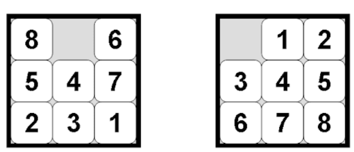
Parallel Functional Programming Fall 2021
Project Report – ParFifteenPuzzle

Kuan-Yao Huang - kh3120@columbia.edu
Aditya Sidharta - aks2266@columbia.edu

December 22, 2021

Problem Formulation

15 Puzzle is a sliding puzzle, which consists of $(N \times N)$ square tiles, where each squared tile is numbered from 1 to $(N^2 - 1)$, leaving a single square tile empty. Tiles that are located adjacent to the empty tile can be moved by sliding them horizontally, or vertically. The goal of the puzzle is to place the tiles in numerical order, leaving the last tile to be located at the bottom right corner of the frame.



It should be noted that not all of the initial state of 15 puzzle is solvable. 15 puzzle is solvable if:

1. N is odd
2. N is even, and the blank tile is on the even / odd row (counting from the bottom row), and the number of inversions is odd / even

Inversion is defined as the number of pairs (a, b), where $a > b$, but a appears before b if we were to flatten the number arrays into a single row. For example, [2 1 3 5 4 6 7 8] has 2 inversions $(2, 1), (5, 4)$.

Methods - A* Algorithm and Other Sequential Implementation

Optimal Solution: Breadth-first-search

Breadth-first-search is the most widely used optimal solver for 15puzzle problem. Starting from the initial state, we collect the neighbors into a queue, and then explore the neighbors layer by layer. However, since number of possible states for 16 puzzle problem is $\frac{16!}{2} =$ 20922789888000(and 24puzzle problem has $\frac{24!}{2} = 7.76 \times 10^{24}$). It is impractical to use this method to solve 15puzzle problem.

Approximation Algorithm I: Greedy Algorithm

Greedy algorithm can perform the approximation to this problem. First, we finish the first two element of the puzzle, and then we solve the row from above to bottom sequentially. However, this algorithm usually not giving as good enough steps.

Approximation Algorithm II: A^* Algorithm

Upon neighbor exploration, it is intuitive to choose the one that is the most "similar" to our final status. We can design a heuristic approach to measure the similarity between two states such as Manhattan distance or Hamming distance. So exploring the state with the best similarity can help to accelerate the process.

We adopted the A^* algorithm to help us minimize the effort to backtrack all of the possible steps. A^* algorithm is an informed search algorithm, which aims to find a path to a given goal node having the smallest cost $c(n)$

$$c(n) = f(n) + g(n)$$

Where $f(n)$ is defined as the step used from starting to current state, and $g(n)$ is the heuristic function that estimates the cost of the cheapest path, attainable or not, from the current state to the goal state. In the 15 puzzle problem, the heuristic function can be defined as the sum of distances between the current entry and the target entry of the digits in that entry. The distance metrics can be chosen to be Manhattan distance or Hamming distance.

A priority queue of the possible configurations prioritizing minimal cost functions is kept during solving. We iteratively pop the most heuristically probable configuration, compute possible next steps and push them to the priority queue. The algorithm will stop when the configuration reaches the goal state as seen on the following algorithm A. 1.

Algorithm 1 A^* algorithm

procedure MANHATTANDISTANCE(S)
    cost ← 0
    for i in 1 →N^2 **do**
        x, y ← divmod $S[i]$
        targetx, targety ← divmod i
        cost ← cost + $\|(x, y), (targetx, targety)\|_1$
    end for
    Return cost
end procedure
Input: Initial State × K
Output: Path length
procedure ASTARALGORITHM(S_i, S_e)
    HashMap ▷ storing visited states
    PriorityQueue pq(S_i, priority=ManhattanDistance(S_i), length=0) ▷ candidates
    while ! pq.empty( ) **do**
        if pq.top().state == S_e **then**
            Return pq.top().length
        end if
        neighbors ← getNeighbors pq.top()
        validNeighbors ← filter neighbors by mp
        pq.pop()
        for neighbor in validNeighbors **do**
            cost ← ManhattanDistance(neighbor)
            Add (neighbor, cost, length + 1) to pq
            Add neighbor state to HashMap
        end for
    end while
    Return −1
end procedure

Haskell Implementation

We design a puzzleState data type, which includes moves away from initial state, Manhattan distance with respect to target state, position of empty cell, and the current status.

```haskell
-- | PuzzleState contains the current move (fn), distance to goal (gn), current position of blank tile (zeroPos), and the current board state (state)
data PuzzleState = PuzzleState {fn::Int,
                                gn::Int,
                                zeroPos::Int,
                                state::Array U DIM1 Int} deriving (Show, Eq)

-- | cmpUboxarray performs comparison between two different arrays, perfomed by doing pairwise comparison across the subsequent values in
```

```
the two arrays
cmpUboxarray:: Array U DIM1 Int -> Array U DIM1 Int -> Ordering
cmpUboxarray a1 a2 = cmp a1 a2 0
     where cmp a1 a2 idx | idx ==  R.size (R.extent a1) = GT
                         | a1!(Z :. idx) == a2!(Z :. idx) = cmp a1 a2 (idx+1)
                         | otherwise = compare (a1!(Z :. idx)) (a2!(Z :. idx))

-- | PuzzleState is ordered by the total incurred cost and distance to goal (fn + gn). Else, it perform comparison between the two array
instance Ord PuzzleState where
     PuzzleState a b _ s1 `compare` PuzzleState c d _ s2 = if a+b /= c+d then (a+b) `compare` (c+d) else cmpUboxarray s1 s2
```

We introduced a repa array of size $N * N$ for storing a state, so we can conveniently generate a swapped array when moving the empty entry. In addition, we also define the ordering between different states to help us compare the priority. The state with lower $f_n + g_n$ is prioritized when doing neighbor expansion.

We also introduced priority queue from package PSQueue and HashMap from unordered-containers. We choose these packages based on their relative performances.

To measure the similarity between a state and target state, we introduced the Manhattan distance, which can be efficiently computed. For example, the cost function of state

$$\begin{bmatrix} 1 & 4 & 2 \\ 3 & 0 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

with respect to

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

is 1(digit 1)+ 1(digit 4)+2 (digit 0) = 4. We also tried using Hamming distance, but this metric usually gives us a slower performance.

```
-- | manhattanDist calculates the total distance of the current state (cur) to the goal board with size (n), performing recurrsion using (idx)
manhattanDist :: Source r Int =>  Array r DIM1 Int -> Int -> Int-> Int
manhattanDist cur idx n | idx == R.size (R.extent cur) = 0
                        | otherwise = diff idx (cur ! ( Z :. idx)) + manhattanDist cur (idx+1) n
                        where diff x y = abs (x `mod` n - y `mod` n) + abs (x `div` n - y `div` n)

-- | hammingDist calculates the number of wrong tiles of the current state (cur) to the goal board with size (n), performing recursion using (idx)
hammingDist :: Source r Int =>  Array r DIM1 Int -> Int -> Int ->Int
hammingDist cur idx n | idx == R.size (R.extent cur) = 0
                      | otherwise = diff idx (cur!(Z :. idx)) + hammingDist cur (idx+1) n
```

```
                    where diff x y | x == y = 1
                                   | otherwise = 0
```

Test Cases Generation

In this project, we have generated our test cases by python. Since even the best solver would likely to take forever to solve some randomly generated cases using A^* algorithm. We limit our test case that is less than 120 steps from target configuration.

Method - Parallel Fifteen Puzzle Implementation

Unlike the other graph search / path finding algorithm, it is non-trivial for us to parallelize A* algorithm, as each time step, the algorithm will try to evaluate the state in the priority queue with the lowest total cost $f(n)$, and expand the neighbor of the chosen state and pushing it back to the priority queue. Some of the difficulty in parallelizing this algorithms are:

1. Parallel threads that is working on a single priority queue might induce race conditions - each thread needs to lock the priority queue to obtain the most potential state, and lock the priority queue to push its neighbors. This will also inhibit concurrency as it needs to queue to update the priority queue

2. In order to avoid redundancy in our computation, we employ Hash Map along with our A* algorithm in order to avoid repeated states visit. Therefore, to perform parallel algorithm, this Hash-map will also potentially cause a race conditions without proper locking. This might also inhibit concurrency.

3. Even though its possible for the non-best states in the priority queue will be part of the optimal / shortest path, most of the states that have high cost does not have a lot of potential. Therefore, this will only results in wastage of computation if we do not choose the expansion strategy on the priority queue carefully.

After brainstorming to solve the potential issues that we might face, we employ three different parralelization strategy:

1. Parallelizing the Neighbor state calculation in each step of A* algortihm *(ParNeighbor)*

2. Parallelizing the number of Priority Queues used to solve a single puzzle *(ParPQ)*

3. Parallelizing the algorithm over k-puzzles *(ParPuzzle)*

ParNeighbor

The first parallelism strategy that comes into our mind for the A^* algorithm is to perform a parallel concurrent neighbor expansion, where the calculation of possible neighbors are parallelized. Within the original sequential A* algorithm, the only 'map' operation that does not depend on the previous step is only on the calculation of possible neighboring state and its cost function (Manhattan Distance). The parallelization attempt is given as follows:

```
getAllNeighborPar:: PuzzleState -> Int -> [PuzzleState]
getAllNeighborPar p n = catMaybes (runEval $ do
    a <- rpar (getUpNeighbor p n)
    b <- rpar (getDownNeighbor p n)
    c <- rpar (getLeftNeighbor p n)
    d <- rpar (getRightNeighbor p n)
return [a, b, c, d])
```

Nevertheless, as the Manhattan distance calculation is not expensive, this will more likely create a massive overhead from the spark and thread creations. Thus, we need to perform parallelization using different strategies.

