N-Parzzle Solver

Ahmad Rawwagah (afr2151), Kimberly Tsao (kt2803),
Ruth Lee (rsl2159)

December 20, 2023

1 Introduction

The N-parzzle game is a rendition of the more popularly recognized, 8-Puzzle. This game starts with an initial board state with N movable tiles and one empty space. There is one tile for each number in the set 1, 2, 3, ... N. The aim of this game is to get from any initial board state (as shown in Figure 1) to the configuration with all tiles in ascending order as shown in Figure 2. The example below shows a 3x3 board for an 8-Puzzle, but our project also runs on 4x4 boards (15-Puzzle), which has a much longer run time because the search space increases exponentially.

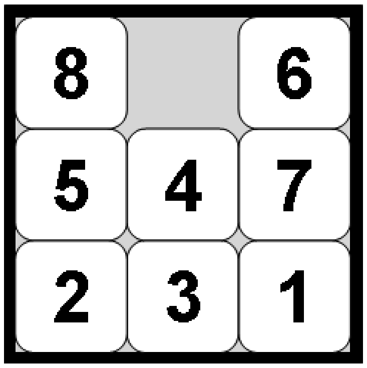


Figure 1: Initial State

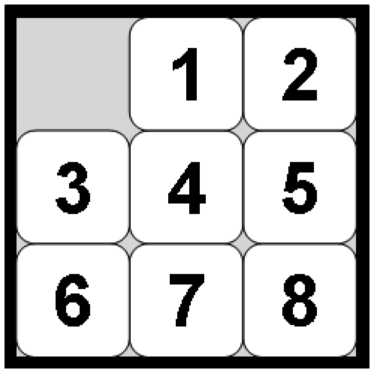


Figure 2: Goal State

2 Implementation

The N-puzzle can be solved using different search algorithms, such as Breadth-First Search (BFS), Depth-First Search (DFS), and A* search algorithms. To later create a version modified to run in parallel, we first implemented a sequential solver in Python using A* search.

2.1 A* Search with Modification

The A* algorithm is a popular search algorithm because of its use of heuristics to find the shortest path to a goal state. At each generated next neighbor state, it considers the number of moves it took to reach the current state from the start state and computes an estimation (using a specified heuristic) to determine the number of moves (the distance) to the goal state.

$F(x) = G(x) + H(x)$
$F(x) =$Total Cost to Current Board
$G(x) = \#$ Moves from Start to Current Board
$H(x) =$Manhattan distance from Current to Goal Board

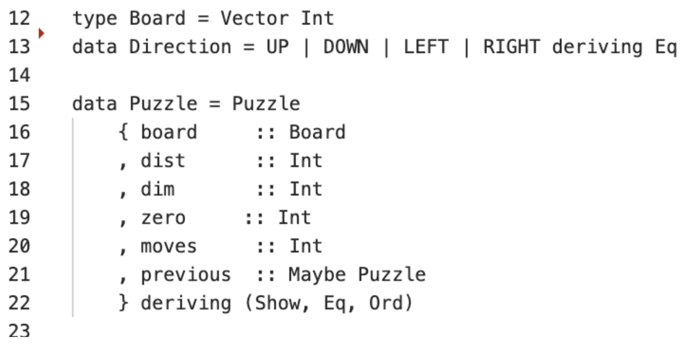
For each tile of a current board, we calculate the Manhattan distance to goal position and then sum the Manhattan distance for every tile. The Manhattan distance is the sum of all vertical and horizontal moves until a tile reaches their goal position.

Although this ensures that we would reach the shortest path, our goal for this sequential algorithm is to quickly find any solution. Therefore, our implementation of the A* algorithm was modified to only consider the estimated number of moves to the goal state ($H(x)$) and prioritize those closer to goal. This means that we first explore boards that are estimated to reach the goal in the least number of moves. This was done using a priority queue.

2.2 Data Structures

We created a Puzzle object so that we could easily store and access information regarding each state of a puzzle such as the vector representation of the board, distance heuristic, number of moves, position of the empty space, and previous puzzle state.

