Nonollel: Parallel Nonogram Solver

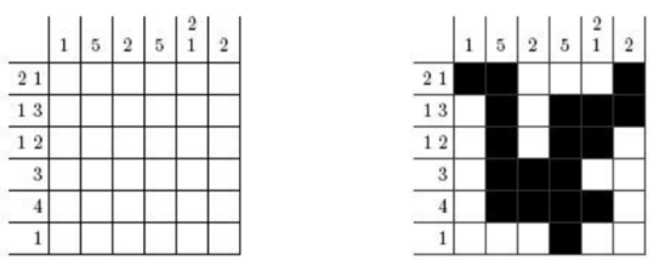
COMS 4495: Parallel Functional Programming

Jason Eriksen (jce2148) & Xurxo Riesco (xr2154)

Dec 21st, 2022

1 Introduction

Nonograms are logic puzzles consisting of an $m \times n$ grid and a set of $m + n$ constraints, each consisting of a sequence of positive numbers denoting the number of consecutive squares that must be colored in either a row or a column. The objective of the puzzle is to color the whole grid while satisfying all constraints. Figure 1 features an example portraying an unsolved and solved version of a puzzle.



(a) 6×6 Nonogram (b) Solved Nonogram

Figure 1: Unsolved and solved Nonogram

2 Approach

There are many possibles approaches to solving a nonogram, given the NP-complete nature of the problem. Our approach consists of an algorithm combining deduction and search, which begins by trying to logically fill in as many initial values as possible in the grid. This is done by using both row and column data and finding the commonality between both sets. This process is repeated until no more deductions can be made. Then, if the puzzle cannot be solved by deduction alone, the algorithm iterates on this partially solved grid by making guesses on the remaining unknown cells in each row and then checking against the column data. This ends once a solution is reached or the data set is deemed unsolvable. Given the aforementioned, the algorithm is best described as a backtracking depth first search prefaced by an iterative deductive step.

3 Implementation

3.1 Sequential Solution

3.1.1 Brute Force Solution

Our first idea was to use the basic *Backtracking Solver* featured in the Haskell Wiki[1]. This approach works by creating a tree of all possible row guesses and then cross referencing against the column data. Although slow, the algorithm was simple and given the tree like nature, it seemed very easy to reach the upper limits of Amdahl's Law by processing branches in parallel. However, the algorithm was struggling to solve even 10×10 puzzles in a reasonable amount of time, even with some basic optimizations we implemented.

3.1.2 Deductive Solution

Given the extremely poor performance and large memory requirements of the *Backtracking Solver*, we opted to use a deductive approach. This solution is heavily based on the *Deducing Solver*, also provided in the Haskell Wiki[1], with modifications mostly concerning the types and some minor performance optimizations. Most noticeably, we chose to introduce a *Cell* type instead of the default *Maybe Bool*. However, we do still preserve the usage of the *Maybe* monad, but not on a per-cell basis.

3.2 Data Types

3.2.1 Cell

The `Cell` type is used to represent each square of the nonogram at a given point in time. There are three possible values for the given type:

- `Filled` – a filled cell
- `Empty` – an empty cell
- `Unknown` – a cell for which a value has not yet been deduced or found

Our implementation relies on the `Cell` type to derive both `Eq` and `Show`.

3.2.2 Row

The `Row` type is used to represent a collection of `Cells` by relying on the native `List` type.

3.2.3 Nonogram

The `Nonogram` type is used to represent a collection of `Rows`. Since the `Nonogram` type is used only to represent the state of the grid and not its constraints, it just needs to concern itself with either rows or columns, and the choice was made arbitrarily. As was the case for the `Row` type, the `Nonogram` type relies on the native `List` type.

3.3 Parallelization

3.3.1 Initial Exploration

To begin, we manually examined the code to identify sections that could be most easily parallelized. Namely, replacing the multiple `map` calls across `transform`, `nonogram`, `solve`, `common` with calls to `parMap` in combination with different strategies. The results in all cases were unsatisfactory causing most of the sparks to be fizzled in the `transform`, `nonogram`, and `solve` cases and most of them to be garbage collected in the case of `common`, but most importantly not achieving speed-ups of above 10-15% when spreading the computation across two cores and measuring with respect to the sequential performance.

