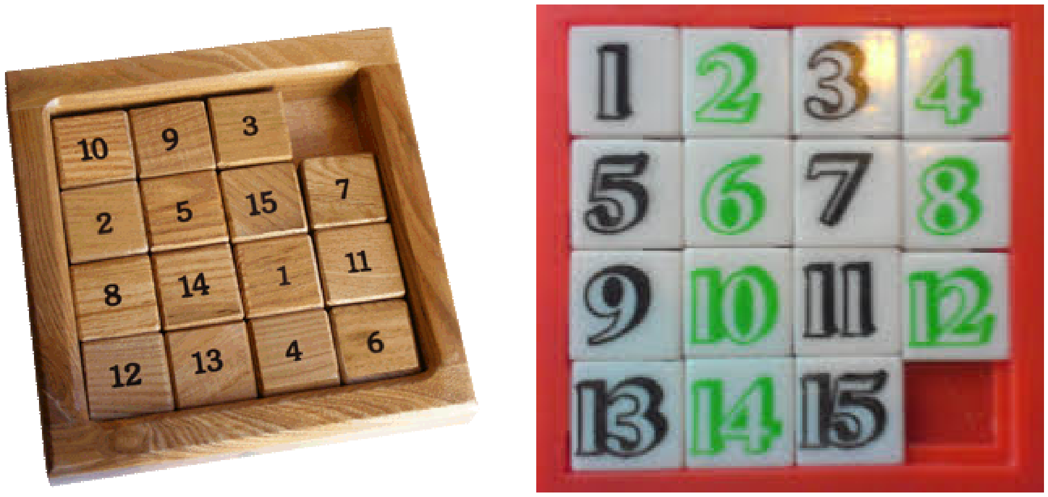
Sliding Tile Puzzle

Eden Chung (ec3661), Maxine Tamas (mt3634)

December 17th, 2025

1 Introduction



(a) Unsolved (b) Solved

Figure 1: Images of the Sliding Tile Puzzle [1]

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 order. Tiles 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. For this project, we will specifically focus on this 4×4 grid, as it provides the ideal amount of depth to make parallelization interesting.

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. In this project, we will try to solve boards of varying depth, ranging from boards with only 3 moves to solve, all the way to 40 moves to solve.

2 Algorithm

The Sliding Tile Puzzle can be solved by iterating over all the possible moves until reaching the solution state. This means we can theoretically use any AI game solver algorithm (if we don't consider exponential state trees), such as Breadth First Search, Depth First Search, and any other similar algorithms. To run these algorithms, the puzzle can be represented as a graph (more specifically, a tree), where each node represents the current game board, and each node's child represents the game board after a specific tile movement.

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

We used this as a reference to ensure that our Haskell versions are working as expected by comparing the output moveset generated by our Haskell algorithms with this Python algorithm.

The following code is a small snippet of the Python algorithm

```python
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 when solving deeper boards (ie boards that require more moves to reach the goal state).

We loosely based our Haskell implementation off this Python implementation, but explored using various different AI solving approaches such as Iterative Deepening DFS, Iterative Deepening A*, along with experimenting with pruning.

3 Haskell Game State Representation

In order to create an AI solver for this game, we need to set up the basic infrastructure and data types that we will be using to store the information in the game.

We need to represent the $n \times n$ board. This can be thought of as a grid, similar to a chessboard, which we flatten out into a one dimensional array in Haskell. We also assign this to the type synonym `Board` for readability.

```haskell
type Board = [Int]
```

Each position in the array contains an integer representing the tile value at that location. The blank tile is represented using 0.

For example, the end goal state is represented as:

```haskell
endStateBoard :: Board
endStateBoard =  [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0]

-- or, to better visualize the 2D board, this is equal to:
endStateBoard :: Board
endStateBoard = [1,2,3,4,
                5,6,7,8,
                9,10,11,12,
                13,14,15,0]
```

In a programmatic representation of the Sliding Tile game, it is difficult to represent the act of "sliding" or "swapping". Instead of the act of swapping, we will think of it as a movement into the empty tile. For example, when we perform the move `MoveLeft` on a Board state, this represents the tile on the right of the blank moving left into the blank tile (thereby swapping with the blank). So, there are at most 4 possible moves at any given game state, Left, Right, Up, Down. We create a new Haskell datatype for easy reference.

```haskell
data Move = MoveLeft | MoveRight | MoveUp | MoveDown
  deriving (Show, Eq, Generic, NFData)
```

After running any of our algorithm solvers, we will get a result that shows how many moves it took, as well as the sequence of moves. An example is listed below

```
Found solution in 30 moves:
[MoveDown,MoveRight,MoveRight,MoveRight,
MoveDown,MoveLeft,MoveDown,MoveLeft,
MoveLeft,MoveUp,MoveRight,MoveRight,
MoveUp,MoveRight,MoveUp,MoveLeft,MoveLeft,
MoveLeft,MoveDown,MoveDown,MoveDown,
MoveRight,MoveRight,MoveRight,MoveUp,
MoveUp,MoveUp,MoveLeft,MoveLeft,MoveLeft]
```

4 Sequential Algorithm

4.1 Iterative Deepening Depth First Search

We first began by implementing a basic algorithm, Depth First Search (DFS). As the name suggests, this algorithm explores the tree depth first, going all the way to a leaf node before backtracking and trying a different possible move.

However, we use a slight variation on DFS to ensure we get the most optimal solution. The approach, called Iterative Deepening DFS (IDDFS) is a mix between a Breadth First Search approach and Depth First. IDDFS starts at a given depth, ie a depth of 1, and tries all possible movesets from the start configuration at that depth, then if a solution is not found, will try at depth + 1. This ensures that if a solution is found, it is guaranteed to be the optimal solution, since we increase the depth by 1 each time.

This is a very naive approach as there is no specific logic for what move it will pick next, ie, it is not guided by any knowledge of the game. Instead, it will always expand a node in the same order of Left, Right, Up, Down. So, we can definitely improve on this approach with a few simple steps such as adding in a heuristic that helps guide what state to explore next.

4.2 Iterative Deepening A*

Iterative Deepening A* (IDA*) is a very similar approach as IDDFS, but instead of naively selecting the next move to explore, it picks the "best" one according to a heuristic.

In this specific game, the heuristic chosen is the Manhattan distance of each tile to its final game state position. This allows the algorithm to differentiate a game state that is "better" or "worse" by seeing how close it is to the goal solution state.

4.3 Iterative Deepening Depth First Search With Pruning

When running iterative DFS, we realized that, especially for boards that require a lot of moves to solve, it took incredibly long to run sequentially (for example, our implementation did not solve the 30-move board after more than two hours). This is mostly due to the fact that IDDFS must explore every branch fully for each depth limit, which leads to a lot of repetition. Because the depth limit d, d starts at 0 and increases by 1 for every iteration, many of the same subtrees are revisited, even if they clearly do not lead toward the goal.

Therefore, to combat long execution times, we can add pruning, which allows the algorithm to eliminate branches that will never lead to an optimal solution. To perform this pruning, we can use the same heuristic as IDA*: the Manhattan distance of the board. The Manhattan distance provides an estimate of "how many moves remain," and allows us to measure the quality of a state (how close we are to a solution) without having to finish the whole tree. However, it's important to note that this is still different from IDA* even though they use the same heuristic: IDA* sorts nodes by heuristic whereas IDDFS with pruning still remains depth-first, we just use the heuristic to decide if we can skip exploring a subtree.

We decided to keep IDDFS and IDDFS with pruning because pruning significantly optimizes the sequential solution which could lead to parallel optimization being less effective. Therefore, we wanted to keep and compare all sequential and parallel solutions to see where parallelization would make the greatest difference.

4.4 Heuristic

The heuristic function we used for pruning in IDDFS and IDA* was written in Haskell as:

```haskell
getManhattanDistanceOfBoard :: Board -> Int
getManhattanDistanceOfBoard b = distanceHelper (numTiles - 1) 0
  where
    distanceHelper :: Int -> Int -> Int
    distanceHelper index total
      | index < 0      = total
      | currTile == 0  = distanceHelper (index - 1) total
      | otherwise      = distanceHelper (index - 1) (total + newDist)
      where
        currTile  = b !! index
        row       = index `div` boardSize
        col       = index `mod` boardSize
        goalIndex = currTile - 1
        goalRow   = goalIndex `div` boardSize
        goalCol   = goalIndex `mod` boardSize
        newDist   = abs (row - goalRow) + abs (col - goalCol)
```

This heuristic computes the Manhattan distance of the entire board by summing, for each tile, the difference between its current row/column and its goal row/column. This essentially allows us to gauge how “close” the current configuration is to the solution state.

This simple equation (and the use of heuristics in general) leads to a significantly better performance as it reduces the amount of branches explored that aren’t getting closer to the solution state.

5 Parallel Algorithm

While our sequential implementations (IDDFS, IDDFS with pruning, and IDA*) can solve many boards, their performance still worsens as solution depth increases. Therefore, we can use parallelization in each of these approaches to see how it improves performance. All of our parallel implementations use Haskell’s `Control.Parallel.Strategies` library and rely on the same underlying game representation and move generation as the sequential versions.

At a high level, all of our parallel algorithms share a similar pattern:

- We expand the search tree from the root to a small “split depth” (for example, depth $d = 3$).
- This produces a frontier of intermediate nodes, where each node consists of a board state and the list of moves taken from the root to reach it.

- Each frontier node is then treated as an independent subproblem and solved in parallel using the corresponding sequential search routine.
- The final solution is chosen by taking the shortest successful move sequence from all subproblem results.

All the parallel algorithms maintain both correctness and optimality. Solutions are still guaranteed to be optimal as pruning is based off an admissible heuristic which ensures that no optimal solutions are pruned.

5.1 Strategies Considered

5.1.1 First Iteration of Parallelization: Naive Depth-Based Parallel Search

Initially, contrary to the parallel approach discussed above, we attempted to parallelize *within every depth limit* of iterative deepening, which meant for each depth d, we created parallel tasks over the nodes reachable at depth d, collected the results, and then repeated this entire process again when increasing d to $d + 1$. This caused unnecessary overhead because the same subtrees were re-expanded and re-parallelized at every iteration of the depth limit, so we did not end up adopting this strategy. We therefore had to make some slight changes from the sequential versions because this first naive parallel implementation performed poorly.

Therefore, we changed the parallelization strategy so that the frontier is generated only once at a shallow "split depth", as discussed. Instead of parallelizing inside every depth iteration, we parallelize across the subtrees rooted at that split depth and reuse those same frontier nodes for all subsequent depth limits. This eliminated execution time substantially.

5.1.2 Second Iteration of Parallelization: Top-Level Parallelism (Split-Depth Parallel Search)

As explained above, in our new approach, we parallelize by expanding the tree to a designated, shallow fixed depth (such as $d = 3$). Then, we generate the frontier, a list of nodes `(Board, [Move])`, which keeps track of the board and the moves taken so far. Once we have these nodes, we delegate each node its own parallel branch. Then, once all solutions are returned, we choose the shortest valid solution to be the final solution. While this strategy does not speed up the work performed inside each subtree, by taking a divide-and-conquer approach, the overall execution time decreases. Ultimately, the speed of this approach is limited by the number of frontier nodes (which is impacted by the chosen depth d) and the cost of solving each subtree. This ended up being the strategy we used for most of the algorithms.

```haskell
type Node = (Board, [Move])

getNodesFromRoot :: Board -> Int -> [Node]
getNodesFromRoot start d0 =
  go [(start, [])] d0
  where
    go frontier 0      = frontier
    go frontier depthN = go nextFrontier (depthN - 1)
      where
      nextFrontier = concatMap expandNode frontier

expandNode :: Node -> [Node]
expandNode (b, path) = [(applyMove b m, path ++ [m]) | m <- validMoves b ]
```

This datatype `Node` and function `getNodesFromRoot` allow us to represent each intermediate search state as `(Board, [Move)` and generate all states for the given depth limit. Starting from the root board with an empty move list, `getNodesFromRoot` repeatedly calls `expandNode` to apply every valid move and generate all possible states given the move sequences.

5.2 Parallel IDDFS

