Nonogram Solver

Dorothy Nelson (dpn2111), Jittisa (Jane) Kraprayoon (jjk2239)

1. Introduction

1.1 Background:

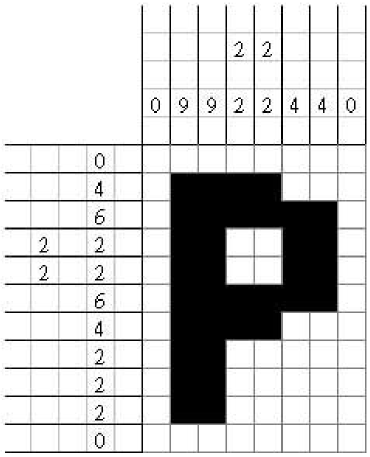

Figure 1. Example nonogram puzzle

In this project, we seek to parallelize a nonogram solver in Haskell. A nonogram is a visual puzzle where, given a grid of white cells, row constraints, and column constraints, players must fill out each cell to construct a picture. Row and column constraints are given for each row and column. "Blocks" are consecutively filled cells. Each constraint describes the number of blocks within a row or column and the lengths of each. From these constraints, players must iteratively infer which cells must be filled. Moreover, each block must be separated by at least one white or unfilled cell. Therefore, a row constraint like "3 2" describes a length 3 block, separated by at least one white cell, and followed by a length 2 block.

1.2 Challenges:

The nonogram becomes challenging with difficult clues. For example, a row with less cells and with larger blocks is easier to solve. Conversely a row that is mostly composed of white cells introduces more ambiguity. Moreover, Puzzles designed for humans generally have one solution and reveal an image. However, it is also possible for Nonograms to have multiple solutions with no discernible pictures.

The nonogram also becomes more computationally complex with large grid sizes. As the number of rows and columns increases, the search space grows exponentially as there are more possibilities to explore. Nonograms have also been shown to be NP-hard and thus there are a multitude of possible approaches to solving one that balance correctness with computational complexity.

---

2. Implementation

2.1 Nonogram Representation:

In our project, each nonogram is represented with key attributes of `height`, `width`, `rowArgs` and `colArgs`. `height` and `width` refer to the size of the grid of cells, while `rowArgs` and `colArgs` are lists of lists containing the constraints for each row and column. These are given as inputs to the algorithm to start solving the nonogram.

2.2 Data Collection and Parsing

Nonograms were taken from https://github.com/mikix/nonogram-db. We stored each nonogram from this database as a .txt file and used our `parseNonogram` function to extract rowArgs and colArgs (constraints) to be inputted into our solver.

2.3 Base Algorithm Overview:

Our nonogram solver base algorithm can be described in three parts: 1) constraint satisfaction, 2) iterative inference, and 3) backtracking for unresolved cases.

1) Constraint Satisfaction

The algorithm first iterates through each row and column constraint. For each, it comes up with possible placements of blocks. `cmputeBlocksSeq :: Int -> [Int] -> [[Int]]` operates on a line and takes the line length and constraints (block lengths) as input and recursively tries to place blocks into different start positions and continues with the remaining blocks. The output is a list of lists where each inner list contains possible start positions for the blocks within that line. For example, `lineLength = 7` and `lineConstraint = [2, 3]` would yield an output of `[[0, 3], [1, 4]]` from `computeBlocksSeq`.

```haskell
computeBlocksSeq :: Int -> [Int] -> [[Int]]
computeBlocksSeq lineLength blockLengths = placeBlocks blockLengths 0
 where
   -- Recursive helper function to place blocks
   placeBlocks :: [Int] -> Int -> [[Int]]
   placeBlocks [] _ = [[]] -- No blocks left to place
   placeBlocks (b:bs) start
     | start + remainingLength > lineLength = []
     | otherwise = do
         pos <- [start .. lineLength - remainingLength]
```

```
        rest <- placeBlocks bs (pos + b + 1) -- Recur with updated start position
        return (pos : rest)
    where
      -- Calculate remaining length
      remainingLength = sum (b : bs) + length bs
```

`generateBlocksSeq` `:: [[Int]] -> [Int] -> Int -> [[Int]]` takes the output of computeBlocks—the potential starting positions of the blocks—and generates a binary array (Ints of 1s and 0s) to represent possible line configurations.

```
generateBlocksSeq :: [[Int]] -> [Int] -> Int -> [[Int]]
generateBlocksSeq blockStarts blockSizes totalLength =
   map (generateBinaryArray blockSizes totalLength) blockStarts
 where
   generateBinaryArray :: [Int] -> Int -> [Int] -> [Int]
   generateBinaryArray sizes len starts = foldl placeBlock (replicate len 0) (zip starts
sizes)

   placeBlock :: [Int] -> (Int, Int) -> [Int]
   placeBlock arr (start, size) =
       take start arr ++ replicate size 1 ++ drop (start + size) arr
```

2) Iterative inference

With the possible line configurations, we then move on to the inference step. If in all possible configurations of a line, a cell is filled, then we know that that cell must be black. Similarly, if in all possible configurations, a cell is unfilled, then we know that that cell is certainly white. The function inferValues performs this step and is described as follows: **`inferValuesSeq`** `:: PartialSolution -> (Array Int (Set.Set [Int]), Array Int (Set.Set [Int])) -> PartialSolution`. A PartialSolution is a grid representing the current progress of the puzzle. This array contains values 0, 1, and -1. 0 represents white cells, 1 represents black cells, and -1 represents cells that are still unknown. **`inferValuesSeq`** is called repeatedly until the nonogram is solved.