3.3.2 Profiling

Considering that our primary concern at this stage was the speed-up of the parallelization, rather than the specifics of the parallelization itself, we opted to take advantage of the profiling options in `stack` to identify the area of the code consuming the bulk of the time. The high level profile overview of the sequential solution was as follows:

```
COST CENTRE           MODULE     %time     %alloc
common                Lib         61.1      58.3
rowsMatchingAux       Lib         21.1      26.8
common.check          Lib          8.8       0.0
rowsMatchingAux.(...) Lib          3.9       7.6
rowsMatching          Lib          2.8       5.0
rowsMatchingAux.l     Lib          0.6       2.2
```

Furthermore, a detailed break-down of the profiling report concerning the `common` function shows that the bulk of the time spent is in function code itself rather than in any of the subcalls. Therefore, it was immediately clear where our efforts would focus. The `common` function, used to find the commonality between all possible ways of placing blocks of a given length in a row consumed the bulk of the time, so we centered our parallelization efforts towards that function. Given that the initial exploration already tried updating the call to `map` with `parMap` fruitlessly, our second attempt surrounded itself with parallelizing the `zipWith` call with our own implementation of its parallel counterpart `parZipWith`:

```haskell
parZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
parZipWith f xs ys = parMap rseq (uncurry f) (zip xs ys)
```

While the results in terms of speed-up were promising than the prior experiments, the preliminary sparks breakdown left a lot of room for improvement:

```
SPARKS: 19569570
(2795085 converted, 11645905 overflowed, 0 dud, 31172 GC'd, 34752 fizzled)
```

Our last option, was to introduce strategy to the computation of:

```haskell
foldr1 (zipWith check) (map (filter isKnownCell) rs)
```

which resulted in the most positive results in terms of speed up and sparks report.

3.3.3 Strategy Exploration

The `rseq` combinator was the obvious choice to balance the overhead of parallel evaluation with the performance benefits, but in terms of evaluation strategies, we explored `parList`, `parListChunk`, and `parBuffer`. We tested each of this on our 30×45 puzzle to allow for a fairly significant amount of spark creation.

Strategy	Converted	Overflowed	Dud	GC'd	Fizzled	Total
parList	10866	0	0	12276	1158	24300
parList(%)	44.7%	0%	0%	50.5%	4.8%	100%
parListChunk	595	0	0	625	400	1620
parListChunk(%)	36.7%	0%	0%	38.6%	24.7%	100%
parBuffer	9900	0	11450	2531	424	24300
parBuffer(%)	40.7%	0%	47.1 %	10.4%	1.7%	100%

This results were obtained using 20 as the parameter for `parListChunk` and 50 as the parameter for `parBuffer`. Since the usage of both those functions varies a lot with the size of the buffer and the size of the chunk, ascertaining an optimal parameter for each would require additional complexity as it would have to be calculated in regard to the length of the rows in the puzzle and passed to the `common` function. Given that we didn't see any performance improvements by either approach that would out weight the performance of `parList` in addition to the added complexity, we made the final choice to use the `parList` evaluation strategy in our implementation.

3.4 Other Implementation Details

Our implementation takes advantage of the Haskell Tool Stack and provides the following targets: `build`, `exec`, `bench`, `test`. The first two are self-explained and detailed in the README, `bench` is further explained in **Section 3.5**.

3.4.1 IO

In order for the solver to work as expected, the input must be given as a filename containing a single list with two elements representing the row and column data, in that order. Each of those lists is comprised of multiple sub-lists, with each element denoting that row or column's cell data. For example, the input corresponding to the Puzzle in Figure 1(a) would be:

```
[[[2,1],[1,3],[1,2],[3],[4],[1]],[[1],[5],[2],[5],[2,1],[2]]]
```

To represent the final state, `Filled` cells are displayed as `#`, and `Empty` cells are displayed as `.`, so the solution portrayed in Figure 1(b) would be:

```
##...#
.#.###
.#.##.
.####.
...#..
```

3.4.2 Testing

Prior to executing a full evaluation of our code, we decided to implement various test leveraging the `HUnit` library to ensure the functional correctness of the implementation. Since the test only concern themselves with correctness, all test results are independent of the number of threads they are executed with. All functions present in the **Lib.hs** have at least a corresponding unit test.