ParPSQ

Intuitively, in each of the time step, its possible that the state that currently has the lowest cost function might not be the most optimal path. In other words, in A* algorithm, its possible that we stop exploring a certain path after we realize that the current path that we explore is impossible to be the best path solution, and continue to explore the second best path, and so on.

Thus, a more effective solution is to perform parallelization by creating multiple sparks on expansion on the top-k ($k <= \|\text{pq}\|$) elements of the priority queue, representing the top-k potential path candidates. As explained in the previous paragraph, It is difficult for us to perform this using a single priority because of the potential concurrency issue. In order to avoid this, we then try to employ k-different priority queues to explore different k states independently. In the implementation of this algorithm, the Hash Map was copied over to each of the threads in order to avoid concurrent read-write issues on the Hash Map as well. We realize that the choice of implementing independent, k-Hash Map for each of the threads might cause a trade-off on the computation, as we need to recompute the same state as each of the thread does not share the same hash map, but we realize that this might be the best solution for now to avoid concurrency issues on Haskell Hash Map.

The algorithm is as follows

Algorithm 2 Parallel PSQ

while PQ.size(pq) < k **do**
    if pq.top().state == S_{target} **then**
        Return pq.top().length
    end if
    neighbors ← getNeighbors pq.top()
    validNeighbors ← filter neighbors by mp
    pq.pop()
    for neighbor in validNeighbors **do**
        cost ← ManhattanDistance(neighbor)
        Add (neighbor, cost, length + 1) to pq
        Add neighbor state to HashMap
    end for
end while
for s in pq **do**
    Create a thread with ipq = (s), run Sequential A* algorithm on (ipq, HashMap)
end for
if any(complete(thread)) **then**
    Kill all other threads
end if
Return result(thread)

One huge part of Haskell Strategies implementation is that it guarantees deterministic parallelism, such that the result of the function is deterministic, despite the algorithm being evaluated in parallel setting. The original output of our sequential A* algorithm on 15-puzzle returns the number of steps taken in order to solve the puzzle. As ParPSQ will return non-deterministic result when we use the original output, as any of the thread that is completed first might be outputted, we have changed the output for the ParPSQ algorithm, outputting $True$ if the puzzle is solvable, $False$ otherwise. In this setting, we are able to guarantee the determinism in our function.

ParPuzzle

Lastly, similar to the Sudoku solution discussed during the lecture, another obvious implementation of Parallelization is to leave the Sequential A* algorithm untouched, and instead paralellize the solver over different puzzles. In order to regulate the number of sparks created in order to avoid buffer pool overflow, `parBuffer` was used. This implementation will achieve significant speed up as each of the thread will be able to solve the puzzle as fast as the sequential implementation, i.e there are no sequential dependency in between two different puzzles.

```
parSolveKpuzzle:: Handle -> Int -> IO()
parSolveKpuzzle handle k = do
allpuzzles <- getAllPuzzles handle k
let result = map solveOnepuzzle allpuzzles ‘using‘ parBuffer 100 rseq
```

```
print result
```

Evaluation and Results

ParNeighbor - Parallelizing Neighbor Expansion

	ParallelPSQ (k=5)
1-Core	13.15
2-Core	13.38
3-Core	13.8
4-Core	14.7
5-Core	15.2

As expected, the parallel neighbor expansion does not work, as we see that the time taken to complete 100 4x4 puzzle actually increase as we increase the number of cores. This is expected, since the extra amount of overhead from spark creation when we increase the number of cores outweighs the benefit of calculating the Manhattan distance in parallel. Furthermore, as the number of possible neighbors in each step of A* algortihm is only four (Swap blank tile above, below, left, and right), thus this algorithm will also not scale well even though if it worked.

