1 Introduction

Sliding-tile puzzles present an interesting platform for search algorithm analysis. I will discuss and compare several strategies and optimizations including choice of heuristic and memory of visited states. For 3x3 puzzles the most difficult puzzles have an optimal solution of 31 moves and for 4x4 puzzles the most difficult puzzles have an optimal solution of 80 moves. While my program is able to solve any 3x3 puzzle with any search strategy in a reasonable amount of time, 4x4 puzzles are more challenging: With informed search strategies the most difficult puzzle I have been able to solve has an optimal solution of 78 moves, and with uninformed strategies (And Hamming heuristic) the limit is 31 moves. I have implemented the programs in JAVA JDK8. Since JAVA does not keep track of exact memory usage, memory usage is estimated and represents an upper bound of actual memory usage.

2 Strategies

1. **Breadth-first search (BFS)**
   BFS is an "uninformed" search that is guaranteed to find an optimal solution if one is present. BFS searches every node of a certain distance before it expands the frontier to the next higher cost. In order to avoid potential endless loops, a history of already visited nodes must be utilized to filter frontier expansion to only unvisited nodes. Because of the required history and the full graph expansion, this strategy requires considerably more memory and computation than the other strategies.

2. **Depth-first search (DFS)**
   The basic strategy of DFS is to expand children nodes before sibling nodes. While DFS finds an optimal solution for some puzzles, there are many for which it explores branches that lead to poor solutions before it searches the optimal branch. With a hard limit set to 50 moves for 3x3 puzzles, the DFS search took as much as 40 moves (vs. max 31 optimal) to solve some puzzles. Note that the performance of a DFS search is dependent on the number of (possibly non-optimal) moves in the first solution that it finds, and not dependent on the board-specific optimal number of moves.

3. **A* (AST)**
   A* is an "informed" search that relies on some heuristic function to estimate the cost required to reach a goal state from frontier nodes in order to schedule frontier nodes to expand. Performance of A* is highly dependent upon the accuracy of heuristic function used. The typical heuristic is a sum of the already sunken cost to reach the node, and an estimate of how much more cost is required to reach the goal. A* is guaranteed to find an optimal solution if one exists, if the cost for each move is constant. For these sliding-tile puzzles, each move has a cost of one, therefore A* search will find optimal solutions.

4. **A* Dumb (ASD)**
   This is an A* search which does not consider memory of already visited states to determine which frontier nodes to expand. Instead, it only prevents backtracking to the immediate grand-parent node. My initial thought was that the lack of an additional data structure to track the visited states would result in a smaller memory load than A*, but the additional inefficiency results in double the number of frontier nodes expanded, which results in a similar total memory usage. It is slightly more time intensive than regular A*.

5. **Iterative Deepening A* (IDA)**
   This is a search which utilizes a heuristic function that is the same as A*, but instead of filtering frontier nodes to only those previously unexplored, it filters them to only those below a threshold estimated cost. When the frontier becomes exhausted, IDA increases the threshold and begins the search again from the initial state. Because it does not have to maintain a set of explored states or a widely dispersed frontier, it uses significantly less memory than other searches presented here. Due to the duplication of exploration of the shallow nodes on each iteration, IDA expands more nodes than AST, but still manages to outperform AST in computation speed and memory usage.

3 Heuristics

1. **Inversions test for solvability**
   Before embarking on a search for a solution to a given puzzle, it is useful to be able to determine whether such a solution actually exists. For sliding-tile puzzles, there are two loop invariants involving the parity of the inversion count which allow us to test whether an initial game state is solvable:
   (a) For odd dimension boards, valid moves do not change the parity of inversion count. Since the goal state has 0 inversions, this means that all states with an even number of inversions are solvable, and the remaining are unsolvable.
   (b) For even dimension boards, the row position of the empty cell has the same parity as the number of inversions.
2. **Hamming distance (HAM)**
   This is the most basic heuristic that I analyzed. It counts the number of cells that are out of place. Since an out of place cell is always at least one move away from its goal position, and cells always swap with the empty cell, this heuristic always underestimates cost and is therefore admissible.
3. **Manhattan distance (MAN)**
   This is a heuristic which works for measuring distance on a 2D plane in which discrete moves must be in one of four cardinal directions. Conveniently, this describes the sliding-tile puzzle. The algorithm measures the sums of the horizontal distances and the vertical distances between the locations of tiles and their goal locations, disregarding any obstacles in their paths. Since the puzzle is full of potential obstacles, it represents a lower bound on the number of moves required to move all tiles into their goal positions, and is therefore an admissible heuristic.
4. **Manhattan distance with interference (INT)**
   This is a heuristic that takes the Manhattan distance concept and extends it with the idea of obstacles. For any tile that is on its home row or column but not in its goal position, any other tiles that are also in their home row or column and blocking the first tile from reaching its goal location are said to be 'interfering' with the first tile. For any such interference to be overcome, at least two additional moves are required to go around and reach the goal state. Thus by adding 2 to the cost for any interferences encountered, this is an admissible heuristic.

