# COMP 472
# ARTIFICIAL INTELLIGENCE

Mini Project 1

## The 11d-Puzzle

Submitted to:
Professor Leila Kosseim

By:
Ryan Nichols #29787739
Jiayin Liu #27532628

Fall 2018

Due: Oct. 15th

**Heuristics:**

Heuristic 1: Hamming Distance
The Hamming Distance heuristic gives the total amount of puzzle pieces that are not in their goal position, it is implemented by counting the differences between current state and goal state.

This method is admissible since it does not overestimate the cost, as the cost is calculated in a straightforward way; is monotonic, because the distance between 2 positions does not change; and is somewhat informed by representing how organized/disorganized the current state is.

Heuristic 2: Manhattan Distance
The Manhattan Distance represents the sum of moves that every out of placed piece needs to move to its goal position. In the context of this project, the Manhattan Distance heuristic also needs to consider diagonal moves in addition to horizontal and vertical moves. The heuristic is implemented by computing the horizontal and vertical differences between a piece's current and goal positions, and the largest of two differences corresponds to the piece's Manhattan Distance.

The Manhattan Distance heuristic is admissible, the calculation of Manhattan Distance is reliable and does not overestimate while having considered diagonal moves; it is also monotonic, as the results are consistent; it is more informed than Hamming Distance since it gives insight on the cost to reach the goal state.

**Difficulties:**

The major difficulty with this project was to get depth first search (DFS) working. The only time DFS would find a solution, we only let the program run for a few minutes max, was when we tried it on small boards like a 2x2. This is because the branching factor is very low and DFS can quickly find the useless leaf nodes and then continue on in search of the solution.

Increasing the size of the board to 3x4 causes DFS to run endlessly. This is because the game as we play it has a high branching factor. If the zero is near the center of the board there are 8 possible moves. Along the sides there are 5 moves and in the corners there are 3 moves available. On the 11-d board there are 4 corner squares, 6 side squares and 2 middle squares. Calculating the weighted average[1] yields a branching factor of roughly 5. On the short time scale where we're willing to wait for a solution, with that branching factor, DFS goes deeper and deeper down paths with no solutions and the run time of the program becomes prohibitively long.

[1] https://cs.stackexchange.com/questions/39534/how-to-find-the-branch-factor-of-8-puzzle

We tried limited depth first search and had similar problems. Either we didn't find a solution or our limit was high enough the program took too long to run. We tried iterative deepening and were able to find solutions for relatively simple problems like the example of a starting board given in the problem description. With iterative deepening the algorithm can't endlessly continue down useless branches though the runtime is somewhat long because the algorithm has to repeat the same calculations for the previous depths every time a new depth is tried.

Another difficulty we had was in sorting the open list for A* as well as best first search (BFS) and then breaking ties. At first we attempted to use the *.sort()* method available in the python language but there was no obvious way to break ties. Next we attempted to hand code a bubble sort, with the addition of a tie breaker based on a nodes latest move, but this extended the runtime of the program too much because bubble sort runs O(n^2). Finally we arrived at the in built *sorted()* [2] method which is capable of sorting tuples by each element in order. In our case we needed a tuple with 3 elements. The first is either f(n) or h(n) depending on the algorithm, the second is the tie breaker which is a value corresponding to the node's latest move and third the node itself.

This configuration led to another problem; because *sorted()* sorts by every element in the tuple in order from first to last which is exactly why we needed it but it also tries to sort the third element of type node which isn't sortable. We were able to pass as a sorting key, a function that only returns the first 2 elements of a tuple and finally we had a sorting solution that broke ties and didn't prohibitively add to the runtime of the program.

Some difficulties emerged as well during the implementation of the heuristics. The incorporation of diagonal moves was not considered at first, and it had caused search algorithms that use heuristics (Best First Search and A*) to run for undetermined time without producing solution for some cases. This problem was first tackled using search depth that limits the search from producing more children after a certain depth, so that the search can go onto other branches sooner. However, setting search depth limit did not produce any noticeable amelioration since child nodes cumulate exponentially, resulting searches continued to run for unknown length.

Through debugging, it was found that the result produced by the heuristics were not representative to the situation, and the problem was identified: the classic Manhattan Distance and Sum of Permutation Inversions calculations were targeted at puzzles with only horizontal and vertical moves allowed. The Manhattan Distance algorithm has been updated to take diagonal moves into consideration; the Sum of Permutation Inversions algorithm however has been replaced with Hamming Distance, an admissible option in this context, since we were unable to find the proper alteration to the original Sum of Permutation Inversions heuristic which often overestimates the cost.