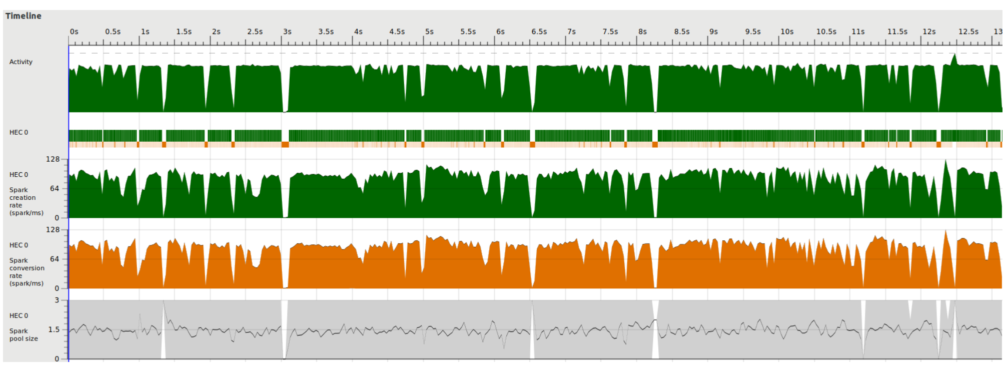


Figure 1: Parallel Neighbor core = 1

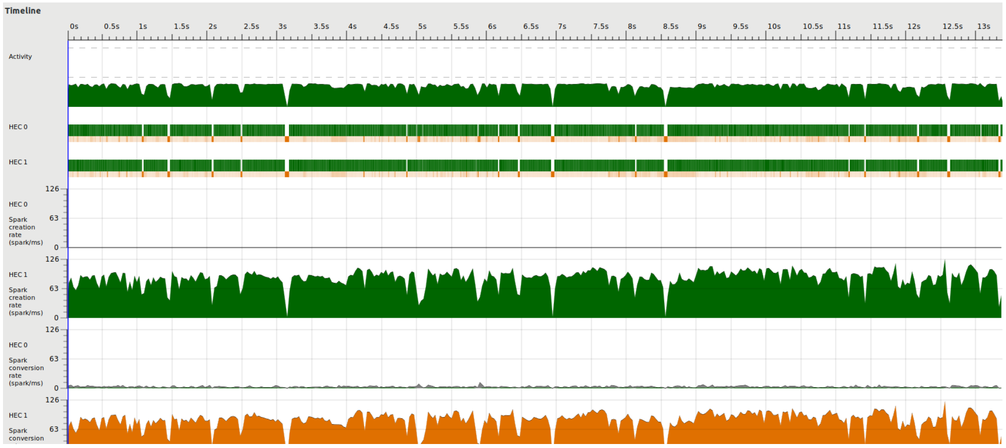


Figure 2: Parallel Neighbor core = 2

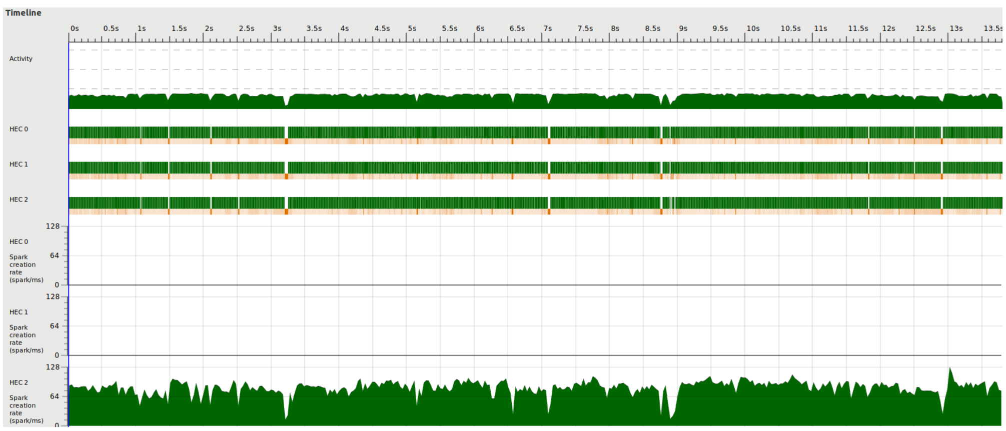


Figure 3: Parallel Neighbor core = 3

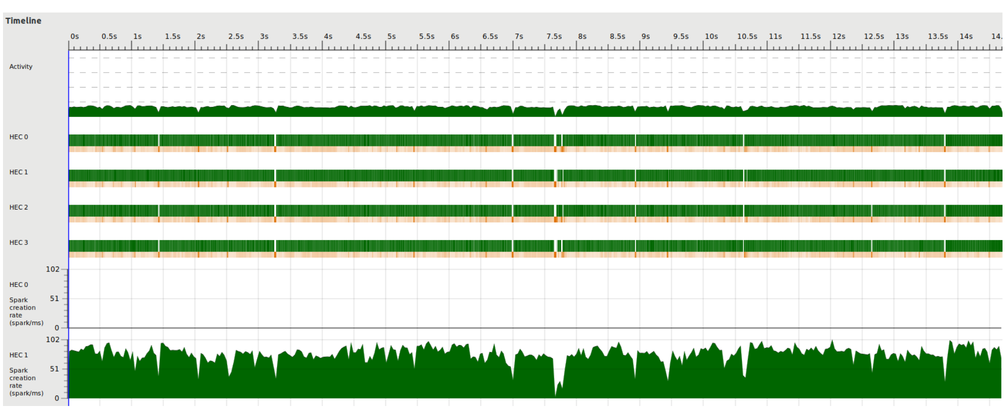


Figure 4: Parallel Neighbor core = 4

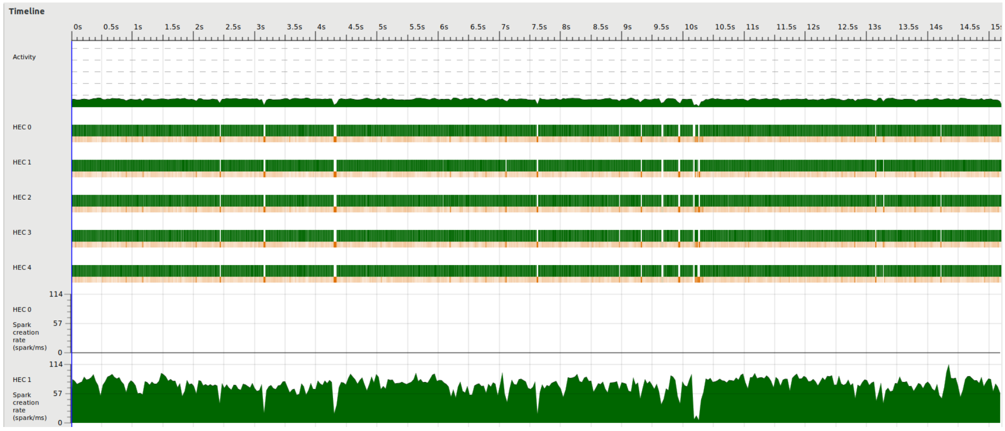
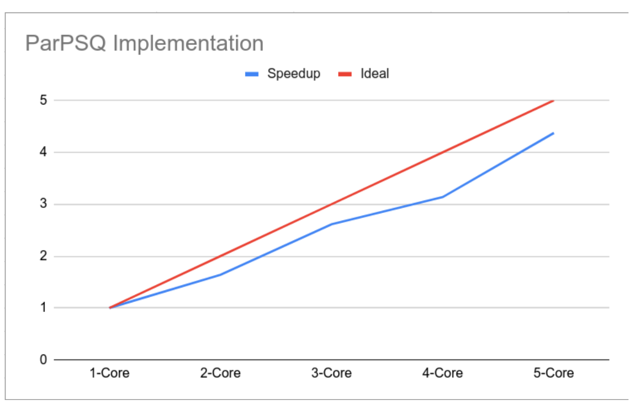


Figure 5: Parallel Neighbor core = 5

As we can see from the threadscope graph, as we are calling this algorithm at each timestep, even though in each timestep we are only calling up to 4 threads at the same time, the number of calls that we made is huge, and thus the job seems to be well distributed among all cores.

ParPSQ - Parallelizing k-Priority Queues

	ParallelPSQ (k=5)	Speedup	Ideal
1-Core	27.54	1.00	1
2-Core	16.8	1.64	2
3-Core	10.54	2.61	3
4-Core	8.78	3.14	4
5-Core	6.3	4.37	5



The algorithm seems to work well, as it offers speedup as compared to the sequential algorithm. This proves that the state in the priority queue that has the lowest cost $f(n)$ in the initial phase of the A* algorithm is not necessarily the best solution, as often the algorithm find an optimal path in exploring k-th best state in the priority queue.

In our experiment, we are fixing the number of Parallel Queues to be 5. Thus, it is understandable that in the 1-Core scenario, we are actually performing worse as compared to the sequential algorithm, as now the Parallel PSQ algorithm needs to interleave computation of various priorities queues in a single core, causing the workload to be multiplied as compared to the sequential algorithm implementation. However, as we increase the number of cores, each of the core will be able to take up different priority queues, and terminating the algorithm once any of the thread returns a result. Thus, this offers a significant improvement, up to 4.37x the 1-core implementation of ParPSQ and roughly 2x as compared to the sequential implementation. It is understandable that the performance is still sub par compared to the embarrassingly parallel *ParPuzzle* algorithm, but nevertheless we are pretty delighted with the result. We believe that increasing the number of cores as well as the number of priority queues to a larger number will not yield any significant improvement to the final result due to 2 reasons. Firstly, the Amdahl's law states that there is a limit on the speedup on parallel algorithm depending on the severity of the sequential fraction of the task. Secondly, we believe that by increasing the number of priority queues, the extra thread that we create will explore state that is less and less likely to be the optimal path, as it currently has a large cost $c(n) = f(n) + g(n)$. Thus, it is less likely to offers any speedup.

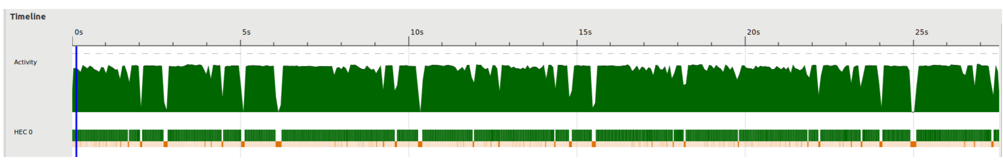


Figure 6: Parallel PSQ core = 1

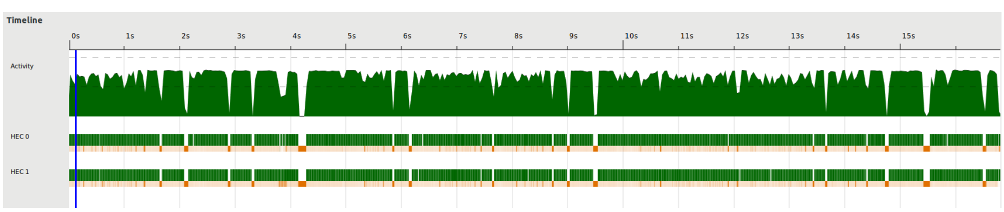


Figure 7: Parallel PSQ core = 2

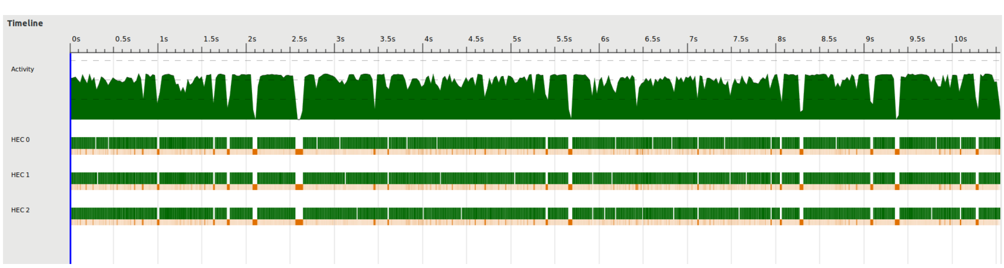


Figure 8: Parallel PSQ core = 3

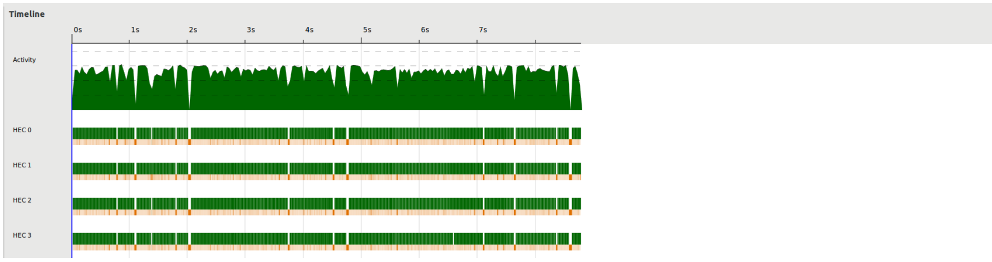


Figure 9: Parallel PSQ core = 4

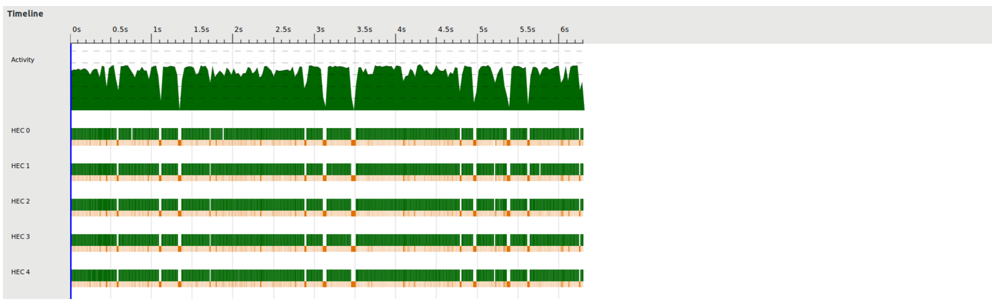


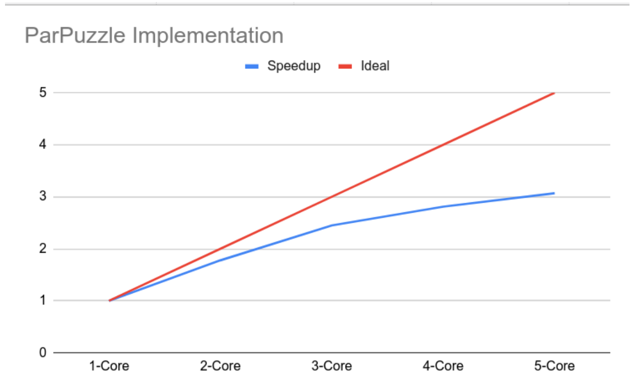
Figure 10: Parallel PSQ core = 5

As shown in the figure above, there are no idle cores and the job seems to be distributed well as long as the number of priorities queue that is used is bigger than the number of cores used. In the case where it is smaller, it is possible that there will be idle time amongst any of the cores as there are not enough jobs to be passed around.

