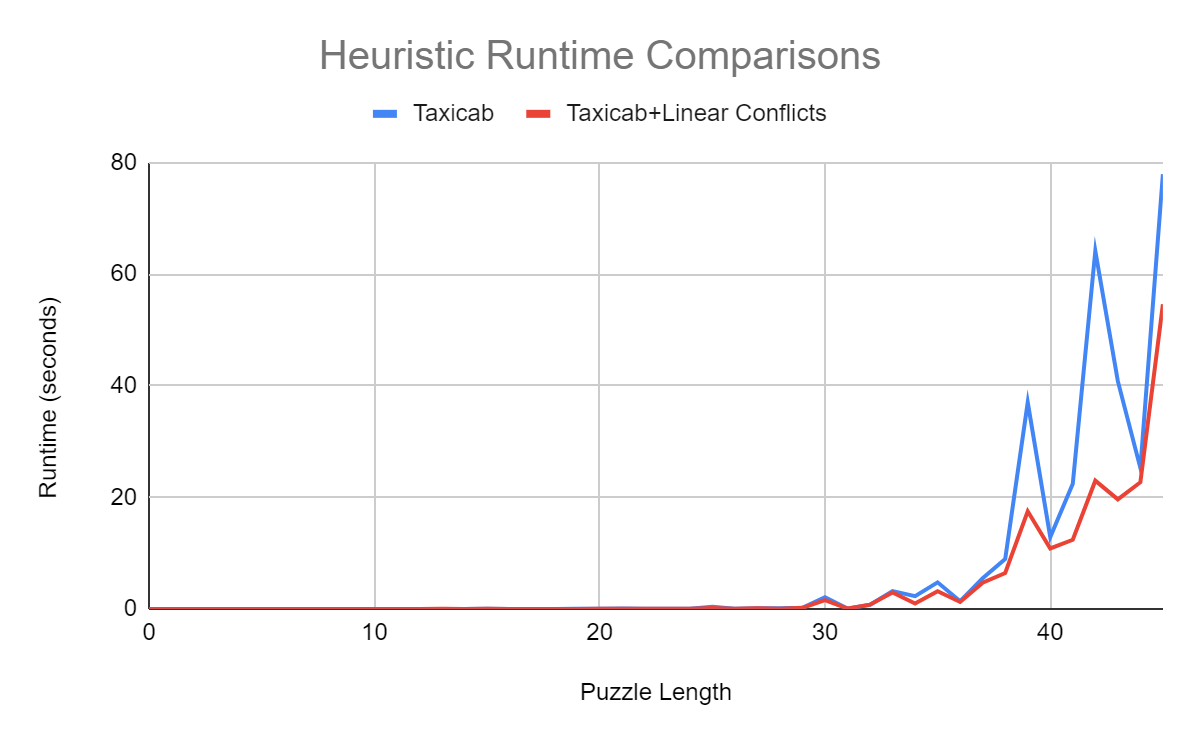
As shown in the above table, ID-DFS processed the most nodes per second while A\* processed the least. This observation is not unexpected. ID-DFS is quite memory efficient, does not perform any time-costly operations when processing nodes, and often reprocesses nodes it has already encountered. Subsequently, much of its runtime is “spent” processing nodes rather than updating memory structures or calling node-evaluating methods. A\* has nearly the opposite characteristics. Although its delayed redundancy checking does force it to partially reprocess some nodes, A\* maintains a function-scope set of (“closed”) nodes already on its heap that greatly limits its node processing redundancy. Its dual data structure requirements and constant calls to a heuristic function method force it to “spend” runtime on operations other than node-processing, lowering its node-processing rate. BFS shares qualities of both algorithms that may explain its intermediate node-processing rate. Like ID-DFS, BFS does not call node-evaluating methods; like A\*, BFS requires two function-scope data structures and largely limits redundant node-processing.

The above table reports an additional observation for each algorithm: the lowest puzzle solution length that required runtimes over 10 seconds. ID-DFS had the lowest, first reaching over 10 seconds of runtime with a 17-move puzzle, while A\* had the highest, requiring more than 10 seconds with a 35-move puzzle--the exact opposite ordering to that of node-processing rate. The observations perhaps suggest that node-processing rate has an inverse relationship with algorithmic efficiency (as measured by greatest path length reached in a given amount of time).

Extension C:

I developed a “new” heuristic that was the sum of taxicab distance and linear conflicts. As briefly discussed in class, linear conflicts arise when tiles in the proper row or column are reversed in order. Linear conflicts cause taxicab distance to severely underestimate the remaining distance to the solution as they require more moves than predicted to solve. Adding linear conflict count to taxicab distance helps account for the “extra” moves, resulting in a more accurate heuristic.

My implementation of linear conflict counting used 2 methods: an overhead “setup” to facilitate counting and an actual counter. The setup was passed a puzzle state as its singular argument, found the goal state, and calculated the size of the puzzle. Based on the goal state and puzzle size, it filled 2 global lists, 1 for rows and 1 for columns, with sets of the letters that would appear in each row/column when the puzzle was solved. (Lists were chosen for their built-in indices; sets, for efficient membership testing.) Similarly, the counter calculated puzzle size based on a passed puzzle state argument. Looping through each row and column, it increased a count variable by 2 every time a pair of non-blank tiles were in the correct row or column but wrong order. Every possible pair of tiles was checked without redundancy, and the global lists generated by the setup were used to determine whether or not tiles were in the proper rows and columns. The count variable was ultimately returned. Within the new A\* method, the setup was called once, passing the start puzzle state, while the counter was called with taxicab distance every time algorithm required heuristic calculation.



As shown by the above graph, the efficiency improvements were overall modest. At best, they were by factors of 2-3. The greater the taxicab distance runtime, the greater the degree to which my new heuristic was more efficient. Runtime generally increased with puzzle solution length, so efficiency improvements tended to increase also as puzzles grew larger. My heuristic therefore only significantly outperformed taxicab distance when runtimes of the latter heuristic rose above 10 seconds, beginning with puzzles with solution lengths greater than 30. I believe my somewhat inefficient implementation--using nested for-loops--of linear conflict counting may have limited my efficiency gains.