The main logic in **`inferValuesSeq`** is in the following code from the function, and also in a helper function **`inferRowOrCol`**:

```
let rowOnes = foldl1 (zipWith (.&.)) placements
    rowZeros = foldl1 (zipWith (.|.)) placements
    inferredRow = inferRowOrCol rowOnes rowZeros
in [((r, c), inferredRow !! c) | c <- [0..numCols - 1], partialSolution ! (r, c) == -1]

-- Infer the result for a row or column based on bitwise results
inferRowOrCol :: [Int] -> [Int] -> [Int]
```

```
inferRowOrCol ones zeros =
   zipWith resolveCell ones zeros
 where
   resolveCell 0 1 = -1   -- The cell must be unknown
   resolveCell 1 1 = 1    -- The cell must be filled (from ones)
   resolveCell 0 0 = 0    -- The cell must be empty (from zeros)
   resolveCell _ _ = -1   -- Fallback for unexpected values
```

We perform bitwise operations to extract indices where it is 1s or 0s across all possible placements for that line.

3) Backtracking

If the nonogram puzzle is ambiguous—meaning that there are multiple solutions and options which cannot be deduced by the iterative solver alone—then we use backtracking to solve the rest of the puzzle. Backtracking works by trying different placements within a row and checking if this step still results in a valid grid. If the grid is still valid, backtrack is called recursively until we reach the basecase of all the rows being completed. In our implementation, backtrack is defined as follows:

```
backtrack :: PartialSolution
         -> Array Int (Set [Int])  -- Row placements
         -> [Constraint]           -- Row constraints
         -> [Constraint]           -- Column constraints
         -> Set Int                -- Completed rows
         -> Set Int                -- Completed columns
         -> [PartialSolution]      -- Accumulated solutions
         -> [PartialSolution]      -- Final list of valid solutions
```

---

3. Sequential Algorithm Benchmark

		Total Time Elapsed (s)
Small	Bloop (10x10)	0.01
Medium	Ubuntu (35x35)	2.43
	42 (23x35)	0.18
Large	Wikimedia (38x39)	1.75
	7 medium puzzles solved sequentially	36.77

Table 1. Sequential algorithm benchmark with puzzles of different sizes

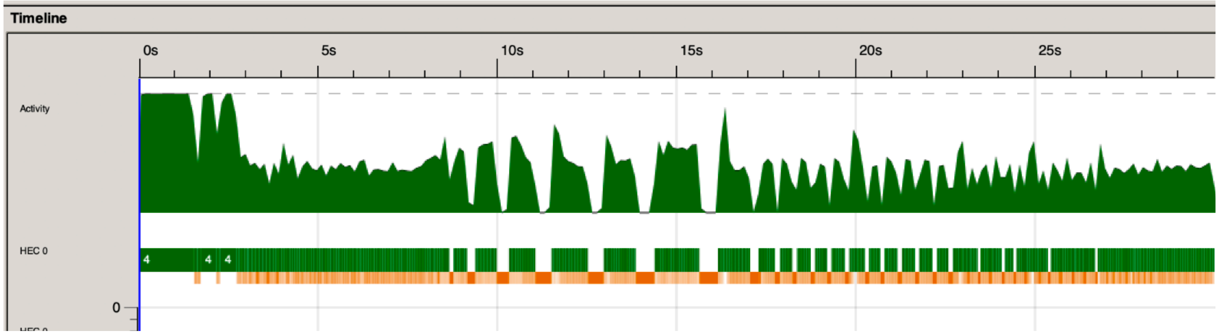

Figure 3. Activity graph for sequential algorithm with 7 puzzles

We first tested our sequential algorithm on puzzles of varying difficulty. As expected, smaller puzzles were solved very quickly (0.01s), and larger puzzles took more time as there is a larger search space for the algorithm to go through. As one puzzle may be too little work, we decided on the benchmark task being to solve 7 medium puzzles. Larger puzzles with sizes 60x70 were also considered and tested, however the terminal would crash with these sizes, so we settled on testing the algorithm on multiple puzzles instead.

4. Parallelization Strategy

Our motivation for parallelization came from the fact that row and column processing can be done independently. For instance, computing the starting placements based on the constraints for one row is not affected by the result of another row. The `Control.Parallel.Strategies` module was used for parallelization and our attempts target separate functions that contribute to the main algorithm.