We also created a Direction, which consists of UP, DOWN, LEFT, and RIGHT so that we could apply the move to the board.



```haskell
type Board = Vector Int
data Direction = UP | DOWN | LEFT | RIGHT deriving Eq

data Puzzle = Puzzle
    { board     :: Board
    , dist      :: Int
    , dim       :: Int
    , zero     :: Int
    , moves     :: Int
    , previous  :: Maybe Puzzle
    } deriving (Show, Eq, Ord)

```

Figure 3: Data structure implementation of Direction and Puzzle

2.3 Parallel Implementation to Generate Neighbor States

Our approach to implementing a parallel generation of the next board states is shown in the code below. Rather than using mapMaybe, we took a parallel approach that works with maybe types. We created a parMapMaybe function rather than just parMap so that we could work with cases where applying the move function on a direction like UP to the board would return nothing. This would be the case if the empty space was on the top row, making the empty space unable to perform UP.

We used 'parList rseq' to perform the move function in parallel where rseq helps prevent Haskell's lazy evaluation by making sure each direction has started being evaluated to head normal form before a list is returned.

```
parMapMaybe :: (a -> Maybe b) -> [a] -> [b]
parMapMaybe f xs = (mapMaybe f xs) `using` (parList rseq)

neighbors :: Puzzle -> [Puzzle]
neighbors p = parMapMaybe (move p) [UP, DOWN, LEFT, RIGHT]
```

Figure 4: Parallel implementation to generate neighbor states.

2.4 Parallel Implementation to Calculate Manhattan Distances

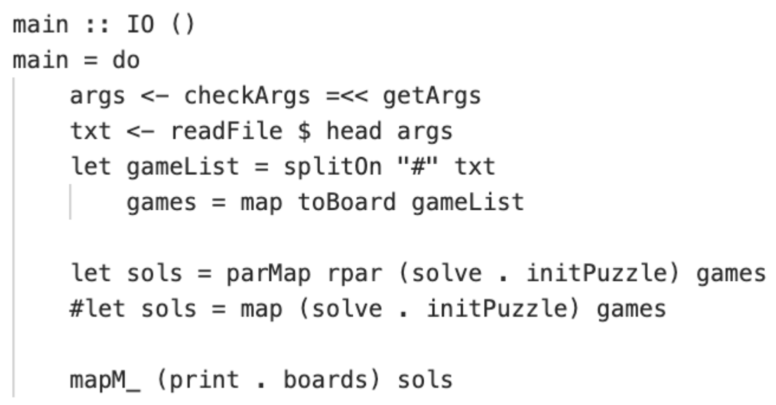
Another strategy we attempted to make our A* search parallel was to make our total heuristic calculations more efficient since the function runs with $O(N^2)$ time complexity by calculating Manhattan distance for every tile in the board. As the dimension of the board size increases, the search space increases exponentially. Our heuristic that is calculated for every Puzzle generated in the sum of Manhattan distances for everytile. Although a simple calculation, when the number of tiles increases exponentially, we could potentially see a benefit to adding parallelization. In Figure 5, we use 'parMap rpar' to map the 'manhattan' function, which calculates the distance of a tile's current position to its goal position.

```
-- Applies heuristic to all tiles
totalDist :: Board -> Int -> Int
totalDist b n = sum $ parMap rpar (\i -> sum [manhattan (b ! matrix2array n i j) n i j | j <- [0..n-1]]) [0..n-1]
```

Figure 5: Parallel implementation to calculate heuristic of each Puzzle state.

2.5 Parallel Implementation to Solve Multiple Boards

We then modified our algorithm to run multiple boards at once in parallel. Specifically we created files with 10,000 3x3 boards and a file with 50 4x4 boards. As shown in Figure 6, our implementation was relatively simple by using functions 'parMap rpar' where rpar sparks for evaluation of the solve function and parMap maps to be evaluated all of the games in parallel.



```
main :: IO ()
main = do
    args <- checkArgs =<< getArgs
    txt <- readFile $ head args
    let gameList = splitOn "#" txt
        games = map toBoard gameList

    let sols = parMap rpar (solve . initPuzzle) games
    #let sols = map (solve . initPuzzle) games

    mapM_ (print . boards) sols
```

Figure 6: Parallel implementation to run multiple boards.

2.6 Parallel Implementation to Explore Multiple Priority Queue States

During our presentation, a new technique to parallelize the A* star algorithm was discussed. Even if a state is at the top of the priority queue, it isn't necessarily the best path so far. Instead of exploring one new frontier state at a time, we could explore the *k*-best frontier states at once by using concurrent threads.