3.4.3 Error Handling

Additionally, our implementation introduces various error checking mechanisms reported via the custom `NonogramException` type to avoid initiating the computation on cases of invalid inputs or erroneous constraints that would deem the puzzle unsolvable prior to the search. The reported exceptions are as follows:

Exception	**Message**
InvalidConstraint	Nonogram constraints are invalid
InvalidNonogram	Nonogram is invalid
InvalidList	Nonogram must have two lists of constraints
ConstraintIsEmpty	Nonogram constraint list is empty
ConstraintHasNegativeValue	Nonogram constraints contain negative values

4 Evaluation

4.1 Settings

Our experiments and performance evaluation was performed on a 2021 16-inch *Macbook Pro* with a 10 core *M1 Pro* CPU and 32GB RAM.

4.2 Initial Testing

4.3 Sequential Results

In order to get a good baseline for our tests, we manually run a series of tests on a variety of different sized puzzles, all ranging around medium to hard difficulty. Our library includes larger puzzles, but for the sake of total benchmark time, we neglected to run any of the larger ones. This program runs each test several times, and ends by providing data about the mean execution time, standard deviation, etc. We use this data, as opposed, to rough approximations with manual runs, to inform our baseline expectations. Below is the data from the sequential evaluation of the 30×40, 30×45, and 40×45 puzzles, respectively.

4.3.1 Parallel Results

To evaluate the performance of our parallelization, we repeated the above process with 2, 3, and 4, and 6 threads in order to gather enough data to form a trend line. As shown below, we achieve maximum speedup when running on 3 threads, with lower speedups for higher thread counts. This is likely due to the sheer amount of overhead required to create and execute the sparks. The amount of code being run by each spark becomes no longer worth the setup time required and the program ends up spending more time garbage collecting or wasting CPU time (fizzling). It's very possible that we could further extend the increases in speedup to higher thread counts with careful management of spark creation via parBuffer or parListChunk (as mentioned above), but this would require a significant amount of testing as not only do these depend on thread count but also on puzzle size and complexity. The additional code complexity would not be worth the likely minimal performance gains, so we opted not to entertain this approach.