4.1 Parallelization Attempt 1: `inferValuesPar`

We first attempted to parallelize **`inferValues`** as each inference for each line is done independently. Specifically, **`parMap`** and **`rseq`** were used to allow multiple rows to be processed simultaneously with **`updateRows`**. inferValues is mainly used in iterativeSolve which calls the function repeatedly. The iterativeSolve function is inherently sequential, as the partial solution grid must be updated before another call can be made in the next step. Because of this, we do not expect significant gains from this parallelization strategy. Below is the line of code where parallelization occurs.

```
updatedRows = partialSolution // concat (parMap rseq updateRow [r1..r2])
```

4.2 Parallelization Attempt 2: `computeBlocksPar`

The second parallelization attempt involves parallelizing computeBlocks—extracting valid starting positions for each block from the constraints. To be able to control the granularity of the task, we chunked multiple cells to be processed together using a helper function chunkList. We noticed that computeBlocksPar benefits from doing this. processChunk applies processPosition which computes the valid placements for a single starting position. parMap and rdeepseq were used to ensure parallelization across threads.

```
computeBlocksPar lineLength blockLengths =
   concat $ parMap rdeepseq processChunk (chunkList chunkSize [0 .. lineLength -
totalRemainingLength])
```

4.3 Parallelization Attempt 3: `generateBlocksPar`

generateBlocksPar uses the same method from computeBlocksPar for parallelization. It groups the cells in chunks using chunkList. processChunk then applies generateBinaryArray which performs the main task of placing the blocks in the binary array. parMap and rdeepseq were used for parallelization.

```
generateBlocksPar blockStarts blockSizes totalLength =
   concat $ parMap rdeepseq processChunk (chunkList chunkSize blockStarts)
```

4.4 Parallelization Summary

With these attempts, we accumulate two different versions (sequential and parallel) of each inferValue, computeBlock, and generateBlocks. For each, you can choose whether to use the sequential version or the parallelized version. We test different combinations of the three functions. For example `solveParallelComputeGenerate` uses both the parallelized version of computeBlocks and generateBlocks, while `solveFullyParallel` uses the parallelized version of computeBlocks, generateBlocks, and inferValues. In total, there are 6 versions of the algorithm:

```
solveSequential :: FilePath -> IO ()
solveSequential = solveNonogramFromFile computeBlocksSeq generateBlocksSeq iterativeSolveSeq

solveParallelComputeBlocks :: FilePath -> IO ()
solveParallelComputeBlocks = solveNonogramFromFile computeBlocksPar generateBlocksSeq iterativeSolveSeq

solveParallelGenerateBlocks :: FilePath -> IO ()
solveParallelGenerateBlocks = solveNonogramFromFile computeBlocksSeq generateBlocksPar iterativeSolveSeq

solveParallelComputeGenerate :: FilePath -> IO ()
solveParallelComputeGenerate = solveNonogramFromFile computeBlocksPar generateBlocksPar
iterativeSolveSeq
```

```
solveParallelIterativeSolve :: FilePath -> IO ()
solveParallelIterativeSolve = solveNonogramFromFile computeBlocksSeq generateBlocksSeq iterativeSolvePar

solveFullyParallel :: FilePath -> IO ()
solveFullyParallel = solveNonogramFromFile computeBlocksPar generateBlocksPar iterativeSolvePar
```

---

4. Results

	Max Speedup
Sequential	1.00
Parallel Compute	1.40
Parallel Generate	1.88
Parallel Compute Generate	1.50
Parallel Iterative Solve	1.71
Fully Parallel	1.50

Table 2. Speedup graph with different algorithms

The table above shows the speedup, which was calculated as (Sequential benchmark) / (shortest time elapsed for algorithm). Parallel Generate performed the best with a x1.88 speed up from the original benchmark, followed by Parallel Iterative Solve with a x1.50 improvement. Fully Parallel and Parallel Compute Generate did not perform as well. This shows that combining multiple parallelized functions is detrimental to speed up time, and causes more overhead computations than it is worth.

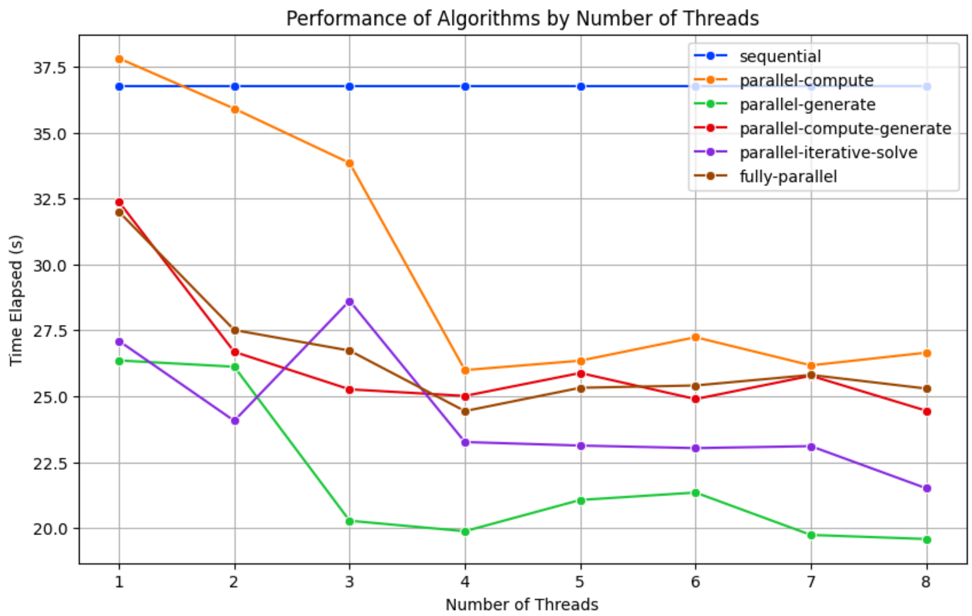

Figure 4. Total time elapsed vs. number of threads

The graph above shows the total time elapsed with increasing number of threads. For all parallelized versions of the algorithm, the time elapsed decreased, showing utilization of the threads. However, all of the algorithms also level out, showing diminishing returns from an excessive number of threads.