Haskell provides support for concurrency through the Control.Concurrent library. This concurrency is lightweight and should therefore have low overhead. Information can be shared between threads via MVars, a kind of synchronised mutable variable. In this way, an MVar that tracks complete searches can be maintained. When a thread reaches a solution, it adds it to the shared data structure. The main thread has been blocked on readMVar so once this solution is found, it'll kill all other threads and return the solution.

Algorithm 1 Parallel Priority Queues

```
if $PQ.sizepsq < k$ then
    run A* algorithm on psq
else
    $complete \leftarrow newEmptyMVar$
    Fork for each queue in psq and run A* algorithm
    tryPutMVar complete solution
    $ret \leftarrow readMVar$
    Kill threads
    Return ret
end if
```

3 Results + Evaluation

Table 1: Runtime comparison of different Parallel Techniques vs. Sequential

Threads	Sequential	ParNeighbors	ParBoard	ParPQ
1	35.59	35.59	35.59	35.59
2	35.59	49.24	31.35	85.31
6	35.59	49.92	38.28	6.24
10	35.59	50.45	39.55	6.3
14	35.59	51.72	39.69	6.73

3.1 Parallel Neighbor States

As Shown in Figure 7, The runtime of the solver only increases as the number of threads used increases. This is true for both 4x4 and 3x3 boards. Looking at the ThreadScope output in Figure 8, We can see that there is no real parallelism present as the activity does not increase across more than one thread.

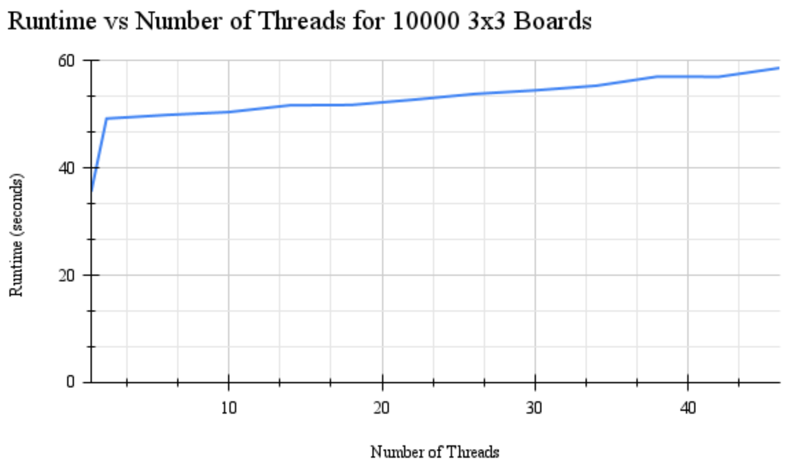
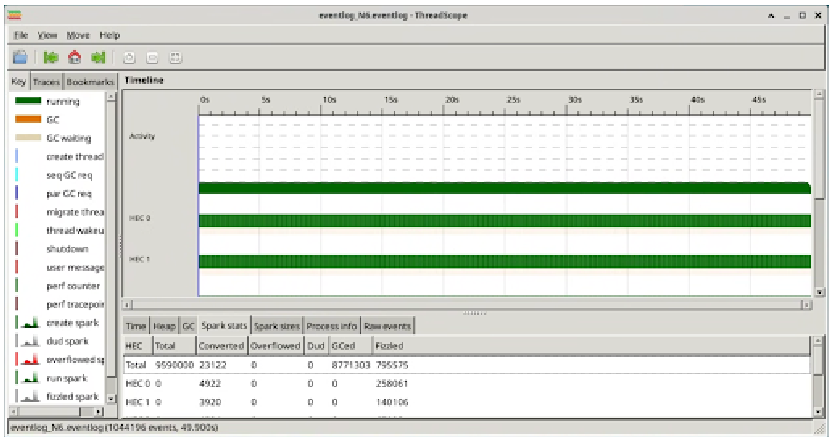


Figure 7: Parallel Neighbors Runtime for 3x3 Boards



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	9590000	23122	0	0	8771303	795575
HEC 0	0	4922	0	0	0	258061
HEC 1	0	3920	0	0	0	140106

eventlog_N6.eventlog (1044196 events, 49.900s)

Figure 8: Parallel Neighbor Threadscope Output for 3x3 Boards using 6 Threads

3.2 Parallel Manhattan Distance

As Shown in Figure 9, The runtime of the solver only increases as the number of threads used increases. This is true for both 4x4 and 3x3 boards. Looking at the ThreadScope output in Figure 10, We can see that there is no real parallelism present as the activity does not increase across more than one thread.