ParPuzzle - Case Level Parallelism

The below is the result for case level parallelism

	ParallelPuzzle	Speedup	Ideal
1-Core	12.73	1.00	1
2-Core	7.16	1.78	2
3-Core	5.2	2.45	3
4-Core	4.53	2.81	4
5-Core	4.15	3.07	5



The figures below shows the workload is nearly evenly distributed between each cores. There is no new spark generation after we scan through the array by `parBuffer 100 rseq`, so the size of spark pool will have a peak at the begining and decrease with time. We noticed that the barbage collection time increases as number of threads increases.

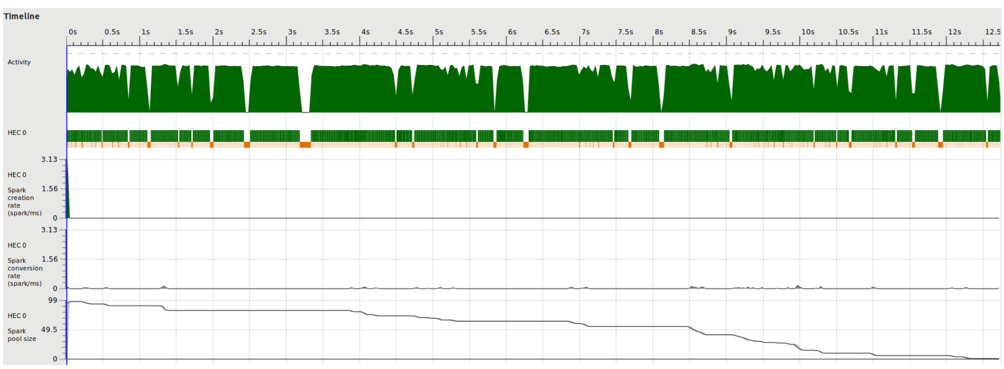


Figure 11: Parallel Puzzle core = 1

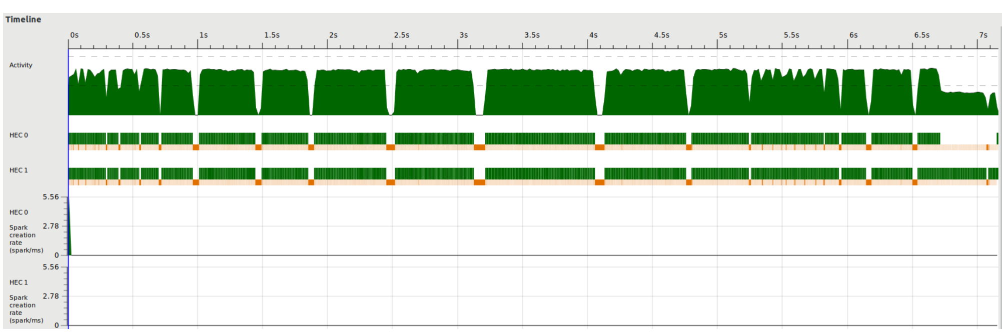


Figure 12: Parallel Puzzle core = 2

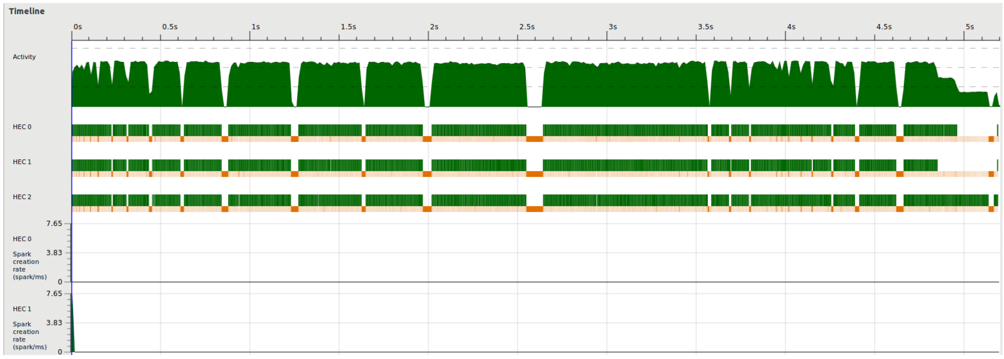


Figure 13: Parallel Puzzle core = 3

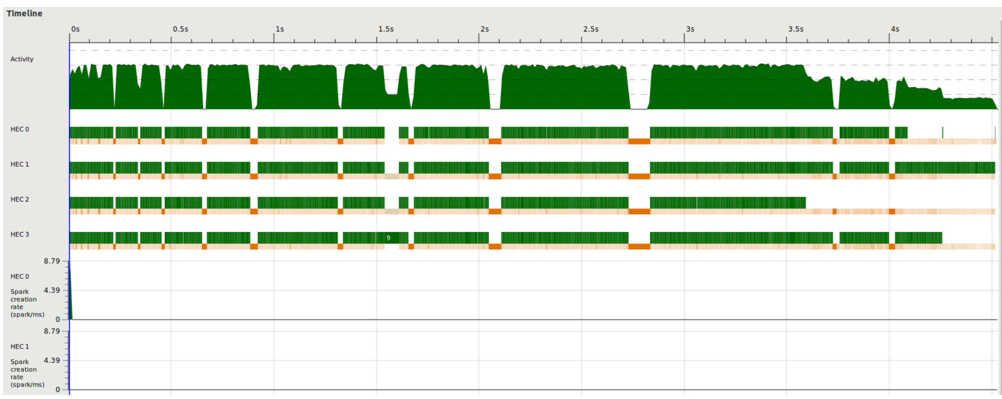


Figure 14: Parallel Puzzle core = 4

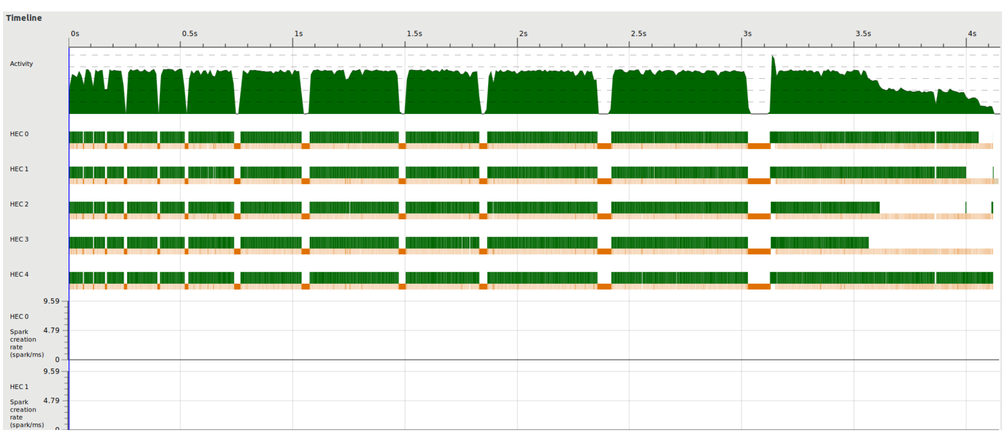


Figure 15: Parallel Puzzle core = 5

We also using other strategy such as `parList rseq`, `parList rpar`, `parBuffer 00 rpar`, their performances are comparable. If we decrease the parBuffer size to lower than 100, more than one spark peak will be found since for this algorithm, we have exactly 100 sparks in for case level parallelism.

Summary Table

	Sequential	ParallelNeighbor	ParallelPSQ (k=5)	ParallelPuzzle
1-Core	13.07	13.15	27.54	12.73
2-Core	12.89	13.38	16.8	7.16
3-Core	13.28	13.8	10.54	5.2
4-Core	13.8	14.7	8.78	4.53
5-Core	15.17	15.2	6.3	4.15

Compared to Sequential	Sequential	ParallelNeighbor	ParallelPSQ (k=5)	ParallelPuzzle
1-Core	1.00	0.99	0.47	1.03
2-Core	1.01	0.98	0.78	1.83
3-Core	0.98	0.95	1.24	2.51
4-Core	0.95	0.89	1.49	2.89
5-Core	0.86	0.86	2.07	3.15

Compared to 1-Core	Sequential	ParallelNeighbor	ParallelPSQ (k=5)	ParallelPuzzle
1-Core	1.00	1.00	1.00	1.00
2-Core	1.01	0.98	1.64	1.78
3-Core	0.98	0.95	2.61	2.45
4-Core	0.95	0.89	3.14	2.81
5-Core	0.86	0.87	4.37	3.07

Conclusion

It is rewarding to challenge a problem that is not easily parallelizable. Our experiments show parallelism can help with exploring better search path which is heuristically less favorable. In our quest to parallelize the A^* algorithm for 15 puzzle problem, we found the main obstacle hindering us to implement high efficiency algorithm is the non-deterministic nature of haskell. In addition, we tried several different parallelization methods and found not all of them is worth parallelization. Thirdly, we found balancing workload from different cores are nontrivial and it needs efforts on experiments. Last but not least, we learn a lesson about the separability between algorithm and parallelism in haskell.

Future Works

The main issue with our implementation is that our parallel solver may do repeated jobs. We found that if there exist a shared hashmap that can handle insertion and lookup concurrently, it is very likely to improve our solver, especially for more difficult cases.

We made an effort to the documentation of this project, as seen in the directory `doc/` and `README.md`. In addition, We are willing to make public this directory.