5. Improvements

5.1 Limitation from Iterative Solve

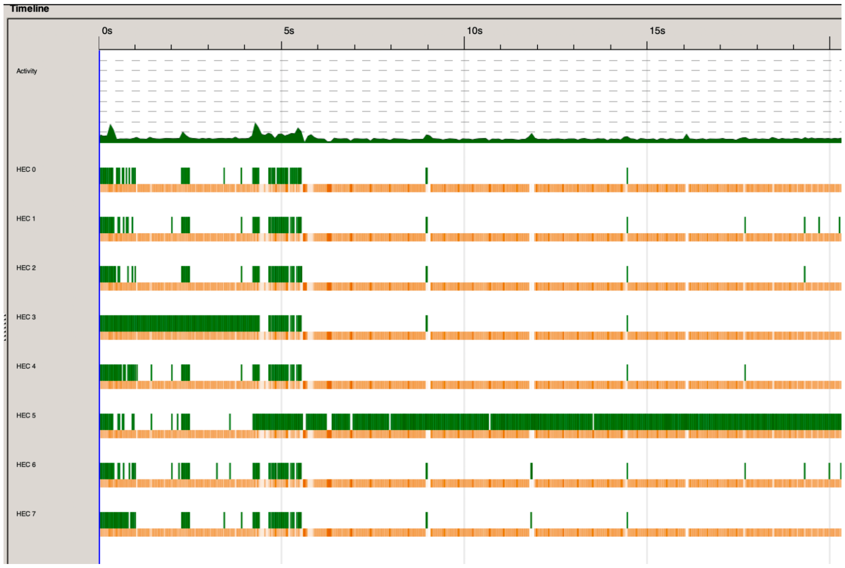

Figure 5. Activity graph for Fully Parallel -N8

As shown in the activity graph (Figure 5), we struggled to parallelize the main process in the algorithm — iterativeSolve, which repeatedly infers values until the nonogram is solved. This is because the nature of iterativeSolve is sequential, where the next step depends on the result of the previous. However, the activity graph does show somewhat successful parallelization when doing computeBlocksPar and generateBlocksPar.

As the puzzles were not ambiguous, backtracking did not end up being used. For the next steps, however, backtracking part of the algorithm would likely gain more from parallelization as multiple placements can be explored concurrently. We can focus on solving smaller nonograms, but mainly through backtracking.

5.2 Garbage Collection

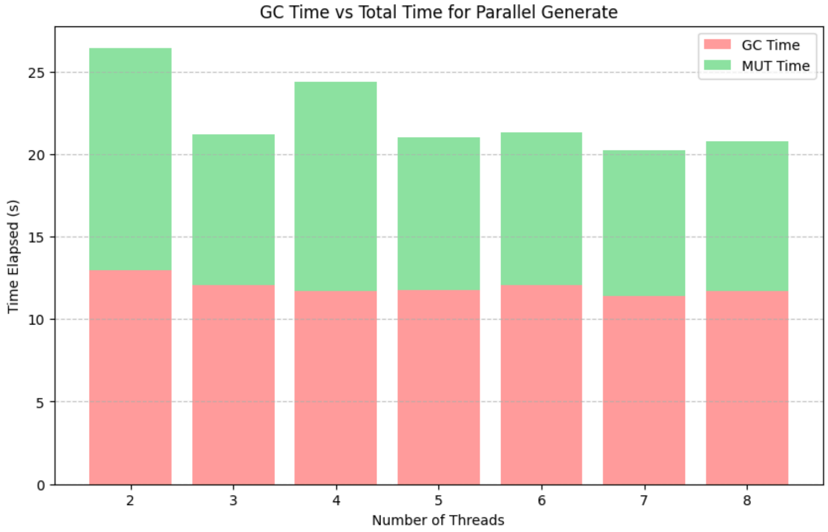

Figure 6. GC Time vs. MUT Time for Parallel Generate -N8

We also noticed that garbage collection takes up as much time as mutator operations (Figure 6). This means that a significant portion of the time is spent on managing memory rather than doing useful work. A future solution could be to try to reduce GC time, perhaps using ParBuffer, which could help limit memory usage.