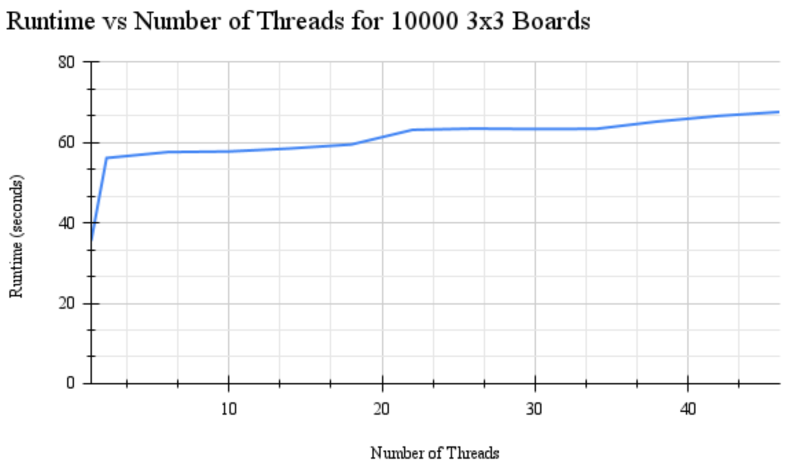
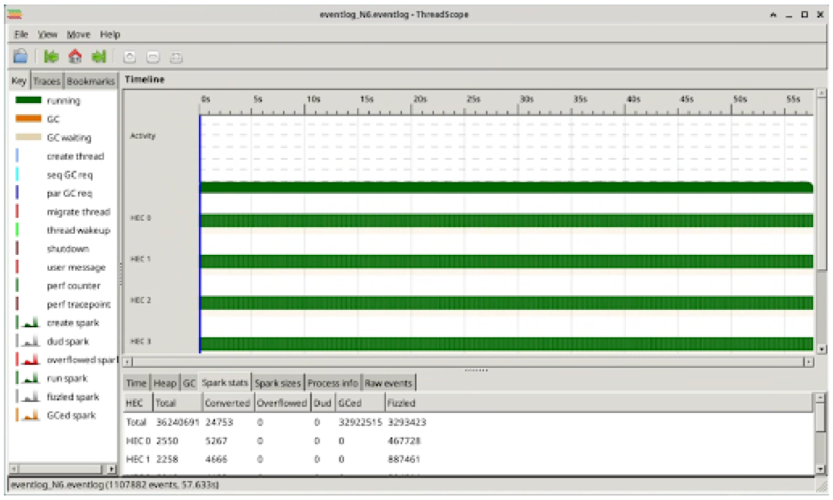


Figure 9: Parallel Distance Runtime for 3x3 Boards



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	36240691	24753	0	0	32922515	3293423
HEC 0	2550	5267	0	0	0	467728
HEC 1	2258	4666	0	0	0	887461

eventlog_N6.eventlog (1107882 events, 57.633s)

Figure 10: Parallel Distance Threadscope Output for 3x3 Boards using 6 Threads

3.3 Parallel Boards

While this approach is expectedly faster than the sequential solution, it’s not necessarily parallelizing the actual A* search process. Of course if you use more threads to simultaneously solve the same number of boards, it’ll complete faster than going through each one at a time.

As seen in Figure 11, the runtime of the solver decreases as more threads are used up to 20 threads, and closely follows the ideal runtime curve. The runtime then increases after more than 20 threads are used.