```haskell
parallelIddfs :: Board -> IO ()
parallelIddfs startBoard = iter 0
  where
    splitDepth = 3 --this can be modified
    frontier = getNodesFromRoot startBoard splitDepth

    iter depthLimit
      | depthLimit < splitDepth =
          case iddfsWithDepthLimit startBoard depthLimit of
            Nothing      -> iter (depthLimit + 1)
            Just solPath -> ...
      | otherwise = do
          let remaining = depthLimit - splitDepth
              leafResults =
                withStrategy (parList rdeepseq)
                  [ fmap (prefixMoves ++)
                      (iddfsWithDepthLimit board remaining)
                  | (board, prefixMoves) <- frontier
                  ]
              solutions = [ s | Just s <- leafResults ]
          case solutions of
            []    -> iter (depthLimit + 1)
            sols  -> ...
```

As mentioned earlier, parallelization only occurs after the algorithm hits the split depth. Before this, each subtree is still solved independently using sequential depth-limited IDDFS. Parallelization is implemented using `withStrategy (parList rdeepseq)` which ensures that each subtree is fully evaluated and that no lazy evaluation delays occur. Without this, Haskell's lazy evaluation could cause subtrees to remain as unevaluated thunks. Therefore, with this as a safeguard, we can wait for all subtrees finish, filter for failed branches, append the successful solutions to their root move prefixes, and choose the shortest and best solution.

5.3 Parallel IDDFS with pruning

```haskell
parallelIddfsPruning :: Board -> IO ()
parallelIddfsPruning startBoard = iter 0
  where
```

```
    splitDepth = 3 --this can be modified
    frontier = getNodesFromRoot startBoard splitDepth

    iter depthLimit
      | depthLimit < splitDepth =
          case iddfsPruningWithDepthLimit startBoard depthLimit of
            Nothing      -> iter (depthLimit + 1)
            Just solPath -> ...
      | otherwise = do
          let remaining = depthLimit - splitDepth
              leafResults =
                withStrategy (parList rdeepseq)
                  [ fmap (prefixMoves ++)
                      (iddfsPruningWithDepthLimit board remaining)
                  | (board, prefixMoves) <- frontier
                  ]
              solutions = [ s | Just s <- leafResults ]
          case solutions of
            []    -> iter (depthLimit + 1)
            sols  -> ...
```

Parallelization follows similar structure here as it does for standard IDDFS. However, since we use a pruning-based sequential solver, this makes the parallelization significantly faster as well. The main benefit of this approach compared to standard IDDFS parallelization is that pruning significantly improves runtime. However, it is important to compare the same sequential versions to their parallel versions because pruning improves efficiency for both the sequential and parallel implementations, this may mean that the parallelization is less effective relatively.

Analyzing the data, which we will go more into depth later in section 6, we found that IDDFS has a 2.72x speedup and IDDFS with pruning has a 2.04x speedup. Both parallel approaches improved their respective sequential performance significantly, but we can say that the standard IDDFS had a slightly higher speedup as it explores more branches.

5.4 Parallel IDA*

```
parallelIDA :: Board -> IO ()
parallelIDA b = do
  let initialDepthSequential = 2
      allLeaves    = getNodesFromRoot b initialDepthSequential
      leafResults  = withStrategy (parList rdeepseq) (map solveFromNode allLeaves)
      solutions    = [ s | Just s <- leafResults ] --get rid of the ones that failed