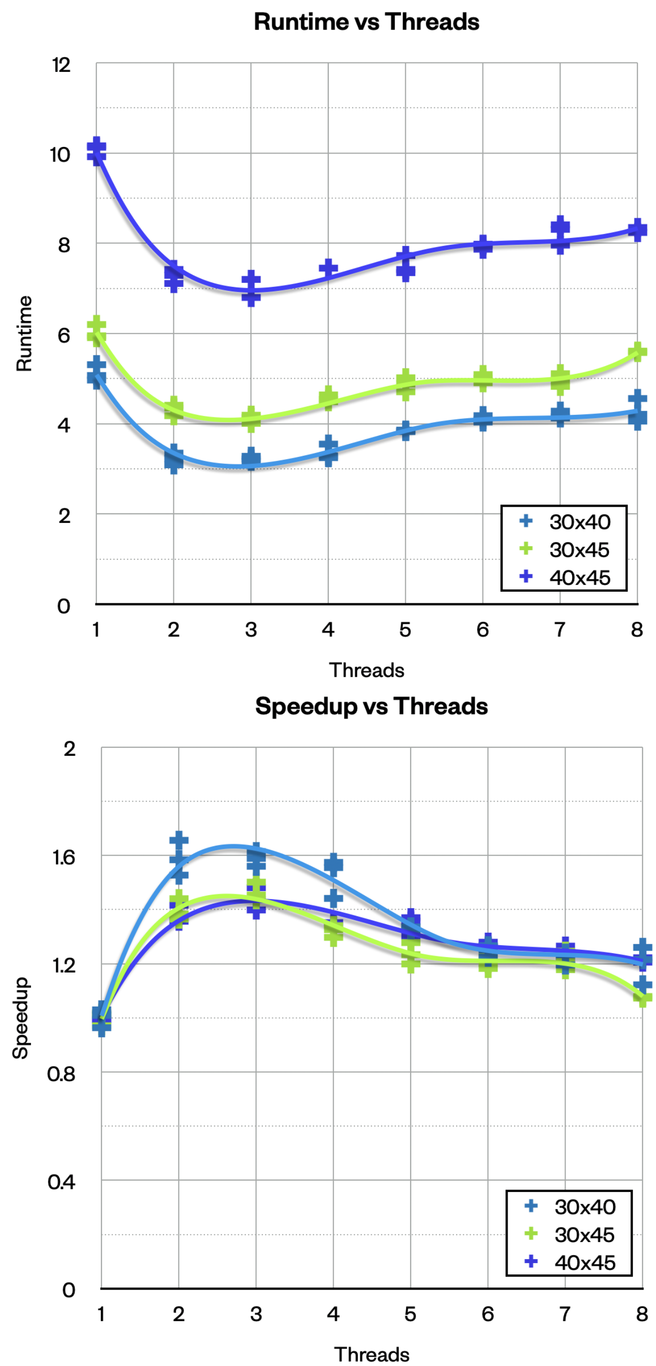


Figure 2: Runtime and speedup vs threads

4.4 Additional Analysis

Despite the 3 thread case being the fastest, the fastest relative speed-up was obtained when comparing the sequential solution to the 2 threaded case. To verify this result we decided to take advantage of `stack bench` to automatize multiple runs of the 10×10, 20×20, 30×40, and 30×45 puzzles. The differences in performance in the 10×10 and the 20×20 puzzles were not statistically significant. The results for the 30×40, and 30×45 are as follows:

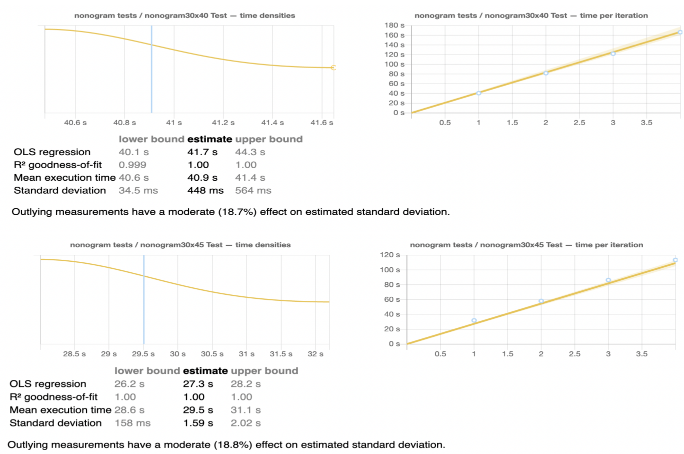


Figure 3: 30x40 and 30x45 sequential benchmarks

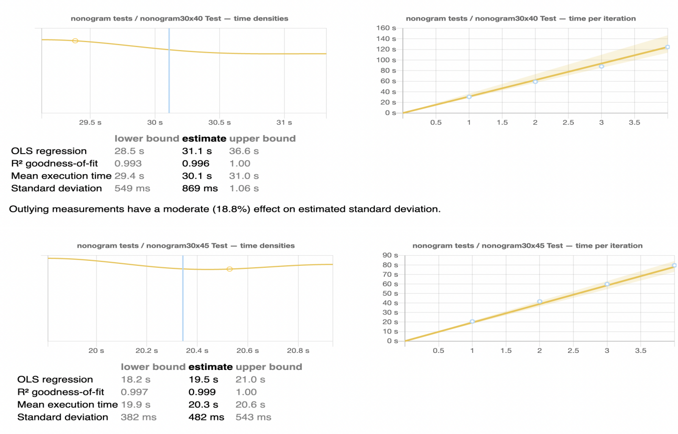


Figure 4: 30x40 and 30x45 parallel (-N2) benchmarks

From this results, it is clear that the complexity and execution type of the puzzle is not uniquely dependent on its size, as the 30×40 puzzle took considerably more time to be solved than the 30×45 puzzle in both settings. Furthermore, we see that speed-up obtained during the bench marking is lower than that obtained during manual testing for the 30×40, 1.34 as opposed to the previous 1.6. While remaining about the same at 1.44 for the 30×45. This gave us a very good indication of what to be expect in the Threadscope visualization as the more plausible explanation from this behavior is that since the 30×40 puzzle is more complex than the 30×45, the memory usage is also higher, slowing the system more when running a higher number of iterations of the solver on a given puzzle.