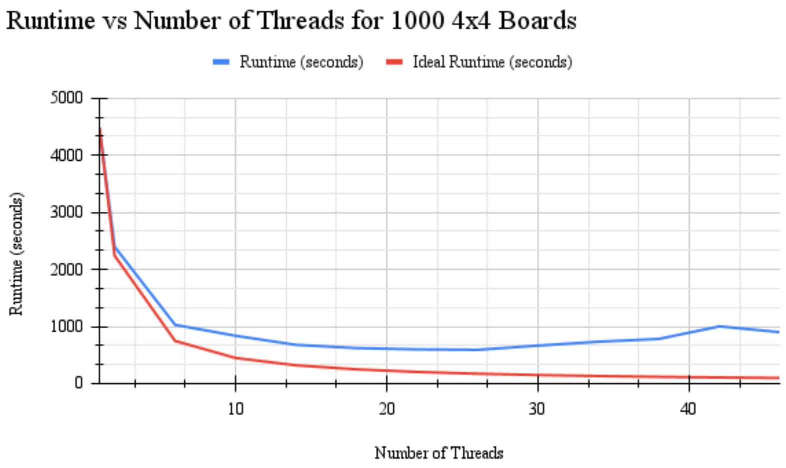
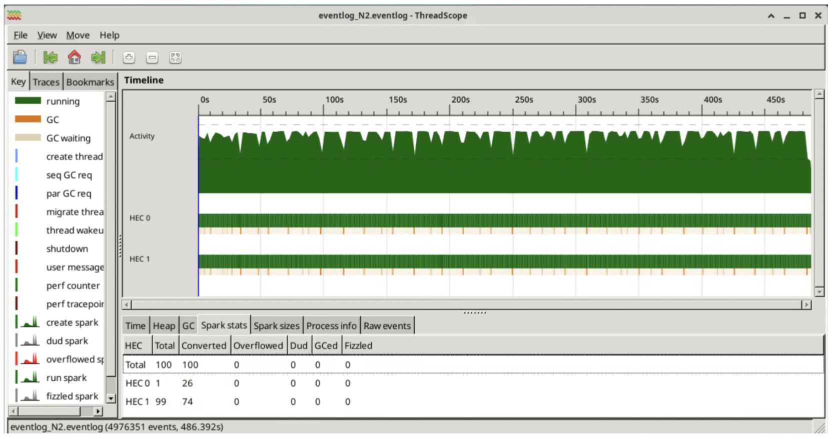


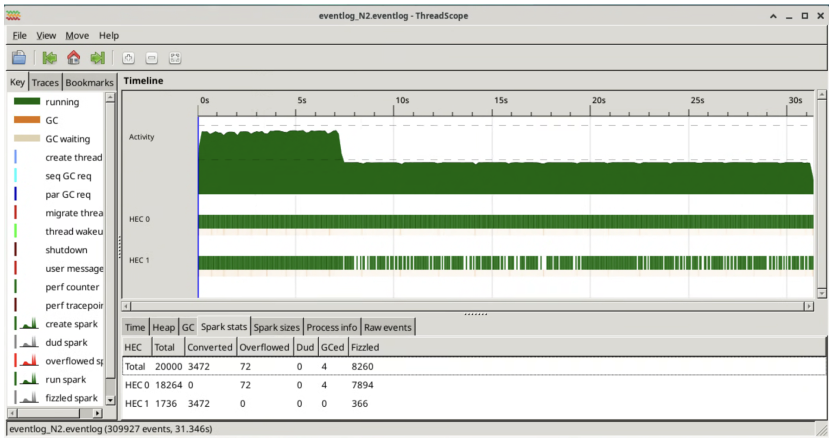
Figure 11: Parallel implementation to run multiple boards.



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	100	100	0	0	0	0
HEC 0	1	26	0	0	0	0
HEC 1	99	74	0	0	0	0

eventlog_N2.eventlog (4976351 events, 486.392s)

Figure 12: Multiple Boards Threadscope for running 50 4x4 boards



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	20000	3472	72	0	4	8260
HEC 0	18264	0	72	0	4	7894
HEC 1	1736	3472	0	0	0	366

eventlog_N2.eventlog (309927 events, 31.346s)

Figure 13: Multiple Boards Threadscope for running 10,000 3x3 boards

3.4 Parallel Multiple Priority Queue States

This technique works well as we can see a significant speedup of 5.83x for $k = 5$ priority queues. Interestingly, in the graph below you can see that the improvement levels off at $N = 5$ and we found that this was related to factors of the k value we set since its determining when and how many times we thread.

For figure 14, we believe that the reason it drops at 5 threads is because we explore 5 priority queue states at once and so it would not need more than 5 threads to run.

Due to the time constraints of this project, we only had time to run on 10 4x4 boards rather than 50.