  case solutions of
    []   -> putStrLn No solution found from any leaf.
    sols -> do
      let best = minimumBy (comparing length) sols
      putStrLn $ Found solution in  ++ show (length best) ++  moves:
      print best
```

The parallel IDA* uses similar methods as the parallel IDDFS. We use a divide and conquer approach that allows us to compute several different search trees in parallel. To divide up the work, we first sequentially compute branches up to a depth of 2. From here, we take each of these leaf nodes and run them in parallel on different threads.

Although it seems like this approach would be able to improve runtime as it's doing computation in parallel, due to the nature of IDA*, a lot of the parallel computation is actually spent searching branches that are "worse", as IDA* uses a heuristic to pick the best branch to search next. In this parallel IDA* approach, each subtree runs its own IDA* independently, but does not share any global cost bound. This means each subtree's choice of next "best" might be worse than the global best to begin with. It therefore can't cut out early when another subtree finds a better solution, which reduces the benefits of the parallel approach, unless there is a way to share the global best across threads.

After running some benchmarks, we were able to see that this parallel approach did not offer much benefit. Specifically, the overhead from starting up threads and running in parallel may even cause this method to be slower than its sequential counterpart. This is likely because the parallel IDA* is doing more redundant work than the sequential IDA*, then adding the overhead for creating sparks and running threads in parallel, we get a longer runtime than the sequential IDA*.

6 Performance Evaluation

All benchmarks were run on a Macbook Pro with a M1 Max chip. It has 10 cores and 10 hardware threads.

6.1 Initial Evaluation and Comparisons

Number of moves	3	8	16	17	30	36	38	40
IDDFS sequential time (s)	0.014	0.013	49.489	226.705				
IDDFS Pruning sequential time (s)	0.013	0.013	0.013	0.012	0.625	106.548	283.695	539.959
IDA* sequential time (s)	0.011	0.013	0.012	0.012	0.653	107.872	295.911	556.515
IDDFS parallel time (s)	0.027	0.012	14.440	83.176				
IDDFS Pruning parallel time (s)	0.013	0.025	0.012	0.018	0.272	71.934	100.648	265.366
IDA* parallel time (s)	0.011	0.013	0.012	0.012	18.767			

All parallel benchmarks were executed using 6 threads. For parallel algorithms that used a split depth, $d = 3$ was used. Blank entries indicate runs that did not complete before 30 minutes (or 1800 seconds).

Table 1: Runtimes of various algorithms for boards of varying number of moves

To compare the performance of the different sequential and parallel algorithms implemented, we ran benchmarks on each of the algorithms. We tested them on a range of boards, from easy (shallow) solutions, to difficult (deeper) solutions. Easy solutions can be solved in a shorter number of moves, meaning the search space is smaller, and hard solutions are solved with a much higher number of moves, which means an exponentially larger search space.

The most interesting results come from looking at the most difficult board chosen, where the optimal solution is 40 moves. If we assume that each board/state can have 4 possible moves, the search space for this configuration would be $4^{40} = 1.2 \times 10^{24}$. Although this represents a loose upper bound (true branching factor likely to be lower as not every board will have 3 possible moves), this is an extremely large search space, so it makes sense that more naive or brute force algorithms were unable to reach a solution within a reasonable time frame, which we considered as 30 minutes.

Investigating the 4 algorithms that were able to solve the 40 moveset board in under 30 minutes, we see that there were two sequential versions and only one parallel version. The sequential versions both involved cutting down the search space, or optimizing what is being searched first; this allowed the algorithms to find the solutions within a reasonable time frame of under 10 minutes. Both the sequential IDDFS pruning approach and IDA* approach had relatively similar timings. On the parallel side, we had a clear winner, IDDFS pruning, so this is our best algorithm that we will further analyze.

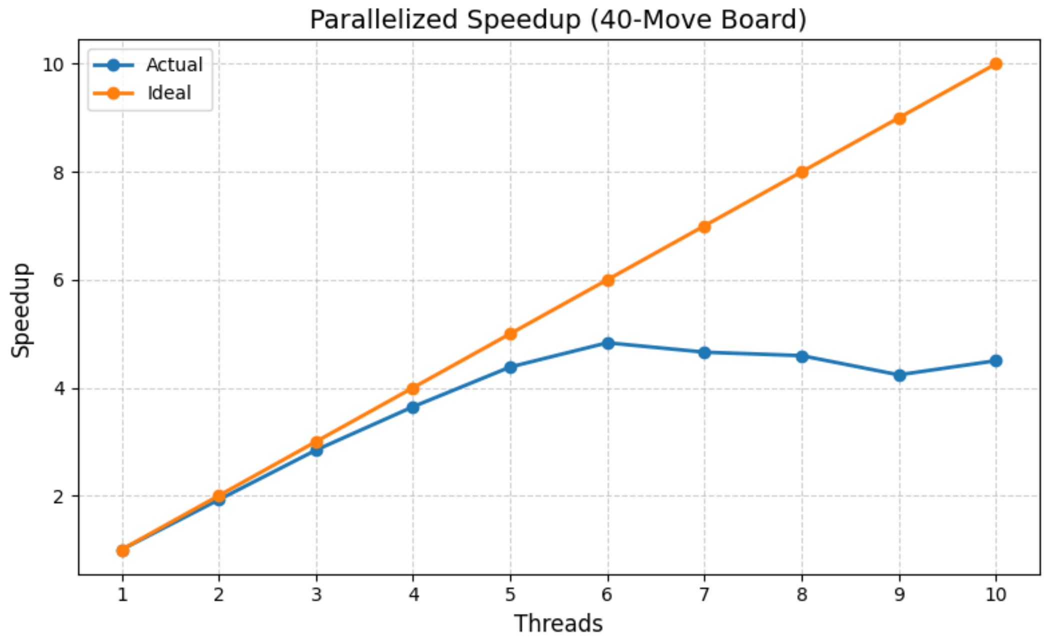


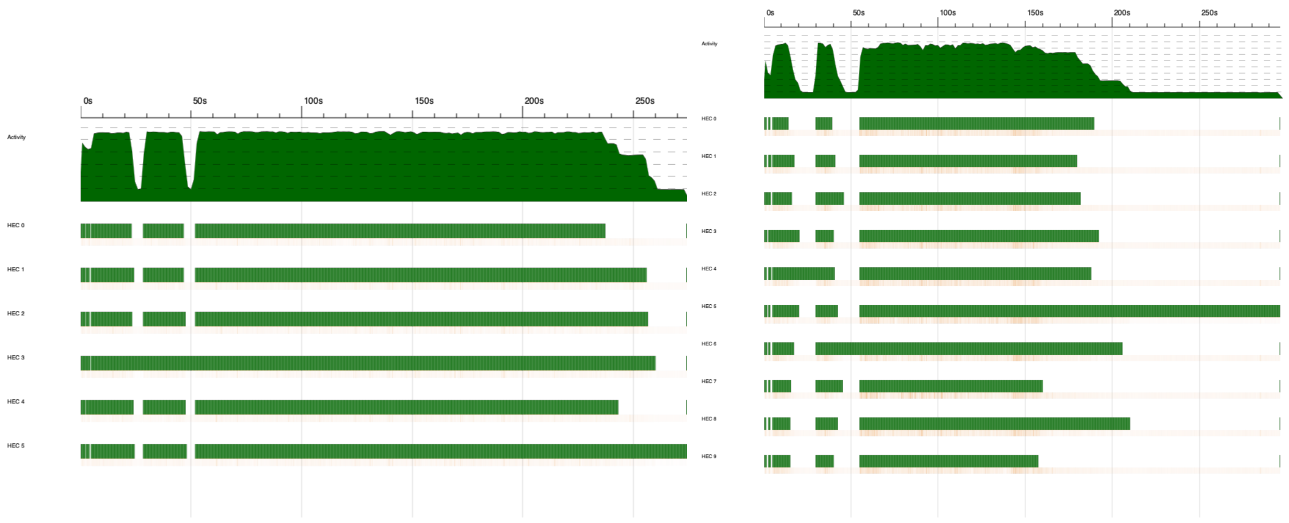
Figure 2: Graph of the parallelized speedup for the IDDFS pruning algorithm[1]

We first graph the speedup with increasing threads to analyze how performant the parallelization is. As we see in Figure 2, although we may not achieve ideal linear scaling, we achieve good speedup up to 6 threads, until which we not only see diminishing returns, but actually start to see slower runtimes too.