4 Optimizations

I used several optimization techniques in the implementation:

1. Storing board state as a primitive array of bytes - This allows the use of several optimized JAVA structures including `ByteBuffer` and `ByteArrayOutputStream` which do not copy the underlying array data, allowing the board state data to be shared between the visited set and the fringe queue. This also is the most compact way to represent the board state without relying on encoding / decoding to a single integer representation, which would require overhead to convert absolute board positions to relative positions and vice-versa, and limit the board size to 4x4. Using bytes, the board size limit is 11x11.
2. I used an iterative method to calculate the Manhattan distance and interference instead of fully calculating it for each new fringe state. This requires 2/9ths the computation for a 3x3 puzzle and 1/8th the computation for a 4x4 puzzle. Before I implemented the lookup tables it resulted in about 15% performance boost, but the lookup tables negate most of the positive effect of this.
3. Using lookup tables for `isValidMove()`, `colOf()`, `rowOf()`, and `manhattan()` - This avoids costly integer modulo and divide instructions, eliminates branch prediction penalties, streamlines instructions into sequences of load/store and basic arithmetic, potentially allows widespread SIMD instruction use, and potentially allows fringe exploration to occur entirely in cache.
4. Returning an iterator of successor nodes - This allows fringe exploration to occur without unnecessary copying of nodes. The exact successor node that is tested for validity is passed onto the fringe queue by reference.

5 Conclusions

1. **DFS is non-optimal**
   The first result I will discuss is the non-optimal nature of the DFS search. If I didn't institute a hard cap on the depth of the fringe stack, DFS can result in some wild results - sometimes thousands of moves deep. This also causes some misleading information on my visualization (Figure 1), as the x-axis readings are "stretched out" by the amount that DFS is non-optimal.

2. **BFS is painfully slow and memory intensive**
   Although my algorithm is otherwise optimized enough to allow BFS to process all valid 3x3 games, I was only able to solve 4x4 games up to 19 moves before running out of memory. Therefore I did not include BFS in the visualization of 4x4 strategies.

3. **Memory of prior nodes visited keep bad algorithms viable**
   The only reason that BFS and DFS perform as well as they do to solve all 3x3 puzzles is the implementation of prior location memory. Without this, their time / memory complexity would grow far to fast to solve any puzzle over 15 moves.

4. **A* with Hamming distance is terrible but usually better than BFS**
   For 3x3 games, A* with Hamming distance is mostly better than BFS until the most difficult games. For 4x4 games, it is memory efficient enough that I can solve up to 30 move games, but the growth in time taken is still troubling. IDA* used with Hamming has improved time efficiency, but is still as memory intensive as A*, and thus fails at the same game complexity.

5. **Interference (INT) is awesome, but is not synergistic with location memory**
   The results for both time and memory efficiency of implementing the Manhattan + Interference memory are outstanding, but negates any positive impact gained from implementing a memory of prior visited locations. This leads me to conclude that prior memory is a good crutch to allow a bad strategy or heuristic to remain viable for somewhat complex problems, but it is not a replacement for a better heuristic.

6. **Manhattan distance (MAN) seems great until you discover a better alternative**
   The first algorithm I implemented was AST-MAN, and then I implemented the slower performing algorithms. In this context, it is easy to feel like MAN is a superior solution, because it is clearly superior to BFS, DFS, and A* with HAM. However, once I successfully implemented Manhattan with interference (INT) and began performance tests, I realized how expectation is colored by prior experience. MAN is easy to implement, but is not a very good heuristic in the grand scheme of things.