Appendix: Code and Unit Tests

ParallelPuzzle.sh Case Level Parallelism

```haskell
module ParallelPuzzle where

import Solver (parSolveKpuzzle)
import Parse (readInt)
import System.Exit(die)
import System.Environment(getArgs, getProgName)
import System.IO(openFile, IOMode(ReadMode))

main :: IO ()
main = do
    args <- getArgs
    case args of
        [filename] -> do
            handle <- openFile filename ReadMode
            k <- readInt handle
            parSolveKpuzzle handle k
        _ -> do
            pn <- getProgName
            die $ "Usage: "++pn++" <filename>"
```

ParallelNeighbor.sh Paralleling Neighbor Expansion

```haskell
module ParallelNeighbor where

import Solver (parNeighborSolveKpuzzle)
import Parse (readInt)
import System.Exit(die)
import System.Environment(getArgs, getProgName)
import System.IO(openFile, IOMode(ReadMode))

main :: IO ()
main = do
    args <- getArgs
    case args of
        [filename] -> do
            handle <- openFile filename ReadMode
            k <- readInt handle
            parNeighborSolveKpuzzle handle k
        _ -> do
            pn <- getProgName
            die $ "Usage: "++pn++" <filename>"
```

ParallelPriorityQueue.sh Paralleling k-Priority Queue

```haskell
module ParallelPriorityQueue where

import Solver (parPSQSolvePuzzle)
import Parse (readInt)
import System.Exit(die)
import System.Environment(getArgs, getProgName)
```

```haskell
import System.IO(openFile, IOMode(ReadMode))

main :: IO ()
main = do
    args <- getArgs
    case args of
        [filename] -> do
            handle <- openFile filename ReadMode
            k <- readInt handle
            parPSQSolvePuzzle handle k
        _ -> do
            pn <- getProgName
            die $ "Usage: "++pn++" <filename>"
```

Solver.hs

```haskell
{-# LANGUAGE FlexibleContexts #-}

module Solver where
import System.IO (hGetLine, Handle)
import Data.PSQueue as PQ (PSQ, singleton, prio, size, findMin, deleteMin,
    key, insert, toList)
import Data.Maybe ( fromJust, catMaybes )
import Data.HashMap.Strict as H (HashMap, singleton, member, lookup,
   insert)
import Data.Array.Repa as R (Array, U, DIM1, fromListUnboxed, Z (Z), (:.)
   ((:.)), (!), index, Shape (size), Source (extent), DIM0, zipWith, D,
   computeUnboxedS )
import Data.List ( zip4)
import Control.Monad ( forM, void )
import Control.Parallel.Strategies(rpar, using, parList, rseq, parBuffer)
import Control.Concurrent ( newEmptyMVar, newMVar, forkIO, tryPutMVar,
   takeMVar, putMVar, readMVar, killThread)
import GHC.IO (unsafePerformIO)

import Puzzle ( PuzzleState, PuzzleState(PuzzleState, gn, fn, state),
   getZeroPos, swapTwo, getAllNeighbor, getAllNeighborPar, solvability)
import Metrics ( manhattanDist )
import Parse ( readInt, getStateVector, getAllPuzzles)

-- | getValidNeighbor filters all neighbor puzzles that improves (fn) or
   have not been discovered previously (not in mp)
getValidNeighbor::[PuzzleState] -> H.HashMap String Int-> [PuzzleState]
getValidNeighbor ps mp = filter (filterInMap mp) ps

-- | filterInMap returns True if the puzzle (puzzle) is not in the HashMap
    (mp) or if the puzzle can now be reached in less steps (fn)
filterInMap :: HashMap String Int -> PuzzleState -> Bool
filterInMap mp puzzle = not (H.member key mp) || fromJust (H.lookup key mp
   ) > fn puzzle
    where key = getHashKey $ state puzzle

-- | addMap add all of the puzzle states (ps) into the given HashMap (mp)
addMap :: Foldable t => t PuzzleState -> HashMap String Int -> HashMap
```

```haskell
    String Int
addMap ps mp = foldr (\ p -> H.insert (getHashKey (state p)) (fn p)) mp ps

-- | addPSQ adds all of the given puzzle states (ps) into the PriorityQueue (psq)
addPSQ :: [PuzzleState] -> PSQ PuzzleState Int -> PSQ PuzzleState Int
addPSQ ps psq = foldr(\ p -> PQ.insert p (fn p + gn p)) psq ps

-- | getHashKey turns the hash result from the given array (li) and return string as the hash key. hash [0, 3, 1, 2] -> "00030102"
getHashKey:: Array U DIM1 Int -> String
getHashKey li = show $ hash li 0

-- | hash perform simple hash function on the given array (l), using recursive function on idx. hash [0, 3, 1, 2] -> "00030102"
hash :: Integral a => Array U DIM1 Int -> Int -> a
hash l idx | (Z:.idx) == R.extent l = 0
           | otherwise = fromIntegral (l!(Z:.idx)) + 100 * hash l (idx+1)

-- | solveBool perform sequential solving on 8-puzzle using A* algorithm, returning True if the puzzle is solvable
solveBool :: (PSQ PuzzleState Int,  Array U DIM1 Int, Int, H.HashMap String Int)-> IO Bool
solveBool (psq, target, n, mp) = do
    let top       = fromJust $ findMin psq
        npsq      = deleteMin psq
        depth     = fn $ key top
        curarray  = state $ key top

    -- if PQ.size psq == 0 then
    if PQ.size psq == 0 then
        return False
    else if curarray == target then
        return True
    else do
        let neighborList = getAllNeighbor (key top) n
            validNeighborList = getValidNeighbor neighborList mp
            newmap = addMap validNeighborList mp
            newpsq = addPSQ validNeighborList npsq
        solveBool (newpsq, target, n, newmap)

-- | solve perform sequential solving on 8-puzzle using A* algorithm
solve :: (PSQ PuzzleState Int,  Array U DIM1 Int, Int, H.HashMap String Int)-> IO Int
solve (psq, target, n, mp) = do
    let top       = fromJust $ findMin psq
        npsq      = deleteMin psq
        depth     = fn $ key top
        curarray  = state $ key top

    -- if PQ.size psq == 0 then
    if PQ.size psq == 0 then
        return (-1)
```

```
    else if curarray == target then
        return depth
    else do
        let neighborList = getAllNeighbor (key top) n
            validNeighborList = getValidNeighbor neighborList mp
            newmap = addMap validNeighborList mp
            newpsq = addPSQ validNeighborList npsq
        solve (newpsq, target, n, newmap)

-- | solveOnepuzzle perform solving on a single 8-puzzle
solveOnepuzzle :: (Int, [Int]) -> Int
solveOnepuzzle (n, state) | solvable = unsafePerformIO $ solve (psq, target, n, mp)
                          | otherwise = -1
  where array = fromListUnboxed (Z :. (n*n) :: DIM1) state
        target = fromListUnboxed (Z :. (n*n) :: DIM1) [0..(n*n-1)]
        gn     = manhattanDist array 0 n
        psq    = PQ.singleton (PuzzleState 0 gn (getZeroPos array 0) array) gn
        mp     = H.singleton (getHashKey array) 0 -- a hashmap storing visited states -> fn
        solvable = solvability array (getZeroPos array 0) n

-- | solveParNeighbor perform solving by parallelizing the calculation of GetAllNeighbor into 4 different threads
solveParNeighbor :: (PSQ PuzzleState Int , Array U DIM1 Int , Int , H.HashMap String Int)-> IO Int
solveParNeighbor (psq, target, n, mp) = do
    let top       = fromJust $ findMin psq
        npsq      = deleteMin psq
        depth     = fn $ key top
        curarray  = state $ key top

    -- if PQ.size psq == 0 then
    if PQ.size psq == 0 then
        return (-1)
    else if curarray == target then
        return depth
    else do
        let neighborList = getAllNeighborPar (key top) n
            validNeighborList = getValidNeighbor neighborList mp
            newmap = addMap validNeighborList mp
            newpsq = addPSQ validNeighborList npsq
        solveParNeighbor (newpsq, target, n, newmap)

-- | solveParPSQ perform solving by creating multiple priority queues and abort the other thread once we have solved the puzzle
solveParPSQ :: (PSQ PuzzleState Int, Array U DIM1 Int, Int, HashMap String Int) -> IO Int
solveParPSQ (psq, target, n, mp) = do
    let top      = fromJust $ findMin psq
        npsq     = deleteMin psq
        depth    = fn $ key top
        curarray = state $ key top
```

```
        k = 5

    -- if PQ.size psq == 0 then
    if PQ.size psq == 0 then
        return (-1)
    else if curarray == target then
        return 1
    else if PQ.size psq < k then do
        let neighborList = getAllNeighbor (key top) n
            validNeighborList = getValidNeighbor neighborList mp
            newmap = addMap validNeighborList mp
            newpsq = addPSQ validNeighborList npsq
        solveParPSQ (newpsq, target, n, newmap)
    else do
        let length = PQ.size psq
        resultV <- newEmptyMVar
        runningV <- newMVar length
        threads <- forM [PQ.singleton (key x) (prio x) | x <- PQ.toList
   psq] $ \ipsq -> forkIO $ do
            if unsafePerformIO(solveBool(ipsq, target, n, mp)) then void (
   tryPutMVar resultV 1) else (do m <- takeMVar runningV

                                    if m == 1

                                          then void (tryPutMVar resultV 0)

                                          else putMVar runningV (m-1))
        result <- readMVar resultV
        mapM_ killThread threads
        return result


-- | puzzleSolver is the base function for other solver
puzzleSolver :: (Num a, Show a, Num v) => Handle -> Int -> ((PSQ
   PuzzleState Int, Array U DIM1 Int, Int, HashMap String v) -> IO a) ->
   IO ()
puzzleSolver handle 0 solver = return ()
puzzleSolver handle k solver = do
    n <- readInt handle
    matrix <- getStateVector handle n n
    let array  = fromListUnboxed (Z :. (n*n) :: DIM1) $ concat matrix
        target = fromListUnboxed (Z :. (n*n) :: DIM1) [0..(n*n-1)]
        gn     = manhattanDist array 0 n
        psq    = PQ.singleton (PuzzleState 0 gn (getZeroPos array 0) array
   ) gn
        mp     = H.singleton (getHashKey array) 0 -- a hashmap storing
   visited states -> fn
        solvable = solvability array (getZeroPos array 0) n

    step  <- if solvable then solver (psq, target, n, mp) else return (-1)
    print step

    puzzleSolver handle (k-1) solver