It definitely seems like this implementation can be improved upon. In order to determine how this current best approach can be improved, we analyze the threadscope graphs, both for the fastest time (6 threads) and a slower time with more threads (10).

With 6 threads, we can see all the threads are used relatively efficiently. There are no threads that are idle for a long time, except partially at the end, but this is largely to be expected. However, for 10 threads, it does not seem to parallelize as well. Most of the

[1]See Appendix for raw timing data (Table 2)



(a) 6 threads

(b) 10 threads

Figure 3: Threadscope graphs for IDDFS pruning with 6 and 10 threads

threads seem to finish their work at a certain point and then are idle, leaving one thread to finish the work, increasing the overall runtime while not doing much extra work. This downgrade in performance with a higher number of threads could be due to the limited frontier size. Since we precompute an initial frontier to a certain depth (we chose a depth of 3), once we run out of useful trees to explore, the threads that finish early no longer have any work to do. In addition, with pruning, the trees are imbalanced, since many trees get pruned quickly due to the heuristic, so the few remaining trees can be deep and those threads end up doing the majority of the work. We also need to consider the added overhead of creating new threads and balancing work, which leads the versions with more than 6 threads to have a longer runtime than the 6 thread version.

6.2 Further Improvements

Due to the poor efficiency of the 10 thread algorithm (Figure 3), we had initially thought work sharing was a good approach to try to combat this issue of the single thread doing most of the work, but quickly realized that this wasn't the case. Pruning, as its name suggests, already removes many trees/paths and reduces the amount of stealable work to start with. In addition, from the threadscope graph above, by the time we only have one thread active, likely there is only one or two remaining computations and are taking a long time; therefore, work sharing/work stealing would not allow the other threads to do any more work, and instead work sharing likely would just add additional bloating for assigning the threads and parallelization, so we chose to not implement it.

Instead, after considering the reasons work sharing wasn't so effective and the reasons why the 10 thread version was less effective than the 6 thread version, we decided a potential next step could be to tune the depth parameter. It seemed like with the 10 thread version, there weren't enough trees to make parallelization worthwhile, so perhaps increasing the depth (and therefore increasing the frontier size) would be helpful.

Indeed, this did prove to be effective, as seen in Figure 4, as we were able to get the fastest possible time at a depth of 6. The overall runtime was 211 seconds, improved from the previous best, 275 seconds (with a depth of 3 and thread count of 6). $d = 6$ seemed to be the perfect sweet spot, where it isn't too shallow (and so there aren't enough tasks), but also isn't too deep (where the overhead is too high).

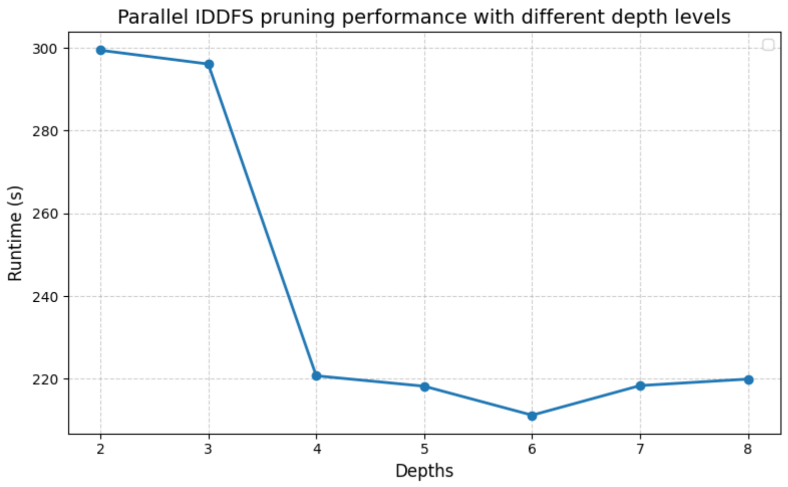


Figure 4: Graph of the runtime for the IDDFS pruning algorithm at varying depths (using 10 threads)[2]

We can now rerun the experiments to see if speedup increases with threads with this new optimal depth, $d = 6$ in Figure 5. This speedup graph is now much closer to the perfect, optimal, linear speedup, although drops off slightly towards the 9th and 10th threads.

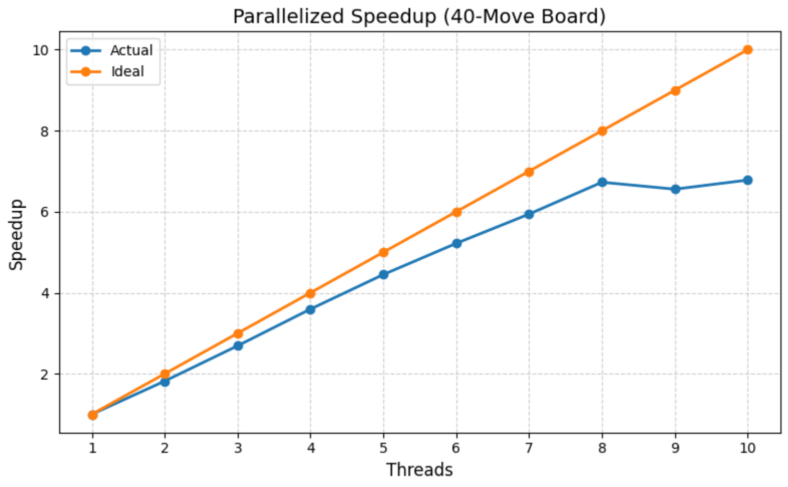


Figure 5: Graph of the runtime for the IDDFS pruning algorithm at $d = 6$[3]

[2]See Appendix for raw timing data (Table 3)

We can also analyze the sparks created and see that out of the 34055 sparks created, 9250 were converted, yielding a rate of ∼27%. This is somewhat expected for a DFS search (especially with pruning), as many sparks are created preemptively before it is known whether a branch will ultimately survive pruning or not. We also had 7923 sparks that overflowed, which again was likely due to the nature of the DFS. This high amount of overflowed sparks indicates that the parallel work was generated faster than it could be scheduled, which suggests that the frontier size at depth 6 successfully exposed enough parallelism, even though not all sparks could be executed. Therefore, it may be difficult to perfectly tune and reduce the amount of overflowed sparks while still maintaining full usage of the 10 threads.