7. **Iterative Deepening and Interference (IDA-INT) is the best combination**
   While INT improves the A* performance considerably, the additional performance achieved by the synergistic combination in IDA-INT allows almost all 4x4 puzzles to be solved, including the easy to remember 78 step puzzle `15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0`. Since I could only solve that puzzle with IDA-INT and not the other strategies, I did not include it in the comparison visualization (Figure 2). I have included a file `4x4-big.txt` with the 80 step puzzles I was still unable to solve with IDA-INT.

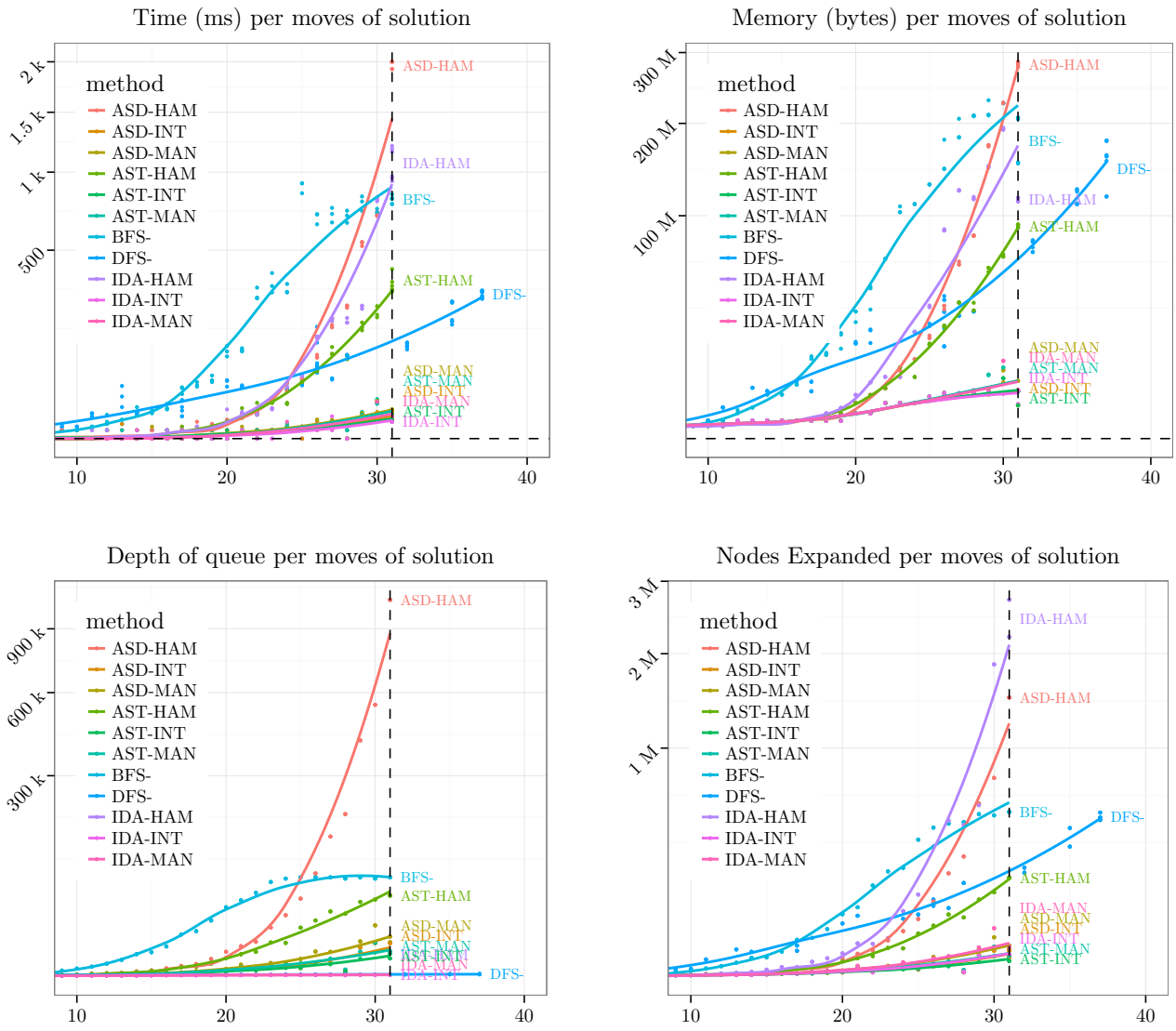

Figure 1: Performance comparison of methods to solve 3x3 puzzles. Sqrt scale.