```

```

-- | solveKpuzzle perform solving on mutliple 8-puzzle in a sequential manner
solveKpuzzle :: Handle -> Int -> IO ()
solveKpuzzle handle k = puzzleSolver handle k solve

-- | parSolveKpuzzle perform solving on mutliple 8-puzzle in a parallel manner, by sparking different threads to solve different puzzles
parSolveKpuzzle:: Handle -> Int -> IO()
parSolveKpuzzle handle k = do
    allpuzzles <- getAllPuzzles handle k
    let result = map solveOnepuzzle allpuzzles `using` parBuffer 100 rseq -- `using` parList rseq
    print result

-- | parNeighborSolveKpuzzle perform solving on multiple 80puzzle in a parallel manner, by sparking different threads to calculate the valid Neighbors
parNeighborSolveKpuzzle :: Handle -> Int -> IO()
parNeighborSolveKpuzzle handle k = puzzleSolver handle k solveParNeighbor

-- | parPSQSolvePuzzle is an interface to parPSQ
parPSQSolvePuzzle :: Handle -> Int -> IO()
parPSQSolvePuzzle handle k = puzzleSolver handle k solveParPSQ
```

Puzzle.hs

```
{-# LANGUAGE FlexibleContexts #-}
module Puzzle where
import Data.Array.Repa as R (Array, U, DIM1, fromListUnboxed, Z (Z), (:.) ((:.)), (!), index, Shape (size), Source (extent), DIM0, zipWith, D, computeUnboxedS )
import System.Random (mkStdGen)
import System.Random.Shuffle (shuffle')
import Metrics (manhattanDist)
import Data.Maybe (catMaybes)
import Control.Parallel.Strategies (runEval, rpar)

-- | PuzzleState contains the current moves (fn), distance to goal (gn), current position of blank tile (zeroPos), and the current board state (state)
data PuzzleState = PuzzleState {fn::Int,
                                gn::Int,
                                zeroPos::Int,
                                state::Array U DIM1 Int} deriving (Show, Eq)

-- | cmpUboxarray performs comparison between two different arrays, perfomed by doing pairwise comparison across the subsequent values in the two arrays
cmpUboxarray:: Array U DIM1 Int -> Array U DIM1 Int -> Ordering
cmpUboxarray a1 a2 = cmp a1 a2 0
    where cmp a1 a2 idx | idx ==  R.size (R.extent a1) = GT
                        | a1!(Z :. idx) == a2!(Z :. idx) = cmp a1 a2 (idx
```

```haskell
+1)
                         | otherwise = compare (a1!(Z :. idx)) (a2!(Z :. idx))

-- | PuzzleState is ordered by the total incurred cost and distance to goal (fn + gn). Else, it perform comparison between the two array
instance Ord PuzzleState where
    PuzzleState a b _ s1 `compare` PuzzleState c d _ s2 = if a+b /= c+d then (a+b) `compare` (c+d) else cmpUboxarray s1 s2

-- | generateArrays returns k number of shuffled matrix of size n for the input of 15-puzzle problem
generateArrays :: (Num a, Enum a) => Int -> a -> [[a]]
generateArrays 0 _ = []
generateArrays k n = let xs = [0..(n * n - 1)] in shuffle' xs (length xs) (mkStdGen k) : generateArrays (k -1) n

-- | formatArray takes the array (a) and the size of the puzzle (n) and return it as a string, according to the input text format of this program
formatArray :: [Int] -> Int -> String
formatArray [] n = ""
formatArray a n = unwords (map show (take n a)) ++ "\n" ++ formatArray (drop n a) n

-- | formatArrays takes the arrays (a:as) and return it as a string according to the input text format of this program
formatArrays :: [[Int]] -> String
formatArrays [] = ""
formatArrays (a:as) = show n ++ "\n" ++ formatArray a n ++ formatArrays as
    where
        n = floor(sqrt(fromIntegral(length a))) :: Int

-- | writeArrays takes the arrays and write it into the filename according to the input text format of this program
writeArrays :: [[Int]] -> FilePath -> IO ()
writeArrays arrays filename =
    writeFile filename (show n ++ "\n" ++ formatArrays arrays)
      where
        n = length arrays


-- | getZeroPos returns the idx within the given array (arr) where the blank tile is located. If fail, return -1
getZeroPos :: Source r Int => Array r DIM1 Int -> Int -> Int
getZeroPos arr idx | idx == R.size (R.extent arr) = -1
                   | arr!(Z :. idx) == 0 = idx
                   | otherwise = getZeroPos arr (idx+1)

-- | swapTwo perform swap between two elements in the array (arr), given two indexes, f and s in the array
swapTwo :: Source r Int => Int -> Int -> Array r DIM1 Int -> Array D DIM1 Int
swapTwo f s arr = R.zipWith (\x y->
```

```
    if      x == f then arr!(Z :. s)
    else if x == s then arr!(Z :. f)
    else y)  (fromListUnboxed sh [0..(R.size sh -1)]) arr
    where sh = R.extent arr


-- | getUpNeighbor return the subsequent PuzzleState by swapping the blank tile with the tile above it. If its impossible, return Nothing
getUpNeighbor :: PuzzleState -> Int -> Maybe PuzzleState
getUpNeighbor (PuzzleState f g ze reparray) n | row < 0 = Nothing
                                              | otherwise = Just $ PuzzleState (f+1) newg (row*n+col) newarray
  where oldrow = ze `div` n
        row = oldrow - 1
        col = ze `mod` n
        newarray = computeUnboxedS $ swapTwo (oldrow*n+col) (row*n+col) reparray
        newg = manhattanDist newarray 0 n

-- | getDownNeighbor return the subsequent PuzzleState by swapping the blank tile with the tile below it. If its impossible, return Nothing
getDownNeighbor :: PuzzleState -> Int -> Maybe PuzzleState
getDownNeighbor (PuzzleState f g ze reparray) n | row >= n = Nothing
                                                | otherwise = Just $ PuzzleState (f+1) newg (row*n+col) newarray
  where oldrow = ze `div` n
        row = oldrow + 1
        col = ze `mod` n
        newarray = computeUnboxedS $ swapTwo (oldrow*n+col) (row*n+col) reparray
        newg = manhattanDist newarray 0 n

-- | getLeftNeighbor return the subsequent PuzzleState by swapping the blank tile with the tile left to it. If its impossible, return Nothing
getLeftNeighbor :: PuzzleState -> Int -> Maybe PuzzleState
getLeftNeighbor (PuzzleState f g ze reparray) n | col < 0 = Nothing
                                                | otherwise = Just $ PuzzleState (f+1) newg (row*n+col) newarray
  where oldcol = ze `mod` n
        row = ze `div` n
        col = oldcol - 1
        newarray = computeUnboxedS $ swapTwo (row*n+oldcol) (row*n+col) reparray
        newg = manhattanDist newarray 0 n

-- | getRightNeighbor return the subsequent PuzzleState by swapping the blank tile with the tile right to it. If its impossible, return Nothing
getRightNeighbor :: PuzzleState -> Int -> Maybe PuzzleState
getRightNeighbor (PuzzleState f g ze reparray) n | col >=n = Nothing
                                                 | otherwise = Just $ PuzzleState (f+1) newg (row*n+col) newarray
  where oldcol = ze `mod` n
        row = ze `div` n
        col = oldcol + 1
```

```
        newarray = computeUnboxedS $ swapTwo (row*n+oldcol) (row*n+col)
   reparray
        newg = manhattanDist newarray 0 n

-- | getAllNeighbor return all of the neighboring state of the current
   PuzzleState
getAllNeighbor:: PuzzleState -> Int -> [PuzzleState]
getAllNeighbor p n = [x | Just x <- [getUpNeighbor p n, getDownNeighbor p
   n, getLeftNeighbor p n, getRightNeighbor p n]]


-- | getAllNeighborPar return all of the neighboring state of the current
   PuzzleState
getAllNeighborPar:: PuzzleState -> Int -> [PuzzleState]
getAllNeighborPar p n = catMaybes (runEval $ do
    a <- rpar (getUpNeighbor p n)
    b <- rpar (getDownNeighbor p n)
    c <- rpar (getLeftNeighbor p n)
    d <- rpar (getRightNeighbor p n)
    return [a, b, c, d])


-- | numinv check the number of inversions in the board (arr)
numinv :: Array U DIM1 Int -> Int
numinv arr = aux arr 0 1 0
    where aux arr i j r | i == R.size (R.extent arr) = r
                        | j == R.size (R.extent arr) = aux arr (i+1) (i+2)
    r
                        | arr!(Z:.i) == 0 || arr!(Z:.j) == 0 = aux arr i (
   j+1) r
                        | arr!(Z:.i) > arr!(Z:.j) = aux arr i (j+1) r
                        | arr!(Z:.i) < arr!(Z:.j) = aux arr i (j+1) (r+1)
                        | otherwise = error "inversion error!"

-- | solvability checks whether the given board (arr) with the current
   zero position (zeropos) is solvable 8-puzzle problem
solvability:: Array U DIM1 Int -> Int -> Int -> Bool
solvability arr zeropos n | odd n && even (numinv arr) = True
                          | even n && even (zeropos `div` n + 1) && even (
   numinv arr) = True
                          | even n && odd  (zeropos `div` n + 1) && odd  (
   numinv arr) = True
                          | otherwise  = False
```