4.5 Threadscope

As it was expected, the high memory consumption of the program can be seen in the large amount of garbage collection that is executed across threads, this results in less total activity than desired since the garbage collection is interleaved between meaningful execution due to the converging and backtracking nature of the algorithm, for this can be seen in the sequential implementation as well. Despite that, the Threadscope results show various positive outcomes as the computation across the different threads mirrors a very even split and the number of converted sparks dominates across all categories.

4.6 Conclusion

While our speed-up is short of linear in terms of growth, we believe that the bottleneck is the nature of the problem rather than our parallelization strategy. An important lesson to note, which was already presented during the semester, is that parallelization in Haskell requires fine tuning of the evaluation strategies as well as the combinators used in each strategy since the added overhead of sparking parallel computations is quick to outweigh the benefits. The main issue we encountered is that said fine-tuning in the case of Nonograms is very dependent on the input size and the complexity of the problem is highly variable across input constraints, making no parallelization strategy ideal in the general case.

Figure 5: Threadscope visualization for 30x40 (N2 above, N1 below)

5 Code Listings

5.1 app/Main.hs

```
module Main (main) where

import Lib
import System.Environment (getArgs, getProgName)

main :: IO ()
main = do
  args <- getArgs
  case args of
    [filename] -> do
      contents <- readFile filename
      let puzzle = read contents :: [[[Int]]]
      let _ = verifyNonogram puzzle
      let (cs, rs) = (head puzzle, last puzzle)
      print puzzle
      let sol = nonogram rs cs
      putStrLn sol
    _ -> do
      pn <- getProgName
      putStrLn $ "Usage: " ++ pn ++ " filename"
```

5.2 src/Lib.hs

```haskell
module Lib where

import Control.Exception (Exception, throw)
import Control.Monad (zipWithM)
import Control.Parallel.Strategies (parList, rseq, withStrategy)
import Data.List (group, transpose)
import Data.Maybe (maybeToList)

data NonogramException = InvalidConstraint
                       | InvalidNonogram
                       | InvalidList
                       | ConstraintIsEmpty
                       | ConstraintHasNegativeValue

instance Show NonogramException where
  show InvalidConstraint = "Nonogram constraints are invalid"
  show InvalidNonogram = "Nonogram is invalid"
  show InvalidList = "Nonogram must have 2 constraint lists"
  show ConstraintIsEmpty = "Nonogram constraint list is empty"
  show ConstraintHasNegativeValue = "Nonogram constraints contain negative values"

instance Exception NonogramException

instance Eq NonogramException where
  x == y = show x == show y

data Cell = Empty | Filled | Unknown deriving (Eq)

instance Show Cell where
  show Empty = "."
  show Filled = "#"
  show Unknown = "?"

type Row = [Cell]

type Nonogram = [Row]

verifyConstraints :: [[Int]] -> Int -> Bool
verifyConstraints (x : xs) l = verifyConstraint x l && verifyConstraints xs l
  where
    verifyConstraint :: [Int] -> Int -> Bool
    verifyConstraint x' l'
      | null x' = throw ConstraintIsEmpty
```

```haskell
      | any (< 0) x' = throw ConstraintHasNegativeValue
      | (sum x' + length x' - 1) > l' = throw InvalidConstraint
      | otherwise = True
verifyConstraints [] _ = True

verifyNonogram :: [[[Int]]] -> Bool
verifyNonogram puzzle
  | length puzzle /= 2 = throw InvalidList
  | null (head puzzle) || null (last puzzle) = throw InvalidNonogram
  | otherwise = verifyNonogram' (head puzzle) (last puzzle)
  where
    verifyNonogram' :: [[Int]] -> [[Int]] -> Bool
    verifyNonogram' rows cols
      | verifyConstraints rows (length rows)
        && verifyConstraints cols (length cols)
      | otherwise = throw InvalidNonogram

nonogram :: [[Int]] -> [[Int]] -> String
nonogram rows columns = case solve rows columns of
  [] -> "Unsolvable\n"
  (grid : _) ->
    unlines
      . map (concatMap show)
      . transpose
      $ grid

transform :: [Cell] -> [Int]
transform =
  map length
    . filter
      ( \y -> case y of
          [] -> False
          (x : _) -> x == Filled
      )
    . group

solve :: [[Int]] -> [[Int]] -> [Nonogram]
solve rs cs = do
  grid <- maybeToList (deduce rs cs)
  grid' <- zipWithM (rowsMatching nc) rs grid
  if map transform (transpose grid') == cs
    then return grid'
    else []
  where
    nc = length cs
```

```haskell

deduce :: [[Int]] -> [[Int]] -> Maybe Nonogram
deduce rs cs = converge step initial
  where
    nr = length rs
    nc = length cs
    initial = replicate nr (replicate nc Unknown)
    step = (improve nc rs . transpose) <.> (improve nr cs . transpose)
    improve n = zipWithM (common n)
    (g <.> f) x = f x >>= g

converge :: (Nonogram -> Maybe Nonogram) -> Nonogram -> Maybe Nonogram
converge f s = do
  s' <- f s
  if s == s' || not (any (elem Unknown) s')
  then return s'
  else converge f s'

isKnownCell :: Cell -> Bool
isKnownCell Unknown = False
isKnownCell _ = True

common :: Int -> [Int] -> Row -> Maybe Row
common n ks partial = case rowsMatching n ks partial of
  [] -> Nothing
  rs -> Just $ withStrategy (parList rseq)
        (foldr1 (zipWith check) (map (filter isKnownCell) rs))
  where
    check :: Cell -> Cell -> Cell
    check x y
      | x == y = x
      | otherwise = Unknown

rowsMatching :: Int -> [Int] -> Row -> [Row]
rowsMatching _ [] [] = [[]]
rowsMatching _ _ [] = []
rowsMatching n ks (Unknown : partial) =
  rowsMatchingAux n ks Filled partial
    ++ rowsMatchingAux n ks Empty partial
rowsMatching n ks (s : partial) =
  rowsMatchingAux n ks s partial

rowsMatchingAux :: Int -> [Int] -> Cell -> Row -> [Row]
rowsMatchingAux _ [] _ _ = [[]]
rowsMatchingAux _ _ Unknown _ = [[]]
```

```
rowsMatchingAux n ks Empty partial =
  [Empty : row | row <- rowsMatching (n - 1) ks partial]
rowsMatchingAux n [k] Filled partial =
  [ replicate k Filled ++ replicate (n - k) Empty
    | n >= k && notElem Empty front && notElem Filled back
  ]
  where
    (front, back) = splitAt (k - 1) partial
rowsMatchingAux n (k : ks) Filled partial =
  [ replicate k Filled ++ Empty : row
    | n > k + 1 && notElem Empty front && blank /= Filled,
      row <- rowsMatching (n - k - 1) ks partial'
  ]
  where
    l = splitAt (k - 1) partial
    front = fst l
    blank = head (snd l)
    partial' = tail (snd l)
```