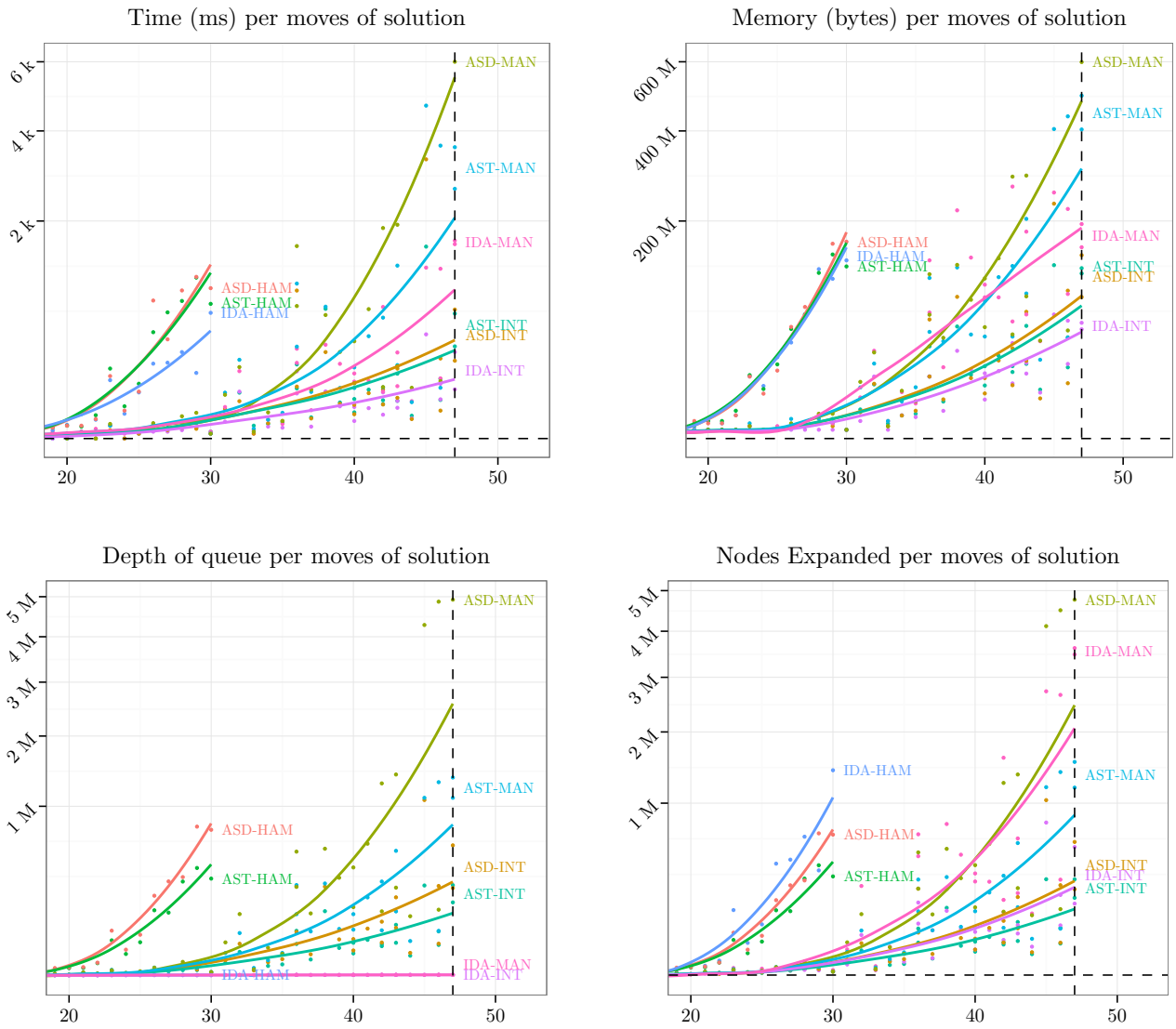

Figure 2: Performance comparison of Heuristics in A* to solve 4x4 puzzles. Sqrt scale.