Sliding Tile Puzzle

Eden Chung (ec3661), Maxine Tamas (mt3634)

November 16th, 2025

1 Context

(a) Unsolved (b) Solved

Figure 1: Images of the Sliding Tile Puzzle

The Sliding Tile Puzzle, invented by Noyes Chapman in 1880, is a puzzle that consists of a $n \times n$ grid, with one tile missing. The player's goal is to use this empty spot to slide the tiles around in order to reach the end configuration. The end configuration normally consists of all tiles arranged in increasing orderTiles cannot be lifted off the board, so the only possible move at each point in time is sliding a tile of the player's choosing into the empty slot.

The most common version of this puzzle is known as the 15 tile puzzle, consisting of a 4×4 grid, with 15 tiles. However, the puzzle can be expanded or shrunk to any size grid.

However, there are several starting configurations that lead to unsolvable puzzles. Therefore, to achieve a starting configuration, one must begin with the solved form of the puzzle then randomly scramble.

2 Algorithm

The Sliding Tile Puzzle has an existing algorithm to determine the correct sequence of moves to achieve the end state. To do so, the puzzle can be represented as a graph (more specifically, a tree), where each node's child represents the game board after a specific tile movement.

We have found an existing Python implementation [2] that will find the correct steps for configurations of any grid, where the game board size n can be modified. This Python implementation uses an iterative deepening DFS approach.

We will use this as a reference to ensure that our Haskell versions are working as expected.

The following code is a small snippet of the Python algorithm

```
def solve(board, maxMoves):
    print('Attempting to solve in at most', maxMoves, 'moves
        ...')
    solutionMoves = []
    solved = attemptMove(board, solutionMoves, maxMoves,
        None)

    if solved:
        displayBoard(board)
        for move in solutionMoves:
            print('Move', move)
            makeMove(board, move)
            displayBoard(board)

        print('Solved in', len(solutionMoves), 'moves:')
        print(', '.join(solutionMoves))
        return True
    else:
        return False


def attemptMove(board, movesMade, movesRemaining, prevMove):
    if movesRemaining < 0:
        return False

    if board == SOLVED_BOARD:
        return True

    for move in getValidMoves(board, prevMove):
        makeMove(board, move)
        movesMade.append(move)

        if attemptMove(board, movesMade, movesRemaining - 1,
            move):
            undoMove(board, move)
```

```
            return True

        undoMove(board, move)
        movesMade.pop()
    return False
```

Due to the recursive nature of this algorithm, as each node (game state) could have up to 4 possible moves, meaning up to 4 child nodes, the time complexity is exponential, meaning for larger game states, the sequential approach can be very slow. Therefore, developing a parallel approach will be important.

Our Haskell implementation will be loosely based off this Python version, however, we will use a slightly different recursive approach called iterative deepening A* (otherwise known as IDA*). IDA* allows us to create heuristics, strategies that help us determine whether a given game state is closer or further away from the end goal. For example, a heuristic could be the Manhattan distance between each tile's current position and their goal position. The heuristics are then used to run DFS on the "best" possible option at any given time, ideally shortening the number of states that have to be explored.

3 Parallelization Opportunities

We plan to analyze the results of several different parallelization approaches to determine what the best approach is. We want to be intential with our parallelization approaches and tactically choose approaches that use parallelization to optimize performance but not to a point where we apply too much parallelization where the system is overworked.

For all of the strategies, we will use `par`, `rpar`, `rseq`, as well as the `Par` Monad.

3.1 Divide and Conquer Approach

In this approach, we would expand from the root and compute different subbranches in parallel. For example, we will precompute several states to a very shallow depth, so this would be quick. For example, we would precompute up to 8 leaf nodes. Then we can run IDA* from each of these leaf nodes, as they can be treated as independent subproblems. This would work similar to DLS (depth-limited search) where a system runs its algorithm up to a certain limit to save memory. However, in this case instead of stopping completely at the limit, we would run IDA* on the different child nodes.

3.2 Work-Pool Approach with the `Par` Monad

We would also consider a work-pool style approach built around the `Par` monad. In this design, we first generate a batch of intermediate states at a chosen depth (similar to the divide-and-conquer strategy), but instead of assigning each state to a specific thread manually, we place these states into a logical "pool" of tasks where each task is assigned to a core whenever a core is freed up. For example, if we have around 100 nodes to go through and 4 cores, these nodes would cycle through the cores and be computed whenever they become available.

4 Data & Evaluation

We will run several experiments. We will pre-compute a valid start state by starting with a solved board and applying a random number of moves. This guarantees that our start state is solvable and we can then use this start state for all experiments. We will also set the game board size to be large so that we can evaluate the speedup between number of cores used. This way, with consistency in board solvability and difficulty, and with a large enough state space where we could see a difference in performance, this puzzle initialization ensures the best set up.

With this set up, we can analyze the differences in the different parallelization strategies, as well as the different heuristics. We would do this by comparing different parallelization implementations to the non-parallelized IDA* implementation using an admissible heuristic (ex: number of incorrectly placed

tiles). Each parallelization strategy would vary by strategy itself, heuristic, cores, etc.

This experimental design allows us to quantify (1) whether the parallel solvers remain correct and optimal, and (2) how effectively each parallelization approach improves performance.

References

[1] http://www.murderousmaths.co.uk/games/loyd/loydfr.htm

[2] https://inventwithpython.com/recursion/chapter12.html