6. References

- https://github.com/mikix/nonogram-db
- https://github.com/Arpanio/nonogram
- https://towardsdatascience.com/solving-nonograms-with-120-lines-of-code-a7c6e0f627e4

7. Code

7.1 Main.hs

```haskell
import NonogramSolverPar (solveSequential, solveParallelComputeBlocks,
solveParallelGenerateBlocks, solveParallelComputeGenerate,
solveParallelIterativeSolve, solveFullyParallel)
import System.Environment (getArgs)

main :: IO ()
main = do
   putStrLn "Solving Nonogram Puzzle..."

   args <- getArgs
   case args of
       [mode, filePath] ->
           case mode of
               "--sequential" -> solveSequential filePath
               "--parallel-compute" -> solveParallelComputeBlocks filePath
               "--parallel-generate" -> solveParallelGenerateBlocks filePath
               "--parallel-compute-generate" -> solveParallelComputeGenerate filePath
               "--parallel-iterative-solve" -> solveParallelIterativeSolve filePath
               "--fully-parallel" -> solveFullyParallel filePath
               _ -> putStrLn "Invalid mode. Usage: ./nonogramSolver [--sequential |
--parallel-compute | --parallel-generate | --parallel-compute-generate |
--parallel-iterative-solve | --fully-parallel] <file-path>"
       _ -> putStrLn "Usage: ./nonogramSolver [--sequential | --parallel-compute |
--parallel-generate | --parallel-compute-generate | --parallel-iterative-solve |
--fully-parallel] <file-path>"

   putStrLn "Puzzle Solved!"
```

7.2 Parser.hs

```haskell
module Parser (parseNonogram) where

import NonogramTypes
import Data.List (isPrefixOf, uncons)
```

```haskell
import Data.List.Split (splitOn)
import Data.Maybe (mapMaybe)


-- Parse
parseNonogram :: FilePath -> IO Nonogram
parseNonogram path = do
 content <- lines <$> readFile path
 let titleLine   = extractValue "title" content
     heightLine  = read (extractValue "height" content) :: Int
     widthLine   = read (extractValue "width" content) :: Int
     rowsSection = extractSection "rows" "columns" content
     colsSection = extractSection "columns" "goal" content
     goalSection = extractValue "goal" content
     rowsHints   = parseHints rowsSection
     colsHints   = parseHints colsSection
 return Nonogram { title = titleLine, height = heightLine, width = widthLine, rows =
rowsHints, columns = colsHints, goal = goalSection }


-- Extract value from key in file
extractValue :: String -> [String] -> String
extractValue key allLines =
 let matchingLines = [line | line <- allLines, key `isPrefixOf` line]
 in case uncons matchingLines of
      Just (matchingLine, _) -> unquote $ drop (length key + 1) matchingLine
      Nothing -> error $ "Key not found: " ++ key -- empty case
 where
   unquote s = filter (`notElem` "\"") s -- no quotes


-- Extract sections
extractSection :: String -> String -> [String] -> [String]
extractSection startKey endKey allLines =
 takeWhile (not . isPrefixOf endKey) . drop 1 . dropWhile (not . isPrefixOf startKey)
$ allLines

-- Parse hints into list of lists of integers
parseHints :: [String] -> [[Int]]
parseHints = mapMaybe safeParseLine

-- Safe parsing of individual lines
```

```
safeParseLine :: String -> Maybe [Int]
safeParseLine line =
 if null line then Nothing
 else Just (map read $ splitOn "," line)
```

7.3 NonogramTypes.hs

```
module NonogramTypes (Nonogram(..)) where


-- Data type for representing a Nonogram puzzle
data Nonogram = Nonogram
 { title   :: String         -- Title of puzzle
 , height  :: Int            -- Height of puzzle grid
 , width   :: Int            -- Width of puzzle grid
 , rows    :: [[Int]]        -- Row constraints (lists of integers)
 , columns :: [[Int]]        -- Column constraints (lists of integers)
 , goal    :: String         -- (Optional) Goal or solution representation as a string
 } deriving (Show)
```