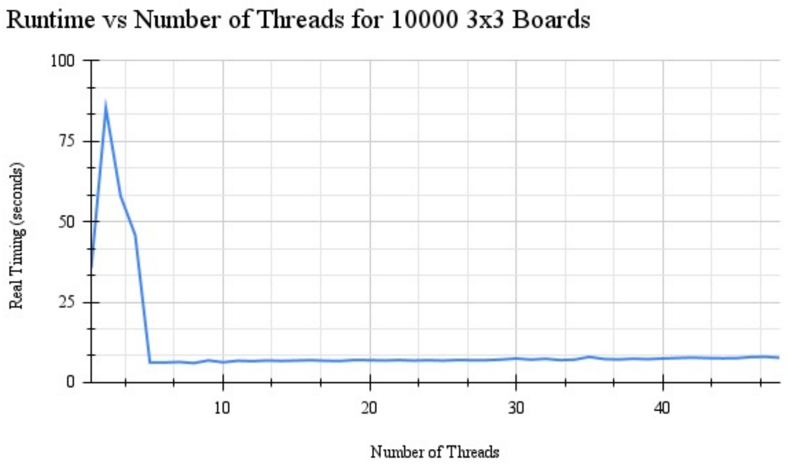


Figure 14: Multiple Priority Queue States Run for 10000 3x3 with N = 5

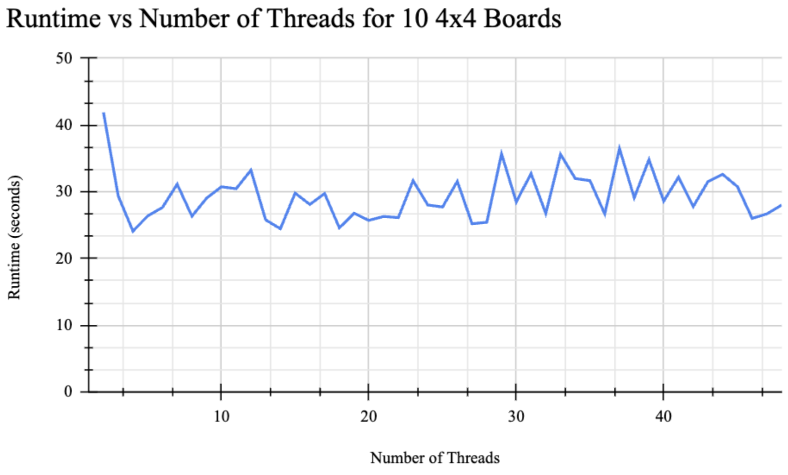
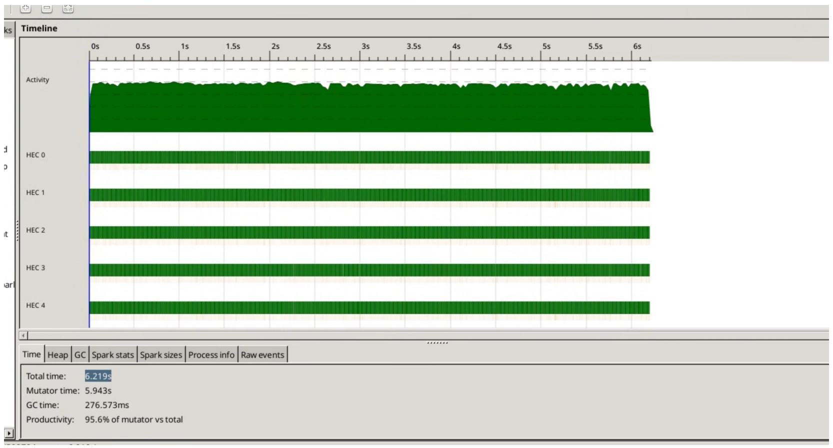
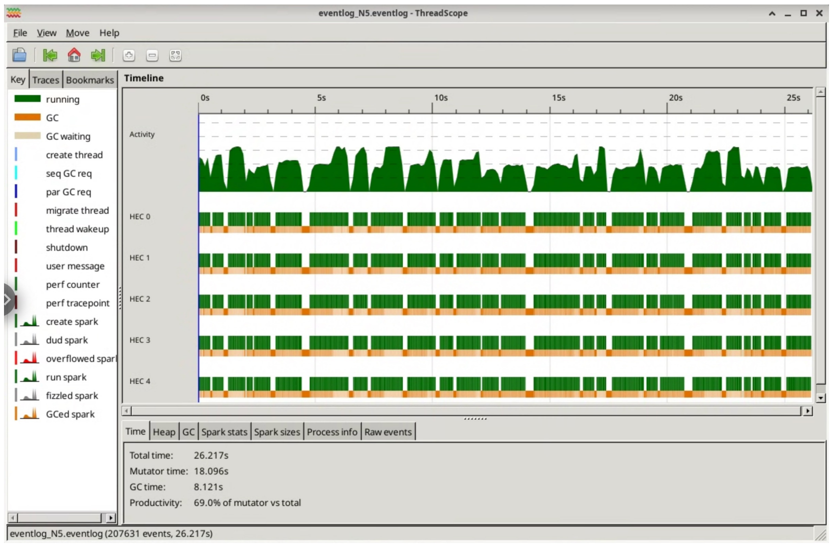