[2] https://www.programiz.com/python-programming/methods/built-in/sorted

**Analysis:**

Through many modifications and trials, it is found that the more inversely ordered the puzzle's start state is compared to its goal state, the more time consuming are the searches and more lengthy the results. For example:

#1
```
starting_list = [1, 0, 3, 7,
                 5, 2, 6, 4,
                 9, 10, 11, 8]
goal_state = [1, 2, 3, 4,
              5, 6, 7, 8,
              9, 10, 11, 0]
```

With a start state ordered in a similar way as the goal state, using BFS, both heuristics take much less than a second to complete the search, and both provide short solution path (6 moves).

#2
```
starting_list = [1, 0, 3, 7,
                 5, 2, 6, 4,
                 9, 10, 11, 8]
goal_state = [11, 2, 3, 4,
              5, 6, 7, 8,
              9, 10, 1, 0]
```

When the pieces 1 and 11 in the goal state switch positions, the BFS then needs at least 2 seconds to compute the result (using Manhattan Distance), and the solution path is 19 moves long.

Following the example above, when the search is lightweight (start state's pieces close to their goal position), the execution time and the solution path of both heuristics are identical. For highly disordered puzzles (pieces far from their goal position), the Manhattan Distance heuristic proves to be more efficient than Hamming Distance in both time consumption and solution quality.

Result of BFS on example #2:

- Hamming Distance: ~10s execution time, 25 moves
- Manhattan Distance: ~2s execution time, 19 moves

The difference in efficiency is due to the informedness of the heuristics. In the case of example #2, the start state is considered highly disordered as pieces are located far from their goal position, and the evaluation by Manhattan Distance heuristic represents such high disorder by producing a higher score (cost to reach goal state). Since this heuristic is based on the number of moves away from goal position, the pieces 0, 1, and 11 weigh in heavier in the Manhattan Distance score as they are the furthest from their goal position.

Meanwhile, using the Hamming Distance heuristic, the furthest pieces 0, 1, and 11 are weighted the same as the rest of out-of-place pieces, which does not estimate the cost

to reach goal state accurately enough. Thus, while computing child nodes, the Hamming Distance heuristic cannot identify the better child(ren) from children having the same number of misplaced pieces, whereas Manhattan Distance has a better chance to tell one cheaper child node from the more costly ones.

Similar results were obtained when comparing the two heuristics on the A* algorithm. On the relatively simple starting point in example #1, the A* algorithm with Hamming distance finds a solution in .00431 seconds and 5 moves. Using A* and Manhattan Distance, again the solution is found in 5 moves, this time taking .0021 seconds. Iterative deepening was able to find the solution in 0.21 seconds in 5 moves as well. These results were obtained on a different machine and the times shouldn't be compared to the testing done for BFS.

When testing the A* algorithm on the more difficult starting position using example #2, the solution was found in 36 seconds in 16 moves. However, when using the Hamming Distance, a solution was not found within 5 minutes, so testing on this combination was stopped. With the hamming distance it's possible to get stuck in a local minimum that isn't near the global minimum of the solution space. If one needs to reduce the heuristic score before improving it to find the global minimum then all possible solution paths will be explored before doing this due to the nature of the A* algorithm. Similarly iterative deepening did not find a solution in a reasonable amount of time either.

In sum, if the start state is similarly ordered as the goal state, it is safe to use either Hamming Distance or Manhattan Distance. In other cases, Manhattan Distance will be the preferred heuristic.

**References:**

[1] Wikipedia. (2018). A* search algorithm. [online] Available at: https://en.wikipedia.org/wiki/A*_search_algorithm [Accessed 30 Sep. 2018].

[2] Stack Overflow. (2018). How can I sort tuples by reverse, yet breaking ties non-reverse? (Python). [online] Available at: https://stackoverflow.com/questions/18414995/how-can-i-sort-tuples-by-reverse-yet-breaking-ties-non-reverse-python#18415016 [Accessed 3 Oct. 2018].

[3] Programiz.com. (2018). Python sorted() - Python Standard Library. [online] Available at: https://www.programiz.com/python-programming/methods/built-in/sorted [Accessed 3 Oct. 2018].

[4] Class notes on state space search, algorithm for DFS

[5] En.wikipedia.org. (2018). Iterative deepening depth-first search. [online] Available at: https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search [Accessed 12 Oct. 2018].