7.4 NonogramSolverPar.hs

```
module NonogramSolverPar (solveSequential, solveParallelComputeBlocks,
solveParallelGenerateBlocks, solveParallelComputeGenerate,
solveParallelIterativeSolve, solveFullyParallel) where

import Data.Array
import Data.List (group, transpose, foldl')
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Bits ((.&.), (.|.))
import Parser (parseNonogram)
import NonogramTypes

import Control.Parallel.Strategies (parMap, rdeepseq, rseq)

computeBlocksSeq :: Int -> [Int] -> [[Int]]
computeBlocksSeq lineLength blockLengths = placeBlocks blockLengths 0
```

```
 where
   -- place blocks
   placeBlocks :: [Int] -> Int -> [[Int]]
   placeBlocks [] _ = [[]] -- No blocks left to place
   placeBlocks (b:bs) start
     | start + remainingLength > lineLength = []
     | otherwise = do
         pos <- [start .. lineLength - remainingLength]
         rest <- placeBlocks bs (pos + b + 1)
         return (pos : rest)
     where
       -- remaining length
       remainingLength = sum (b : bs) + length bs

-- Compute all valid placements of blocks w/ chunked parallelization
computeBlocksPar :: Int -> [Int] -> [[Int]]
computeBlocksPar lineLength blockLengths =
   concat $ parMap rdeepseq processChunk (chunkList chunkSize [0 .. lineLength -
totalRemainingLength])
 where
   totalRemainingLength = sum blockLengths + length blockLengths - 1
   chunkSize = 10

   -- Divide list into chunks
   chunkList :: Int -> [a] -> [[a]]
   chunkList _ [] = []
   chunkList n xs = take n xs : chunkList n (drop n xs)

   -- Process chunk of starting positions
   processChunk :: [Int] -> [[Int]]
   processChunk positions = concatMap processPosition positions

   -- Process single starting position
   processPosition :: Int -> [[Int]]
   processPosition start = placeBlocks blockLengths start

   --Helper to place blocks
   placeBlocks :: [Int] -> Int -> [[Int]]
   placeBlocks [] _ = [[]] -- No blocks left to place
   placeBlocks (b:bs) start
     | start + remainingLength > lineLength = []
     | otherwise = do
```

```
         pos <- [start .. lineLength - remainingLength]
         rest <- placeBlocks bs (pos + b + 1)
         return (pos : rest)
     where
       remainingLength = sum (b : bs) + length bs



generateBlocksSeq :: [[Int]] -> [Int] -> Int -> [[Int]]
generateBlocksSeq blockStarts blockSizes totalLength =
   map (generateBinaryArray blockSizes totalLength) blockStarts
 where
   generateBinaryArray :: [Int] -> Int -> [Int] -> [Int]
   generateBinaryArray sizes len starts = foldl placeBlock (replicate len 0) (zip starts sizes)

   placeBlock :: [Int] -> (Int, Int) -> [Int]
   placeBlock arr (start, size) =
       take start arr ++ replicate size 1 ++ drop (start + size) arr

-- Parallelized generateBlocks with chunking
generateBlocksPar :: [[Int]] -> [Int] -> Int -> [[Int]]
generateBlocksPar blockStarts blockSizes totalLength =
   concat $ parMap rdeepseq processChunk (chunkList chunkSize blockStarts)
 where
   chunkSize = 10

   -- Divide list into chunks
   chunkList :: Int -> [a] -> [[a]]
   chunkList _ [] = []
   chunkList n xs = take n xs : chunkList n (drop n xs)

   -- Process chunk of blockStarts
   processChunk :: [[Int]] -> [[Int]]
   processChunk chunk = map (generateBinaryArray blockSizes totalLength) chunk

   -- Generate binary array for single starting configuration
   generateBinaryArray :: [Int] -> Int -> [Int] -> [Int]
   generateBinaryArray sizes len starts =
       foldl' placeBlock (replicate len 0) (zip starts sizes)

   -- Place single block in  array
```

```haskell
   placeBlock :: [Int] -> (Int, Int) -> [Int]
   placeBlock arr (start, size) =
       take start arr ++ replicate size 1 ++ drop (start + size) arr




validGroups :: [Int] -> String -> Bool
validGroups expected line = groupLengths == expected
 where
   -- Filter out zeroes and compute lengths of groups of ones
   groupLengths = map length . filter (all (== '1')) $ group line

------------------------------------------------------------------------------------------------------------------------------------------------

type Constraint = [Int]
type PartialSolution = Array (Int, Int) Int

-- Helper function: Extract groups of 1s from a row/column as strings
extractGroupsAsString :: [Int] -> String
extractGroupsAsString = map toChar
 where
   toChar 1 = '1'
   toChar 0 = '0'
   toChar _ = '-'

-- Function to validate single row or column
validateLine :: [Int] -> Constraint -> Bool
validateLine line constraint
 | all (== -1) line = True -- Skip untouched lines
 | sum (filter (== 1) line) > sum constraint = False -- Exceeding constraints
 | any (\x -> x == -1) line && sum (filter (== 1) line) <= sum constraint = True -- Partially solved line
 | otherwise = validGroups constraint (extractGroupsAsString line)

-- Main validation function
valid :: [Constraint] -> [Constraint] -> PartialSolution -> Set.Set Int -> Set.Set Int -> Bool
valid rowArgs colArgs partialSolution completedRows completedCols =
 let
   -- Get dimensions of grid
```

```
   ((r1, c1), (r2, c2)) = bounds partialSolution
   -- Extract localRows and columns from partial solution
   localRows = [[partialSolution ! (r, c) | c <- [c1..c2]] | r <- [r1..r2]]
   cols = transpose localRows
   -- Validate localRows
   validRows = all (\(r, row) ->
     let
       constraint = rowArgs !! r
     in
       if r `Set.member` completedRows
       then validGroups constraint (extractGroupsAsString row)
       else validateLine row constraint
     ) $ zip [0..] localRows
   -- Validate columns
   validCols = all (\(c, col) ->
     let
       constraint = colArgs !! c
     in
       if c `Set.member` completedCols
       then validGroups constraint (extractGroupsAsString col)
       else validateLine col constraint
     ) $ zip [0..] cols

 in
   validRows && validCols