```
SPARKS: 34055 (9250 converted, 7923 overflowed, 0 dud, 492 GC'd, 8198 fizzled)

INIT    time    0.007s  (  0.007s elapsed)
MUT     time  1661.490s  (204.365s elapsed)
GC      time   10.706s  (  7.665s elapsed)
EXIT    time    0.002s  (  0.002s elapsed)
Total   time  1672.205s  (212.039s elapsed)

Alloc rate    1,281,173,356 bytes per MUT second

Productivity  99.4% of total user, 96.4% of total elapsed
```

Finally, we can analyze the updated threadscope graph and we can see that all 10 threads are being used simultaneously, which is ideal, and is consistent with the fact that the runtime is the fastest.

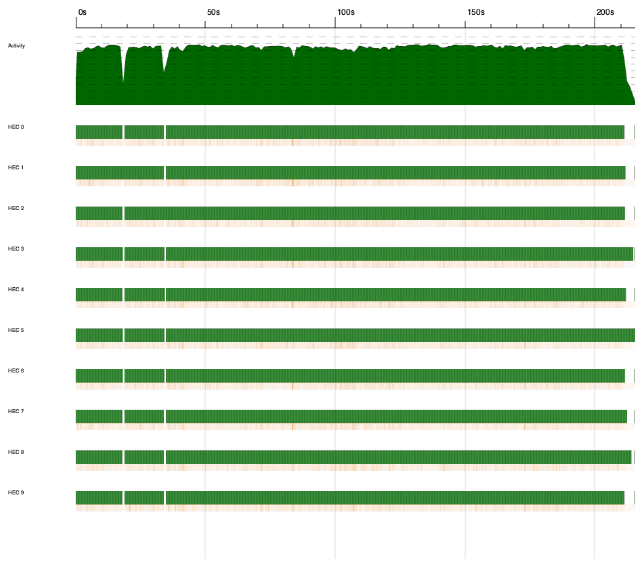


Figure 6: Graph of the parallelized speedup for the IDDFS pruning algorithm

From this graph, clearly this is an improvement on the previous iteration with depth = 3 as all threads are now being used efficiently; we don't have many threads waiting idly,

[3]See Appendix for raw timing data (Table 4)

which is an improvement, resulting in our best parallel algorithm.

7 Discussion and Conclusion

We explored various algorithmic solutions to solving the 15 tile sliding tile game, evaluating both sequential and parallel solutions. We were able to increase runtime on the sequential level by adding heuristics and pruning, and further improve on this runtime by adding in various parallelization approaches, including work sharing and divide and conquer.

Ultimately, our best results came from parallelizing the IDDFS pruning approach, and fine tuning the depth of sequential exploration, as this allowed us to make full use of all 10 threads.

Overall, we can see that parallelization is valuable in increasing the runtime especially when the search space gets very large, however, the overhead in creating threads and running in parallel can lead to poor performance for smaller search spaces.

To achieve the best results, parameters need to be tested and fine tuned in order to make the best use of the threads in parallel to optimize runtime.

For further exploration, it would be interesting to see if there are any other ways to improve the runtime even further, perhaps with global shared bounds across threads (would benefit parallel IDA*) or dynamically adjusting the depth value, which we manually had to test and fine tune. However, even without these, our approaches were able to achieve a significant speed increase from the first naive IDDFS version.

References

[1] Poskitt, K. (2015). The Sam Loyd sliding block puzzle. The Sam Loyd Sliding Block Puzzle. http://www.murderousmaths.co.uk/games/loyd/loydfr.htm

[2] Sweigart, A. (2021). Sliding-Tile Solver. Invent with Python. https://inventwithpython.com/recursion/chapter12.html

A Appendix

A.1 Raw data tables

Threads	1	2	3	4	5	6	7	8	9	10
Parallel Time (s)	1332.297	692.537	468.643	365.319	303.953	275.691	286.020	290.054	314.366	296.107

Table 2: Speedup for IDDFS Pruning on 40-Move Board with $d = 3$

Depth	2	3	4	5	6	7	8
Parallel Time (s)	299.429	296.107	220.759	218.239	211.234	218.392	219.962

Table 3: Speedup for IDDFS Pruning on 40-Move Board with varying depth

Threads	1	2	3	4	5	6	7	8	9	10
Parallel Time (s)	1432.492	787.443	532.376	398.286	321.960	274.484	241.081	212.914	218.484	211.234

Table 4: Speedup for IDDFS Pruning on 40-Move Board with $d = 6$

A.2 Code

```haskell
{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}

module Puzzle
  ( Board, Move(..)
  , boardSize, numTiles
  , startBoardUnsolvable
  , startBoard3, startBoard8, startBoard16, startBoard17, startBoard30, startBoard36
    , startBoard38, startBoard40
  , startBoardByMoves
  , getRandomBoard, getRandomSolvableBoard
  , isSolvableBoard, isSolved
  , printBoard
  , getManhattanDistanceOfBoard
  , validMoves, applyMove
  , iddfs, iddfsWithDepthLimit
  , iddfspruning, iddfsWithPruning
  , ida, idaWithDepthLimit
  , parallelIDA, getNodesFromRoot
  , parallelIddfs, parallelIddfsPruning,
  ) where

import Data.Maybe (fromJust)
import System.Random.Shuffle (shuffleM)
import Data.List (intercalate, elemIndex, minimumBy)
import Data.Ord  (comparing)
import Control.Parallel.Strategies (withStrategy, parList, rdeepseq)
import Control.DeepSeq (NFData)
import GHC.Generics (Generic)

-- Simple board type:
-- 0 = empty tile
type Board = [Int]

-- node of the search tree, to be used when running in parallel
type Node = (Board, [Move])

data Move = MoveLeft | MoveRight | MoveUp | MoveDown
  deriving (Show, Eq, Generic, NFData)

-- 4x4 board (15-puzzle)
boardSize :: Int
boardSize = 4

numTiles :: Int
numTiles = boardSize * boardSize

-----------------
-- BOARD SETUP --
-----------------

-- Sample board, unsolvable
startBoardUnsolvable :: Board
startBoardUnsolvable   =
  [ 1,  2,  3,  4
  , 5,  6,  7,  8
  , 9, 10, 11, 12
  ,13, 15, 14,  0
  ]

startBoard3 :: Board
startBoard3 =
  [ 1,  2,  3,  4
  , 5,  6,  7,  8
  , 9, 10, 11, 12
  , 0, 13, 14, 15
  ]

startBoard8 :: Board
startBoard8 =
  [ 1,  2,  3,  4
  , 5,  6, 12,  7
  , 9, 10,  8, 15
  ,13, 14, 11,  0
  ]

startBoard16 :: Board
startBoard16 =
  [ 2,  3,  4,  8
  , 1,  6,  7, 12
  , 0, 10, 11, 15
  , 5,  9, 13, 14
  ]

startBoard17 :: Board
startBoard17 =
  [ 2,  3,  4,  8
  , 1,  6,  7, 12
  ,10,  0, 11, 15
  , 5,  9, 13, 14
  ]

startBoard30 :: Board
startBoard30 =
  [ 2,  7,  3,  4
  , 10,  1,  12, 8
  , 6, 11, 15, 14
```

```haskell
  , 5, 9, 13, 0
  ]

startBoard36 :: Board
startBoard36 =
  [ 2,  7,  0,  3
  ,10,  1,  8,  4
  , 6, 11, 12, 14
  , 5,  9, 15, 13
  ]

startBoard38 :: Board
startBoard38 =
  [ 10,  2,  7,  3
  , 6,  1,  8, 0
  , 11,12, 14, 4
  , 5, 9, 15, 13
  ]

startBoard40 :: Board
startBoard40 =
  [10,  2,  0,  7
  , 6,  1,  8,  3
  ,11, 12, 14,  4
  , 5,  9, 15, 13
  ]

startBoardByMoves :: String -> Board
startBoardByMoves 3  = startBoard3
startBoardByMoves 8  = startBoard8
startBoardByMoves 16 = startBoard16
startBoardByMoves 17 = startBoard17
startBoardByMoves 30 = startBoard30
startBoardByMoves 36 = startBoard36
startBoardByMoves 38 = startBoard38
startBoardByMoves 40 = startBoard40
startBoardByMoves _    = startBoard3 -- this is the default one