5.3 benchmark/Main.hs

```haskell
module Main where

import Criterion
import Criterion.Main (defaultMain)
import Lib

main :: IO ()
main = do
  nonogram10x10 <- readFile "puzzles/10x10.txt"
  nonogram20x20 <- readFile "puzzles/20x20.txt"
  nonogram30x40 <- readFile "puzzles/30x40.txt"
  nonogram30x45 <- readFile "puzzles/30x45.txt"
  let puzzle10x10 = read nonogram10x10 :: [[[Int]]]
  let puzzle20x20 = read nonogram20x20 :: [[[Int]]]
  let puzzle30x40 = read nonogram30x40 :: [[[Int]]]
  let puzzle30x45 = read nonogram30x45 :: [[[Int]]]
  let (rows10x10, cols10x10) = (head puzzle10x10, last puzzle10x10)
  let (rows20x20, cols20x20) = (head puzzle20x20, last puzzle20x20)
  let (rows30x40, cols30x40) = (head puzzle30x40, last puzzle30x40)
  let (rows30x45, cols30x45) = (head puzzle30x45, last puzzle30x45)
  defaultMain
    [ bgroup
        "nonogram tests"
        [ bench "nonogram10x10 Test" $ whnf (nonogram rows10x10) cols10x10,
          bench "nonogram20x20 Test" $ whnf (nonogram rows20x20) cols20x20,
          bench "nonogram30x40 Test" $ whnf (nonogram rows30x40) cols30x40,
          bench "nonogram30x45 Test" $ whnf (nonogram rows30x45) cols30x45,
        ]
    ]
```