--------------------------------------------------------------------------------------
------------------------------------------------------------------

inferValuesSeq :: PartialSolution -> (Array Int (Set.Set [Int]), Array Int (Set.Set
[Int])) -> PartialSolution
inferValuesSeq partialSolution (rowPlacements, colPlacements) =
   let ((r1, c1), (r2, c2)) = bounds partialSolution
       numCols = c2 - c1 + 1
       numRows = r2 - r1 + 1

       -- Update localRows
       updatedRows = partialSolution // concat (parMap rseq updateRow [r1..r2])
         where
           updateRow r =
               let placements = Set.toList $ rowPlacements ! r
               in if null placements
```

```
                  then []  -- If there are no placements, skip updates for this row
                  else
                      let rowOnes = foldl1 (zipWith (.&.)) placements
                          rowZeros = foldl1 (zipWith (.|.)) placements
                          inferredRow = inferRowOrCol rowOnes rowZeros
                      in [((r, c), inferredRow !! c) | c <- [0..numCols - 1],
partialSolution ! (r, c) == -1]

       -- Update columns
       updatedCols = updatedRows // concatMap updateCol [c1..c2]
         where
           updateCol c =
               let placements = Set.toList $ colPlacements ! c
               in if null placements
                  then []  -- If there are no placements, skip updates for this column
                  else
                      let colOnes = foldl1 (zipWith (.&.)) placements
                          colZeros = foldl1 (zipWith (.|.)) placements
                          inferredCol = inferRowOrCol colOnes colZeros
                      in [((r, c), inferredCol !! r) | r <- [0..numRows - 1],
updatedRows ! (r, c) == -1]

   in updatedCols


inferValuesPar :: PartialSolution -> (Array Int (Set.Set [Int]), Array Int (Set.Set
[Int])) -> PartialSolution
inferValuesPar partialSolution (rowPlacements, colPlacements) =
   let ((r1, c1), (r2, c2)) = bounds partialSolution
       numCols = c2 - c1 + 1

       -- Parallel update for localRows
       updatedRows = partialSolution // concat (parMap rseq updateRow [r1..r2])
         where
           updateRow r =
               let placements = Set.toList $ rowPlacements ! r
               in if null placements
                  then []  -- If there are no placements, skip updates for this row
                  else
                      let rowOnes = foldl1 (zipWith (.&.)) placements
                          rowZeros = foldl1 (zipWith (.|.)) placements
                          inferredRow = inferRowOrCol rowOnes rowZeros
```

```
                      in [((r, c), inferredRow !! c) | c <- [0..numCols - 1],
partialSolution ! (r, c) == -1]

       -- Sequential update for columns (after localRows are done)
       numRows = r2 - r1 + 1
       updatedCols = updatedRows // concatMap updateCol [c1..c2]
         where
           updateCol c =
               let placements = Set.toList $ colPlacements ! c
               in if null placements
                  then []  -- If there are no placements, skip updates for this column
                  else
                      let colOnes = foldl1 (zipWith (.&.)) placements
                          colZeros = foldl1 (zipWith (.|.)) placements
                          inferredCol = inferRowOrCol colOnes colZeros
                      in [((r, c), inferredCol !! r) | r <- [0..numRows - 1],
updatedRows ! (r, c) == -1]

   in updatedCols

-- Infer result for row or column based on bitwise results
inferRowOrCol :: [Int] -> [Int] -> [Int]
inferRowOrCol ones zeros =
   zipWith resolveCell ones zeros
 where
   resolveCell 0 1 = -1   -- unknown
   resolveCell 1 1 = 1    -- filled (from ones)
   resolveCell 0 0 = 0    -- empty (from zeros)
   resolveCell _ _ = -1   -- unexpected
--------------------------------------------------------------------------------------
------------------------------------------------------------------


-- Determine completed localRows and columns in solution array
-- Input: 2D array representing current grid state
-- Output: A tuple of two sets. localRows/colCompleted: indices of localRows/cols that
are completed,
-- How: Iterate through localRows and columns, check if all cells are not -1.
updateCompletions :: Array (Int, Int) Int -> (Set Int, Set Int)
updateCompletions solutionArray =
   let ((r1, c1), (r2, c2)) = bounds solutionArray
       -- Extract localRows and check if all values in each row are not -1
```

```
       rowsCompleted = Set.fromList [r | r <- [r1..r2], all (/= -1) [solutionArray ! (r, c) | c <- [c1..c2]]]
       -- Extract columns and check if all values in each column are not -1
       colsCompleted = Set.fromList [c | c <- [c1..c2], all (/= -1) [solutionArray ! (r, c) | r <- [r1..r2]]]
   in (rowsCompleted, colsCompleted)


type Placement = [Int]
type PlacementsDict = (Array Int (Set.Set [Int]), Array Int (Set.Set [Int])) -- (rowPlacements, colPlacements)

-- Inputs: solutionArray, rowPlacements & colPlacements, completedRows & completedCols
-- Output: updated Bool (indicates whether it made any updates), PlacementsDict (updated rowPlacements & colPlacements)
-- Calls: updateRow, isValidRow, updateCol, isValidCol
updatePlacements :: Array (Int, Int) Int -> PlacementsDict -> Set.Set Int -> Set.Set Int -> (Bool, PlacementsDict)
updatePlacements solutionArray (rowPlacements, colPlacements) completedRows completedCols =
   let
       -- Update localRows
       (updatedRows, updatedRowPlacements) = processRows solutionArray rowPlacements completedRows
       -- Update columns
       (updatedCols, updatedColPlacements) = processColumns solutionArray colPlacements completedCols
       -- Combine results
       updated = updatedRows || updatedCols
   in