--------------------------------------------------------------------------------
-- HELPERS TO GENERATE RANDOM BOARD AND CHECK WHETHER IT CAN BE SOLVED (was actually
     not used in the project in the end) --
--------------------------------------------------------------------------------

getRandomBoard :: IO Board
getRandomBoard = shuffleM [0..15]

getRandomSolvableBoard :: IO Board
getRandomSolvableBoard = do
  b <- getRandomBoard
  if isSolvableBoard b
    then return b
    else getRandomSolvableBoard
```

```haskell

isSolvableBoard :: Board -> Bool
isSolvableBoard b = (invParity + blankParity) `mod` 2 == 1 -- either inv or blank parity is odd, but not both (this is due to ignoring the empty space, if we don't ignore it, then we want both to be even)
  where
    invCount = length [ () | i <- [0..length b - 1], j <- [i+1..length b - 1],
                            b !! i /= 0, b !! j /= 0, b !! i > b !! j ]
    invParity = invCount `mod` 2
    blankRowFromBottom = boardSize - (indexOf 0 b `div` boardSize)
    blankParity = blankRowFromBottom `mod` 2

-- Check if board is in solved configuration
isSolved :: Board -> Bool
isSolved b = b == [1..(boardSize * boardSize - 1)] ++ [0]

-- Print the board
printBoard :: Board -> IO ()
printBoard b = do
  let rows = chunk boardSize b
  putStrLn ---------
  mapM_ printRow rows
  putStrLn ---------
  where
    printRow row =
      putStrLn . intercalate   $
        map showTile row

    showTile 0 =  .   -- use . for empty
    showTile n = pad2 n

    pad2 n =
      if n < 10 then ' ':show n else show n

-----------------------------------
-- HELPERS FOR MOVES AND SOLVING --
-----------------------------------

-- Split list into chunks of size n
chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk n xs =
  let (h,t) = splitAt n xs
  in h : chunk n t

-- Helper: find index of a value in the board
indexOf :: Eq a => a -> [a] -> Int
indexOf x xs = fromJust (elemIndex x xs)

getManhattanDistanceOfBoard :: Board -> Int
getManhattanDistanceOfBoard b = distanceHelper (numTiles - 1) 0
  where
```

```haskell
    distanceHelper :: Int -> Int -> Int
    distanceHelper index total
      | index < 0      = total
      | currTile == 0  = distanceHelper (index - 1) total
      | otherwise      = distanceHelper (index - 1) (total + newDist)
      where
        currTile  = b !! index
        row       = index `div` boardSize
        col       = index `mod` boardSize
        goalIndex = currTile - 1
        goalRow   = goalIndex `div` boardSize
        goalCol   = goalIndex `mod` boardSize
        newDist   = abs (row - goalRow) + abs (col - goalCol)


validMoves :: Board -> [Move]
validMoves b =
  let emptyIdx = indexOf 0 b
      n        = boardSize
      er       = emptyIdx `div` n
      ec       = emptyIdx `mod` n
  in  [ m
      | (cond, m) <-
          [ (ec < n - 1, MoveLeft)   -- tile to the right moves left
          , (ec > 0    , MoveRight)  -- tile to the left move right
          , (er < n - 1, MoveUp)     -- tile below can moves up
          , (er > 0    , MoveDown)   -- tile above can moves down
          ]
      , cond
      ]


-- Simple move: slide tile at index i into the empty slot
slideIntoEmpty :: Board -> Int -> Board
slideIntoEmpty b tileIndex =
  let emptyIndex = indexOf 0 b
      v1 = b !! tileIndex
      v2 = b !! emptyIndex
      replace i v xs = take i xs ++ [v] ++ drop (i+1) xs
      b' = replace tileIndex v2 b
  in  replace emptyIndex v1 b'

-- assume that all inputs are valid based off validMoves function
applyMove :: Board -> Move -> Board
applyMove b move = case move of
  MoveLeft  -> slideIntoEmpty b (tile + 1)
  MoveRight -> slideIntoEmpty b (tile - 1)
  MoveUp    -> slideIntoEmpty b (tile + boardSize)
  MoveDown  ->  slideIntoEmpty b (tile - boardSize)
  where
    tile = indexOf 0 b

```

```haskell
-----------------------
-- SOLVING FUNCTIONS --
-----------------------

iddfs :: Board -> Maybe [Move]
iddfs b = recursivelyIddfsWithDepthLimit 0
  where
    recursivelyIddfsWithDepthLimit depth =
      case iddfsWithDepthLimit b depth of
        Just moves -> Just moves
        Nothing    -> recursivelyIddfsWithDepthLimit (depth + 1)

iddfsWithDepthLimit :: Board -> Int -> Maybe [Move]
iddfsWithDepthLimit b depth = dfs b depth []
  where
    dfs :: Board -> Int -> [Move] -> Maybe [Move]
    dfs board d path
      | isSolved board = Just (reverse path)
      | d == 0 = Nothing
      | otherwise  = tryMoves (validMoves board) board d path

    tryMoves :: [Move] -> Board -> Int -> [Move] -> Maybe [Move]
    tryMoves [] _ _ _ = Nothing
    tryMoves (m:ms) board d path =
      case dfs (applyMove board m) (d - 1) (m:path) of
        Just sol -> Just sol
        Nothing  -> tryMoves ms board d path

iddfsPruningWithDepthLimit :: Board -> Int -> Maybe [Move]
iddfsPruningWithDepthLimit b depthLimit = dfs b 0 []
  where
    dfs :: Board -> Int -> [Move] -> Maybe [Move]
    dfs board depth path
      | isSolved board          = Just (reverse path)
      | depth == depthLimit     = Nothing
      | depth + h > depthLimit  = Nothing  -- prune: even best-case path too long
      | otherwise               = tryMoves (validMoves board) board depth path
      where
        h = getManhattanDistanceOfBoard board

    tryMoves :: [Move] -> Board -> Int -> [Move] -> Maybe [Move]
    tryMoves [] _ _ _ = Nothing
    tryMoves (m:ms) board depth path =
      case dfs (applyMove board m) (depth + 1) (m : path) of
        Just sol -> Just sol
        Nothing  -> tryMoves ms board depth path

iddfspruning :: Board -> Maybe [Move]
iddfspruning b = iter 0
  where
    iter depthLimit =
      case iddfsPruningWithDepthLimit b depthLimit of
```