Figure 15: Multiple Priority Queue States Run for 10 4x4 with N = 5



```
Total time:     6.219s
Mutator time:   5.943s
GC time:        276.573ms
Productivity:   95.6% of mutator vs total
```

Figure 16: Multiple Priority Queue States Threadscope for 10000 3x3 with N = 5



```
Total time:     26.217s
Mutator time:   18.096s
GC time:        8.121s
Productivity:   69.0% of mutator vs total
```

```
eventlog_N5.eventlog (207631 events, 26.217s)
```

Figure 17: Multiple Priority Queue States Threadscope for 10 4x4 with N = 5

4 Future Work

The biggest issue with our implementation is likely memory usage that prevented us from testing on larger puzzles such as 24-Puzzle or 35-Puzzle. In the future, we'd like to improve our code by using a hash map to keep track of visited states so we don't repeat searches. However, this may introduce a problem with concurrency because we'd have to ensure we aren't overwriting the hash map from different threads. We'd also like to implement various strategies of pruning.

Listing 1: NParzzle Main.hs

```
module Main (main) where
import Data.Maybe (mapMaybe, fromMaybe)
import Data.Vector (Vector, (!), (//))
import Data.List.Split (splitOn)
import qualified Data.PQueue.Prio.Min as PQ
import qualified Data.Vector as V
import System.Environment (getArgs)
import System.Exit (exitSuccess)
import Control.Monad (forM, void)
import Control.Parallel.Strategies
-- import Control.Parallel (par, pseq)
import Control.DeepSeq (NFData, rnf)
import Control.Concurrent (newEmptyMVar, forkIO, tryPutMVar, readMVar, killThrea

type Board = Vector Int
data Direction = UP | DOWN | LEFT | RIGHT deriving Eq
instance NFData Direction where
    rnf dir = dir `seq` ()

data Puzzle = Puzzle
    { board      :: Board
    , dist       :: Int
    , dim        :: Int
    , zero       :: Int
    , moves      :: Int
    , previous   :: Maybe Puzzle
    } deriving (Show, Eq, Ord)

instance NFData Puzzle where
    rnf puzzle = puzzle `seq` ()

initPuzzle :: [Int] -> Puzzle
initPuzzle xs = Puzzle b d dm z 0 Nothing
    where
        b = V.fromList xs
        d = totalDist b dm
        dm = dimension b
        z = fromMaybe (error "Couldn't find zero tile") (V.elemIndex 0 b)

dimension :: Board -> Int
dimension = round . sqrt . fromIntegral . V.length
```

```
matrix2array :: Int -> Int -> Int -> Int
matrix2array n row col = n * row + col

array2matrix :: Int -> Int -> (Int, Int)
array2matrix n i = (i `div` n, i `mod` n)

manhattan :: Int -> Int -> Int -> Int  -> Int
manhattan v n i j =
    if v == 0
        then 0
        else rowDist + colDist
    where
        rowDist = abs (i - ((v-1) `div` n))
        colDist = abs (j - ((v-1) `mod` n))

totalDist :: Board -> Int -> Int
totalDist b n = sum [manhattan (b ! matrix2array n i j) n i j | i <- [0..n-1], j

swap :: Puzzle -> Int -> Int -> Puzzle
swap p i j = p { board = b
               , dist = totalDist b n
               , zero = k
               , moves = moves p + 1
               , previous = Just p }
    where
        k = matrix2array n i j
        b = prev // [(zero p, prev ! k), (k, 0)]
        prev = board p
        n = dim p

move :: Puzzle -> Direction -> Maybe Puzzle
move p dir = case dir of
    UP -> if i > 0
        then Just $ swap p (i-1) j else Nothing
    DOWN -> if i < n-1
        then Just $ swap p (i+1) j else Nothing
    LEFT -> if j > 0
        then Just $ swap p i (j-1) else Nothing
    RIGHT -> if j < n-1
        then Just $ swap p i (j+1) else Nothing
    where
        (i, j) = array2matrix n (zero p)