5.4 test/Spec.hs

```haskell
module Main where

import Control.Exception
import Control.Monad
import Lib
import Test.HUnit.Base
import Test.HUnit.Text (runTestTT)

assertException :: (Exception e, Eq e) => e -> IO a -> IO ()
assertException ex action =
  handleJust isWanted (const $ return ()) $ do
    _ <- action
    assertFailure $ "Expected exception: " ++ show ex
  where
    isWanted = guard . (== ex)

tests :: Test
tests =
  TestList
    [ TestLabel "testVerifyNonogram" testVerifyNonogram,
      TestLabel "testVerifyNonogram'" testVerifyNonogram',
      TestLabel "testVerifyConstraints" testVerifyConstraints,
      TestLabel "testVerifyConstraints'" testVerifyConstraints',
      TestLabel "testVerifyConstraints''" testVerifyConstraints'',
      TestLabel "testVerifyConstraints'''" testVerifyConstraints''',
      TestLabel "testTransform" testTransform,
      TestLabel "testTransform'" testTransform',
      TestLabel "testSolve" testSolve,
      TestLabel "testSolve'" testSolve',
      TestLabel "testisKnownCell" testisKnownCell,
      TestLabel "testisKnownCell'" testisKnownCell',
      TestLabel "testisKnownCell''" testisKnownCell'',
      TestLabel "testDeduce" testDeduce,
      TestLabel "testDeduce'" testDeduce',
      TestLabel "testDeduce''" testDeduce'',
      TestLabel "testConverge" testConverge,
      TestLabel "testConverge'" testConverge',
      TestLabel "testConverge''" testConverge'',
      TestLabel "testCommon" testCommon,
      TestLabel "testCommon'" testCommon',
      TestLabel "testCommon''" testCommon'',
      TestLabel "testCommon'''" testCommon''',
      TestLabel "testCommon''''" testCommon'''',
```

```
      TestLabel "testCommon'''''" testCommon''''',
      TestLabel "testCommon''''''" testCommon''''''
    ]

{- Valid Nonogram returns True -}
testVerifyNonogram :: Test
testVerifyNonogram = TestCase $ assertEqual "verifyNonogram" True
                    (verifyNonogram [[[1,1],[2],[3]],[[1,1],[1,1],[1]]])

{- InvalidList raises -}
testVerifyNonogram' :: Test
testVerifyNonogram' = TestCase $ assertException InvalidList (evaluate
                    (verifyNonogram [[[4],[2],[3]]]) :: IO Bool)

{- Valid constraints return True -}
testVerifyConstraints :: Test
testVerifyConstraints = TestCase $ assertEqual "verifyConstraints" True
                    (verifyConstraints [[1, 1],[2],[3]] 6)

{- InvalidConstraint raises -}
testVerifyConstraints' :: Test
testVerifyConstraints' = TestCase $ assertException InvalidConstraint (evaluate
                    (verifyConstraints [[1,1],[2],[3]] 2) :: IO Bool)

{- ConstraintIsEmpty raises -}
testVerifyConstraints'' :: Test
testVerifyConstraints'' = TestCase $ assertException ConstraintIsEmpty (evaluate
                    (verifyConstraints [[]] 2) :: IO Bool)

{- ConstraintHasNegativeValue raises -}
testVerifyConstraints''' :: Test
testVerifyConstraints''' = TestCase $ assertException ConstraintHasNegativeValue
                    (evaluate (verifyConstraints [[-1]] 2) :: IO Bool)

{- Transform works for multi element list -}
testTransform :: Test
testTransform = TestCase $ assertEqual "transform"
                [1, 1] (transform [Filled, Empty, Filled])

{- Transform works for single element list -}
testTransform' :: Test
testTransform' = TestCase $ assertEqual "transform'"
                [1] (transform [Filled, Empty])