Parse.hs

```
module Parse where

import System.IO (hGetLine, Handle)

-- | readInt parse the input handle and return an Integer from its first
   line
readInt :: Handle -> IO Int
readInt handle = do
```

```haskell
    str <- hGetLine handle
    return (read str::Int)

-- | printList print a given list (l) into IO
printList::Show a =>[a] -> IO ()
printList l =
    print $ show l

-- | getStateVector parse the input handle and return lists of list of
   integer, which is the initial game board
getStateVector :: Handle -> Int -> Int -> IO [[Int]]
getStateVector handle n 0 = return []
getStateVector handle n cur = do
    line <- hGetLine handle
    let tokens = (\x -> read x::Int) <$> words line
    post <- getStateVector handle n (cur-1)
    return (tokens:post)

-- | GetAllPuzzles read all of the matrices in the handle and return a
   list of (n, array) where n is the size of the puzzle and array is the
   initial state of puzzle
getAllPuzzles :: Handle -> Int -> IO [(Int, [Int])]
getAllPuzzles handle 0 = return []
getAllPuzzles handle k = do
    n <- readInt handle
    matrix <- getStateVector handle n n
    latter <- getAllPuzzles handle (k-1)
    return ((n, concat matrix): latter)
```

Metrics.hs

```haskell
{-# LANGUAGE FlexibleContexts #-}
module Metrics where
import Data.Array.Repa as R (Array, U, DIM1, fromListUnboxed, Z (Z), (:.)
   ((:.)), (!), index, Shape (size), Source (extent), DIM0, zipWith, D,
   computeUnboxedS )

-- | manhattanDist calculates the total distance of the current state (cur
   ) to the goal board with size (n), performing recurrsion using (idx)
manhattanDist :: Source r Int =>  Array r DIM1 Int -> Int -> Int-> Int
manhattanDist cur idx n | idx == R.size (R.extent cur) = 0
                        | otherwise = diff idx (cur ! ( Z :. idx)) +
   manhattanDist cur (idx+1) n
                        where diff x y = abs (x `mod` n - y `mod` n) + abs
    (x `div` n - y `div` n)

-- | hammingDist calculates the number of wrong tiles of the current state
    (cur) to the goal board with size (n), performing recursion using (idx
   )
hammingDist :: Source r Int =>  Array r DIM1 Int -> Int -> Int ->Int
hammingDist cur idx n | idx == R.size (R.extent cur) = 0
                      | otherwise = diff idx (cur!(Z :. idx)) +
   hammingDist cur (idx+1) n
                      where diff x y | x == y = 1
```

```
                                                         | otherwise = 0
```

Test case generator

```python
import random
import numpy as np

dirs = [-1,0,1,0,-1]

def swapzero(step, n):
    arr = np.array([i for i in range(n*n)])
    x , y = 0, 0
    for _ in range(step):
        d = random.randint(0,3)
        dx = x + dirs[d]
        dy = y + dirs[d+1]
        if dx >= 0 and dy >= 0 and dx < n and dy < n:
            tmp = arr[x*n+y]
            arr[x*n+y] = arr[dx*n+dy]
            arr[dx*n+dy] = tmp
            x = dx
            y = dy
    return arr


if __name__ == '__main__':

    case_num = 100
    outfile = "./input.txt"

    with open(outfile, 'w') as f:
        f.write(f"{case_num}\n")
        for i in range(case_num):
            size = 4
            f.write(f"{size}\n")
            l = swapzero(80, size)
            for i in range(l.shape[0]):
                if (i+1) % size == 0:
                    f.write(f"{l[i]}\n")
                else:
                    f.write(f"{l[i]} ")
```

Unit Test

```haskell
import Test.Tasty ( defaultMain, testGroup, TestTree )
import Test.Tasty.HUnit ( testCase, assertEqual, Assertion, (@?=) )
import Lib (numinv, getAllNeighborPar, solvability, getStateVector,
    getValidNeighbor, readInt, solveKpuzzle, generateArrays, formatArray,
    formatArrays, manhattanDist, hammingDist, getZeroPos, swapTwo,
    getUpNeighbor, PuzzleState (PuzzleState), getRightNeighbor,
    getLeftNeighbor, getDownNeighbor, getAllNeighbor, hash, getHashKey,
    addMap)
import System.IO (openFile, IOMode (ReadMode))
import Data.Array.Repa (DIM1, fromListUnboxed, Z (Z), (:.) ((:.)), Array,
    U, computeS)
```

```haskell
import Data.HashMap.Strict as H ( fromList, singleton )
import Data.PSQueue as PQ (fromList, singleton)

main :: IO ()
main = defaultMain unitTests

unitTests = testGroup "Unit Tests" [
  testCase "getStateVectorTest" getStateVectorTest,
  testCase "generateArraysTest" generateArraysTest,
  testCase "formatArrayTest" formatArrayTest,
  testCase "formatArraysTest" formatArraysTest,
  testCase "manhattanDistTest" manhattanDistTest,
  testCase "hammingDistTest" hammingDistTest,
  testCase "getZeroPosTest" getZeroPosTest,
  testCase "swapTwoTest" swapTwoTest,
  testCase "getUpNeighborTest" getUpNeighborTest,
  testCase "getDownNeighborTest" getDownNeighborTest,
  testCase "getLeftNeighborTest" getLeftNeighborTest,
  testCase "getRightNeighborTest" getRightNeighborTest,
  testCase "getAllNeighborTest" getAllNeighborTest,
  testCase "getAllNeighborParTest" getAllNeighborParTest,
  testCase "hashTest" hashTest,
  testCase "getHashKeyTest" getHashKeyTest,
  testCase "addMapTest" addMapTest,
  testCase "getValidNeighborTest" getValidNeighborTest,
  testCase "numinvTest" numinvTest,
  testCase "solvabilityTest" solvabilityTest]

getStateVectorTest :: Assertion
getStateVectorTest = do
  x <- fn "test/test.txt"
  x @?= [0,4,2,1,3,8,6,5,7]
    where fn filename = do
                          handle <- openFile filename ReadMode
                          do
                              k <- readInt handle
                              n <- readInt handle
                              print n
                              matrix <- getStateVector handle n n
                              let array = concat matrix
                              return array


generateArraysTest :: Assertion
generateArraysTest = do
  generateArrays 3 3 @?=
   [[6,8,1,7,2,5,3,0,4],[7,0,8,1,4,3,2,5,6],[5,3,2,7,6,8,0,1,4]]
  generateArrays 2 2 @?= [[3,2,1,0],[1,3,0,2]]

formatArrayTest :: Assertion
formatArrayTest = do
  formatArray [1,2,3,4] 2 @?= "1 2\n3 4\n"
  formatArray [1,2,3,4] 4 @?= formatArray [1,2,3,4] 6
```

```
formatArraysTest :: Assertion
formatArraysTest =
  formatArrays [[1,2,3,4],[1,2,3,4,5,6,7,8,9]] @?=  "2\n1 2\n3 4\n3\n1 2
   3\n4 5 6\n7 8 9\n"

manhattanDistTest :: Assertion
manhattanDistTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  manhattanDist x 0 2 @?= 4

hammingDistTest :: Assertion
hammingDistTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  hammingDist x 0 2 @?= 2

getZeroPosTest :: Assertion
getZeroPosTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  getZeroPos x 0 @?= 3
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [3,0,1,2]
  getZeroPos y 0 @?= 1

swapTwoTest :: Assertion
swapTwoTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  computeS (swapTwo 0 3 x) @?= y

getUpNeighborTest :: Assertion
getUpNeighborTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let puzx = PuzzleState 0 0 0 x
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  let puzy = PuzzleState 0 0 3 y
  let res = fromListUnboxed (Z :. (2*2) :: DIM1) [3,0,2,1]
  let puzres = PuzzleState 1 4 1 res

  getUpNeighbor puzx 2 @?= Nothing
  getUpNeighbor puzy 2 @?= Just puzres

getDownNeighborTest :: Assertion
getDownNeighborTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let puzx = PuzzleState 0 0 0 x
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  let puzy = PuzzleState 0 0 3 y
  let res = fromListUnboxed (Z :. (2*2) :: DIM1) [2,1,0,3]
  let puzres = PuzzleState 1 2 2 res

  getDownNeighbor puzx 2 @?= Just puzres
  getDownNeighbor puzy 2 @?= Nothing

getLeftNeighborTest :: Assertion
getLeftNeighborTest = do
```

```
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let puzx = PuzzleState 0 0 0 x
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  let puzy = PuzzleState 0 0 3 y
  let res = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,0,2]
  let puzres = PuzzleState 1 4 2 res

  getLeftNeighbor puzx 2 @?= Nothing
  getLeftNeighbor puzy 2 @?= Just puzres


getRightNeighborTest :: Assertion
getRightNeighborTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let puzx = PuzzleState 0 0 0 x
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  let puzy = PuzzleState 0 0 3 y
  let res = fromListUnboxed (Z :. (2*2) :: DIM1) [1,0,2,3]
  let puzres = PuzzleState 1 2 1 res

  getRightNeighbor puzx 2 @?= Just puzres
  getRightNeighbor puzy 2 @?= Nothing

getAllNeighborTest :: Assertion
getAllNeighborTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let puzx = PuzzleState 0 0 0 x
  let res1 = fromListUnboxed (Z :. (2*2) :: DIM1) [2,1,0,3]
  let puzres1 = PuzzleState 1 2 2 res1
  let res2 = fromListUnboxed (Z :. (2*2) :: DIM1) [1,0,2,3]
  let puzres2 = PuzzleState 1 2 1 res2

  getAllNeighbor puzx 2 @?= [puzres1, puzres2]

getAllNeighborParTest :: Assertion
getAllNeighborParTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let puzx = PuzzleState 0 0 0 x
  let res1 = fromListUnboxed (Z :. (2*2) :: DIM1) [2,1,0,3]
  let puzres1 = PuzzleState 1 2 2 res1
  let res2 = fromListUnboxed (Z :. (2*2) :: DIM1) [1,0,2,3]
  let puzres2 = PuzzleState 1 2 1 res2

  getAllNeighborPar puzx 2 @?= [puzres1, puzres2]

hashTest :: Assertion
hashTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  hash x 0 @?= 3020100
  hash y 0 @?= 20103

getHashKeyTest :: Assertion
getHashKeyTest = do
```

```
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  getHashKey x @?= "3020100"
  getHashKey y @?= "20103"

addMapTest :: Assertion
addMapTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  let z = fromListUnboxed (Z :. (2*2) :: DIM1) [1,0,3,2]
  let puzx = PuzzleState 0 0 0 x
  let puzy = PuzzleState 0 0 0 y
  let puzz = PuzzleState 0 0 0 z
  let mp = H.singleton (getHashKey x) 0
  let resmp = H.fromList [(getHashKey x, 0), (getHashKey y, 0), (getHashKey z, 0)]
  addMap [puzy, puzz] mp @?= resmp

getValidNeighborTest :: Assertion
getValidNeighborTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,2,0]
  let z = fromListUnboxed (Z :. (2*2) :: DIM1) [1,0,3,2]
  let puzx = PuzzleState 1 0 0 x
  let puzy = PuzzleState 1 0 0 y
  let puzz = PuzzleState 1 0 0 z
  let mp = H.singleton (getHashKey x) 0
  getValidNeighbor [puzx, puzy, puzz] mp @?= [puzy, puzz]

numinvTest :: Assertion
numinvTest = do
  let x = fromListUnboxed (Z :. (2*2) :: DIM1) [3,1,0,2]
  numinv x @?= 1
  let y = fromListUnboxed (Z :. (2*2) :: DIM1) [0,1,2,3]
  numinv y @?= 3

solvabilityTest :: Assertion
solvabilityTest = do
  let x = fromListUnboxed (Z :. (3*3) :: DIM1) [1,8,2,0,4,3,7,6,5]
  solvability x 3 3 @?= True
  let y = fromListUnboxed (Z :. (3*3) :: DIM1) [8,1,2,0,4,3,7,6,5]
  solvability y 3 3 @?= False
```