        n = dim p

parMapMaybe :: NFData b => (a -> Maybe b) -> [a] -> [b]
parMapMaybe f xs = runEval $ parList rdeepseq (mapMaybe f xs)
```

```
neighbors :: Puzzle -> [Puzzle]
neighbors p = mapMaybe (move p) [UP, DOWN, LEFT, RIGHT]

oldSolve :: Puzzle -> Puzzle
oldSolve p = go (PQ.fromList [(dist p, p)])
    where
        go frontier = if dist puzzle == 0
                then puzzle
                else go newFrontier
            where
                ((_, puzzle), topFrontier) = PQ.deleteFindMin frontier

                prev = case previous puzzle of
                    Nothing -> neighbors puzzle
                    Just n  -> filter (\x -> board x /= board n) (neighbors puzz

                ps  = zip [moves q + dist q | q <- prev] prev
                newFrontier = foldr (uncurry PQ.insert) topFrontier ps


solve :: Puzzle -> IO Puzzle
solve p = do
    let psq = PQ.fromList [(dist p, p)]
    solveParQ psq

solveParQ :: PQ.MinPQueue Int Puzzle -> IO Puzzle
solveParQ psq = do
    let k   = 5
    if PQ.size psq < k then do
        let ((_, puzzle), topFrontier) = PQ.deleteFindMin psq

        let prev = case previous puzzle of
                Nothing -> neighbors puzzle
                Just n  -> filter (\x -> board x /= board n) (neighbors puzzle)

        let ps = zip [moves q + dist q | q <- prev] prev
        let newFrontier = foldr (uncurry PQ.insert) topFrontier ps
        solveParQ newFrontier
    else do
        complete <- newEmptyMVar
        threads <- forM [uncurry PQ.singleton x| x <- PQ.toList psq] $ \pqFork -
            pSolved <- goSolve pqFork
            void (tryPutMVar complete pSolved)
        ret <- readMVar complete
        mapM_ killThread threads
```

```haskell
        return ret

goSolve :: PQ.MinPQueue Int Puzzle -> IO Puzzle
goSolve frontier = if dist puzzle == 0
                   then return puzzle
                   else goSolve newFrontier
  where
    ((_, puzzle), topFrontier) = PQ.deleteFindMin frontier

    prev = case previous puzzle of
      Nothing -> neighbors puzzle
      Just n  -> filter (\x -> board x /= board n) (neighbors puzzle)

    ps  = zip [moves q + dist q | q <- prev] prev
    newFrontier = foldr (uncurry PQ.insert) topFrontier ps


boards :: Puzzle -> [[Int]]
boards p = map V.toList (reverse $ brds p)
    where
        brds q = case previous q of
            Nothing -> [board q]
            Just r  -> board q : brds r

steps :: IO Puzzle -> IO Int
steps p = do
    pUnwrapped <- p
    let ret = length (boards pUnwrapped) - 1
    return ret

toBoard :: String -> [Int]
toBoard input = toIntBoard (words <$> (drop 1 . clearInput . lines $ input))

clearInput :: [String] -> [String]
clearInput xs = filter (/="") $ map (head . splitOn "#") xs

toIntBoard :: [[String]] -> [Int]
toIntBoard = concatMap (map read)

checkArgs :: [String] -> IO [String]
checkArgs a = if null a then putStrLn "Usage:␣stack␣exec␣nparzzle-exe␣<file>" >>

main :: IO ()
main = do
    args <- checkArgs =<< getArgs
    txt <- readFile $ head args
```

```
let gameList = splitOn "#" txt
    games = map toBoard gameList
-- print gameList
-- print games
-- print $ length games
let sols = map (solve . initPuzzle) games
mapM_ (\sol -> do
          numSteps <- steps sol
          print numSteps
      ) sols
```

5 References

1. https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Weijin-Zhu-Spring-2020.pdf
2. https://doi.org/10.1002/int.10027
3. https://guptaanna.github.io/15418Project/
4. https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/