```

```haskell
{- Solve works for solvable nonogram -}
testSolve :: Test
testSolve = TestCase $ assertEqual "solve"
            [[[Filled, Filled, Empty, Empty, Empty],
              [Filled, Filled, Filled, Empty, Empty],
              [Empty, Empty, Empty, Filled, Filled],
              [Empty, Filled, Filled, Filled, Empty],
              [Filled, Empty, Filled, Filled, Empty]]]
            (solve [[2],[3],[2],[3],[1, 2]] [[2,1],[2,1],[1,2],[3],[1]])

{- Solve returns [] for unsolvable nonogram -}
testSolve' :: Test
testSolve' = TestCase $ assertEqual "solve" [] (solve [[1], [4]] [[2]])

{- Filled is a knonw cell -}
testisKnownCell :: Test
testisKnownCell = TestCase $ assertEqual "isKnownCell" True (isKnownCell Filled)

{- Empty is a knonw cell -}
testisKnownCell' :: Test
testisKnownCell' = TestCase $ assertEqual "isKnownCell'" True (isKnownCell Empty)

{- Empty is a knonw cell -}
testisKnownCell'' :: Test
testisKnownCell'' = TestCase $ assertEqual "isKnownCell'" False (isKnownCell Unknown

{- Deduce row works for empty rows and cols -}
testDeduce :: Test
testDeduce = TestCase $ assertEqual "deduce" (Just []) (deduce [] [])

{- Deduce row works for fully deducible Nonogram -}
testDeduce' :: Test
testDeduce' = TestCase $ assertEqual "deduce'" (Just [[Filled]])
              (deduce [[1]] [[1]])

{- Deduce row works for unsolvable Nonogram -}
testDeduce'' :: Test
testDeduce'' = TestCase $ assertEqual "deduce''" Nothing
               (deduce [[1, 1], [1, 1]] [[1, 2], [2, 1]])

{- f is const and returns Nothing -}
testConverge :: Test
testConverge = TestCase $ assertEqual "converge" Nothing (converge
               (const Nothing) [[Filled, Empty], [Empty, Filled]])
```

```haskell
{- f is const and returns something -}
testConverge' :: Test
testConverge' = TestCase $ assertEqual "converge'"
                 (Just [[Filled, Empty], [Empty, Filled]])
                 (converge (const (Just [[Filled, Empty], [Empty, Filled]]))
                 [[Filled, Empty], [Empty, Filled]])

{- f is const and modifies the input -}
testConverge'' :: Test
testConverge'' = TestCase $ assertEqual "converge''"
                  (Just [[Empty, Filled], [Filled, Empty]])
                  (converge (\x -> if x == [[Filled, Empty], [Empty, Filled]]
                                   then Just [[Empty, Filled], [Filled, Empty]]
                                   else Just x)
                  [[Filled, Empty], [Empty, Filled]])

{- n=0 ks=[] -}
testCommon :: Test
testCommon = TestCase $ assertEqual "common" (Just []) (common 0 [] [])

{- n>0 ks=[] -}
testCommon' :: Test
testCommon' = TestCase $ assertEqual "common'" (Just [])
               (common 3 [] [Filled, Empty, Unknown])

{- n=0 ks=[_] -}
testCommon'' :: Test
testCommon'' = TestCase $ assertEqual "common''" Nothing (common 0 [1] [])

{- n>0 len(ks) > len(partial) -}
testCommon''' :: Test
testCommon''' = TestCase $ assertEqual "common'''" Nothing
                 (common 3 [1, 2] [Filled, Empty])

{- n>0 len(ks) < len(partial) -}
testCommon'''' :: Test
testCommon'''' = TestCase $ assertEqual "common''''"
                  (Just [Filled, Empty, Empty])
                  (common 3 [1] [Filled, Empty, Unknown])

{- n>0 len(ks) == len(partial) partial is known -}
testCommon''''' :: Test
testCommon''''' = TestCase $ assertEqual "common'''''"
                   (Just [Filled, Empty, Filled])
                   (common 3 [1, 1] [Filled, Empty, Filled])
```

```
{- n>0 len(ks) == len(partial) partial is not known -}
testCommon'''''' :: Test
testCommon'''''' = TestCase $ assertEqual "common''''''"
                        (Just [Filled, Empty, Filled])
                        (common 3 [1, 1] [Filled, Unknown, Filled])

main :: IO Counts
main = runTestTT tests
```

6 References

References

[1] https://wiki.haskell.org/Nonogram

[2] https://stackoverflow.com/questions/13350164/how-do-i-test-for-an-error-in-haskell

[3] https://mmhaskell.com/testing/profiling

[4] http://www.cs.columbia.edu/s̃edwards/classes/2019/4995-fall/reports/pagerank.pdf