Automatic pipelines

```
for i in 1 2 3 4 5
do
  for name in "ParallelNeighbor" "ParallelPriorityQueue" "Sequential" "ParallelPuzzle"
  do
    time ./app/$name input.txt +RTS -lf -N$i
  done
if [ ! -d "eventlog/n$i/" ]
then
```

```
  mkdir "eventlog/n$i/"
fi
mv *.eventlog "eventlog/n$i/"
done
```

```
stack build
stack exec ghc-pkg unregister libiserv
stack ghc -- -threaded -rtsopts -eventlog app/Main.hs
stack ghc -- -threaded -rtsopts -eventlog -main-is ParallelNeighbor app/
    ParallelNeighbor.hs
stack ghc -- -threaded -rtsopts -eventlog -main-is ParallelPriorityQueue
    app/ParallelPriorityQueue.hs
stack ghc -- -threaded -rtsopts -eventlog -main-is ParallelPuzzle app/
    ParallelPuzzle.hs
stack ghc -- -threaded -rtsopts -eventlog -main-is Sequential app/
    Sequential.hs
```

yaml files

```
# This file was automatically generated by 'stack init'
#
# Some commonly used options have been documented as comments in this file
    .
# For advanced use and comprehensive documentation of the format, please
    see:
# https://docs.haskellstack.org/en/stable/yaml_configuration/

# Resolver to choose a 'specific' stackage snapshot or a compiler version.
# A snapshot resolver dictates the compiler version and the set of
    packages
# to be used for project dependencies. For example:
#
# resolver: lts-3.5
# resolver: nightly-2015-09-21
# resolver: ghc-7.10.2
#
# The location of a snapshot can be provided as a file or url. Stack
    assumes
# a snapshot provided as a file might change, whereas a url resource does
    not.
#
# resolver: ./custom-snapshot.yaml
# resolver: https://example.com/snapshots/2018-01-01.yaml
resolver:
  url: https://raw.githubusercontent.com/commercialhaskell/stackage-
    snapshots/master/lts/18/17.yaml

# User packages to be built.
# Various formats can be used as shown in the example below.
#
# packages:
# - some-directory
# - https://example.com/foo/bar/baz-0.0.2.tar.gz
#   subdirs:
```

```
#   - auto-update
#   - wai
packages:
- .
extra-deps:
- PSQueue-1.1.0.1
- repa-3.4.1.4

# Dependency packages to be pulled from upstream that are not in the
   resolver.
# These entries can reference officially published versions as well as
# forks / in-progress versions pinned to a git hash. For example:
#
# extra-deps:
# - acme-missiles-0.3
# - git: https://github.com/commercialhaskell/stack.git
#   commit: e7b331f14bcffb8367cd58fbfc8b40ec7642100a
#
# extra-deps: []

# Override default flag values for local packages and extra-deps
# flags: {}

# Extra package databases containing global packages
# extra-package-dbs: []

# Control whether we use the GHC we find on the path
# system-ghc: true
#
# Require a specific version of stack, using version ranges
# require-stack-version: -any # Default
# require-stack-version: ">=2.7"
#
# Override the architecture used by stack, especially useful on Windows
# arch: i386
# arch: x86_64
#
# Extra directories used by stack for building
# extra-include-dirs: [/path/to/dir]
# extra-lib-dirs: [/path/to/dir]
#
# Allow a newer minor version of GHC than the snapshot specifies
# compiler-check: newer-minor
```

```
name:                15puzzle
version:             0.1.0.0
github:              "alexunxus/PFP_final_project"
license:             BSD3
author:              "Kuan-Yao Huang, Aditya Sidharta"
maintainer:          "aditya.sdrt@gmail.com"
copyright:           "2021 - Kuan-Yao Huang, Aditya Sidharta"

extra-source-files:
- README.md
```

```
- ChangeLog.md

# Metadata used when publishing your package
# synopsis:            Short description of your package
# category:            Web

# To avoid duplicated efforts in documentation and dealing with the
# complications of embedding Haddock markup inside cabal files, it is
# common to point users to the README.md file.
description:         Please see the README on GitHub at <https://github.
    com/alexunxus/PFP_final_project#readme>

dependencies:
- base >= 4.7 && < 5
- PSQueue
- tasty
- tasty-hunit
- random-shuffle
- random
- unordered-containers
- repa
- parallel

library:
  source-dirs: src

executables:
  15puzzle-exe:
    main:                Main.hs
    source-dirs:         app
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    - -eventlog
    - -Wall
    - -Werror
    dependencies:
    - 15puzzle
    - PSQueue
    - unordered-containers
    - repa
    - parallel

  15puzzle-generate:
    main:                GenFile.hs
    source-dirs:         app
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    - -eventlog
    - -main-is GenFile
    - -Wall
```

```
    - -Werror
    dependencies:
    - 15puzzle
    - random-shuffle
    - random
    - parallel

  sequential-exe:
    main:                Sequential.hs
    source-dirs:         app
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    - -eventlog
    - -main-is Sequential
    - -Wall
    - -Werror
    dependencies:
    - 15puzzle
    - PSQueue
    - unordered-containers
    - repa
    - parallel

  parneighbor-exe:
    main:                ParallelNeighbor.hs
    source-dirs:         app
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    - -eventlog
    - -main-is ParallelNeighbor
    - -Wall
    - -Werror
    dependencies:
    - 15puzzle
    - PSQueue
    - unordered-containers
    - repa
    - parallel

  parpq-exe:
    main:                ParallelPriorityQueue.hs
    source-dirs:         app
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    - -eventlog
    - -main-is ParallelPriorityQueue
    - -Wall
    - -Werror
```

```
    dependencies:
    - 15puzzle
    - PSQueue
    - unordered-containers
    - repa
    - parallel


  parpuzzle-exe:
    main:                ParallelPuzzle.hs
    source-dirs:         app
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    - -eventlog
    - -main-is ParallelPuzzle
    - -Wall
    - -Werror
    dependencies:
    - 15puzzle
    - PSQueue
    - unordered-containers
    - repa
    - parallel

tests:
  15puzzle-test:
    main:                Test.hs
    source-dirs:         test
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    - -Wall
    - -Werror
    dependencies:
    - 15puzzle
    - PSQueue
    - tasty
    - tasty-hunit
    - random-shuffle
    - random
    - unordered-containers
    - repa
    - parallel
```