```
        Just moves -> Just moves
        Nothing    -> iter (depthLimit + 1)

iddfsWithPruning :: Board -> Maybe [Move]
iddfsWithPruning = iddfspruning

ida :: Board -> Maybe [Move]
ida b = iter (getManhattanDistanceOfBoard b)
  where
    -- depth includes the heuristic (g + h)
    iter depth =
      case idaWithDepthLimit b depth of
        Just moves -> Just moves
        Nothing    -> iter (depth + 1)


idaWithDepthLimit :: Board -> Int -> Maybe [Move]
idaWithDepthLimit b depth = dfs b 0 []
  where
    dfs :: Board -> Int -> [Move] -> Maybe [Move]
    dfs board d path
      | f > depth  = Nothing
      | isSolved board = Just (reverse path)
      | otherwise  = tryMoves (validMoves board) board d path
      where
        h = getManhattanDistanceOfBoard board
        f = d + h

    tryMoves :: [Move] -> Board -> Int -> [Move] -> Maybe [Move]
    tryMoves [] _ _ _ = Nothing
    tryMoves (m:ms) board d path =
      case dfs (applyMove board m) (d + 1) (m : path) of
        Just sol -> Just sol
        Nothing  -> tryMoves ms board d path

------------------------
-- PARALLEL FUNCTIONS --
------------------------

getNodesFromRoot :: Board -> Int -> [Node]
getNodesFromRoot start d0 =
  go [(start, [])] d0
  where
    go :: [Node] -> Int -> [Node]
    go frontier 0      = frontier
    go frontier depthN = go nextFrontier (depthN - 1)
      where
        nextFrontier :: [Node]
        nextFrontier = concatMap expandNode frontier


expandNode :: Node -> [Node]
```

```haskell
expandNode (b, path) = [(applyMove b m, path ++ [m]) | m <- validMoves b ]

solveFromNode :: Node -> Maybe [Move]
solveFromNode (b, currentMoves) =
  case ida b of
    Nothing       -> Nothing
    Just resultingMoves   -> Just (currentMoves ++ resultingMoves)

parallelIDA :: Board -> IO ()
parallelIDA b = do
  let initialDepthSequential = 2
      allLeaves    = getNodesFromRoot b initialDepthSequential
      leafResults  = withStrategy (parList rdeepseq) (map solveFromNode allLeaves)
      solutions    = [ s | Just s <- leafResults ] --get rid of the ones that failed

  case solutions of
    []   -> putStrLn No solution found from any leaf.
    sols -> do
      let best = minimumBy (comparing length) sols
      putStrLn $ Found solution in  ++ show (length best) ++  moves:
      print best

-- parallelized over subtrees at a shallow split depth.
parallelIddfs :: Board -> IO ()
parallelIddfs startBoard = iter 0
  where
    -- how deep we go sequentially before splitting work
    splitDepth :: Int
    splitDepth = 3

    -- precompute the frontier at depth = splitDepth
    frontier :: [Node]
    frontier = getNodesFromRoot startBoard splitDepth
    -- each Node = (boardAtDepthK, movesFromRootToHere)

    iter :: Int -> IO ()
    iter depthLimit
      | depthLimit < splitDepth =  -- too shallow to bother splitting
          case iddfsWithDepthLimit startBoard depthLimit of
            Nothing      -> iter (depthLimit + 1)
            Just solPath -> do
              putStrLn $ Found solution in  ++ show (length solPath)
                       ++  moves (depth limit  ++ show depthLimit ++ ):
              print solPath

      | otherwise = do
          let remaining = depthLimit - splitDepth

          -- for this depth limit, search each subtree in parallel
          let leafResults :: [Maybe [Move]]
              leafResults =
                withStrategy (parList rdeepseq)
```

```
                  [ fmap (prefixMoves ++)
                      (iddfsWithDepthLimit board remaining)
                  | (board, prefixMoves) <- frontier
                  ]

              solutions = [ s | Just s <- leafResults ]

          case solutions of
            []    -> iter (depthLimit + 1)
            sols  -> do
              let best = minimumBy (comparing length) sols
              putStrLn $ Found solution in  ++ show (length best)
                       ++  moves (depth limit  ++ show depthLimit ++ ):
              print best

-- Global iterative deepening over depthLimit,
-- parallelized over subtrees at a shallow split depth
parallelIddfsPruning :: Board -> IO ()
parallelIddfsPruning startBoard = iter 0
  where
    -- how deep to go sequentially before we split work
    splitDepth :: Int
    splitDepth = 6

    -- frontier at depth = splitDepth (boards and prefix move sequences)
    frontier :: [Node]
    frontier = getNodesFromRoot startBoard splitDepth

    iter :: Int -> IO ()
    iter depthLimit
      -- For very shallow limits, just do everything from the root sequentially
      | depthLimit < splitDepth =
          case iddfsPruningWithDepthLimit startBoard depthLimit of
            Nothing      -> iter (depthLimit + 1)
            Just solPath -> do
              putStrLn $ Found solution in
                       ++ show (length solPath)
                       ++  moves (depth limit
                       ++ show depthLimit
                       ++ , pruned, sequential):
              print solPath

      | otherwise = do
          let remaining = depthLimit - splitDepth

          let leafResults :: [Maybe [Move]]
              leafResults =
                withStrategy (parList rdeepseq)
                  [ fmap (prefixMoves ++)
                      (iddfsPruningWithDepthLimit board remaining)
                  | (board, prefixMoves) <- frontier
                  ]

              solutions = [ s | Just s <- leafResults ]

          case solutions of
            []    -> iter (depthLimit + 1)
            sols  -> do
              let best = minimumBy (comparing length) sols
              putStrLn $ Found solution in
                       ++ show (length best)
                       ++  moves (depth limit
                       ++ show depthLimit
                       ++ , pruned, parallel):
              print best
```

A.3 GitHub Link

For further information or to run the code, please visit https://github.com/eden-chung/Sliding-Tiles-Parallel