       (updated, (updatedRowPlacements, updatedColPlacements))

-- Process localRows: remove completed localRows and validate remaining placements
processRows :: Array (Int, Int) Int -> Array Int (Set.Set Placement) -> Set.Set Int -> (Bool, Array Int (Set.Set Placement))
processRows solutionArray rowPlacements completedRows =
   let
       indexesForDeletion = [i | i <- indices rowPlacements, i `Set.member` completedRows]
       validatePlacement rowIdx placement = all isValidCell (zip [0..] placement)
           where
               isValidCell (cellIdx, value) =
```

```
                    (value /= 0 || solutionArray ! (rowIdx, cellIdx) /= 1) &&
                    (value /= 1 || solutionArray ! (rowIdx, cellIdx) /= 0)
        processRow (rowUpdated, acc) rowIdx
            | rowIdx `Set.member` completedRows = (rowUpdated, acc // [(rowIdx, Set.empty)])
            | otherwise =
                let
                    validPlacements = Set.filter (validatePlacement rowIdx) (rowPlacements ! rowIdx)
                in
                    if validPlacements /= rowPlacements ! rowIdx
                        then (True, acc // [(rowIdx, validPlacements)]) -- make updates to acc array
                        else (rowUpdated, acc)
        (updated, newPlacements) = foldl' processRow (False, rowPlacements) (indices rowPlacements)
    in
        (updated, newPlacements)


-- Process columns: remove completed columns and validate remaining placements
processColumns :: Array (Int, Int) Int -> Array Int (Set.Set Placement) -> Set.Set Int -> (Bool, Array Int (Set.Set Placement))
processColumns solutionArray colPlacements completedCols =
    let
        indexesForDeletion = [i | i <- indices colPlacements, i `Set.member` completedCols] -- create a list of indices to remove later
        validatePlacement colIdx placement = all isValidCell (zip [0..] placement) -- colIdx is index of column being validated, placement is a candidate column placement
            where -- iterates over each cell in placement to get row indx and value
                isValidCell (cellIdx, value) =
                    (value /= 0 || solutionArray ! (cellIdx, colIdx) /= 1) &&
                    (value /= 1 || solutionArray ! (cellIdx, colIdx) /= 0)
        processCol (colUpdated, acc) colIdx
            | colIdx `Set.member` completedCols = (colUpdated, acc // [(colIdx, Set.empty)]) -- mark column as empty if completed
            | otherwise =
                let
                    validPlacements = Set.filter (validatePlacement colIdx) (colPlacements ! colIdx)
                in
                    if validPlacements /= colPlacements ! colIdx
```

```
                       then (True, acc // [(colIdx, validPlacements)])
                       else (colUpdated, acc)
       (updated, newPlacements) = foldl' processCol (False, colPlacements) (indices
colPlacements)
   in
       (updated, newPlacements)




-------------------------------------------------------------------------------------
-----------------------------------------------------------------
-- Output: a list of PartialSolution(s)
backtrack :: PartialSolution
         -> Array Int (Set [Int])  -- Row placements
         -> [Constraint]           -- Row constraints
         -> [Constraint]           -- Column constraints
         -> Set Int                -- Completed localRows
         -> Set Int                -- Completed columns
         -> [PartialSolution]      -- Accumulated solutions
         -> [PartialSolution]      -- Final list of valid solutions
backtrack solutionArray rowPlacements rowArgs colArgs completedRows completedCols
solutions
   | Set.size completedRows == length rowArgs =
       -- Base case: if all localRows are completed, add solution to list if unique
       if not (any (== solutionArray) solutions)
       then solutionArray : solutions
       else solutions
   | otherwise =
       -- Iterate over localRows in rowPlacements
       foldl tryPlacement solutions (assocs rowPlacements)
 where
   -- Try a placement for a given row
   tryPlacement :: [PartialSolution] -> (Int, Set.Set [Int]) -> [PartialSolution]
   tryPlacement solList (row, options) =
       if Set.member row completedRows
       then solList -- Skip completed localRows
       else foldl (tryOption row) solList (Set.toList options)

   -- Try an option for a specific row
   tryOption :: Int -> [PartialSolution] -> [Int] -> [PartialSolution]
   tryOption row solList option =
       let
```

```haskell
           originalRow = [solutionArray ! (row, col) | col <- colIndices]
           updatedSolution = solutionArray // [((row, col), option !! (col - c1)) | col <- colIndices]
           (completedRowsNext, completedColsNext) = updateCompletions updatedSolution
           newRowPlacements = rowPlacements // [(row, Set.empty)]
       in
       if not (valid rowArgs colArgs updatedSolution completedRowsNext completedColsNext)
       then solList
       else
           -- Append solutions returned from recursive call to current list
           backtrack updatedSolution newRowPlacements rowArgs colArgs completedRowsNext completedColsNext solList
       where
           ((_, c1), (_, c2)) = bounds solutionArray
           colIndices = [c1..c2]


iterativeSolveSeq :: PartialSolution
              -> PlacementsDict       -- Row and column placements
              -> [Constraint]         -- Row constraints
              -> [Constraint]         -- Column constraints
              -> Set Int              -- Completed localRows
              -> Set Int              -- Completed columns
              -> (PartialSolution, PlacementsDict, Set Int, Set Int)
iterativeSolveSeq solutionArray (rowPlacements, colPlacements) rowArgs colArgs completedRows completedCols =
   let
       -- Infer values and update completions
       inferredSolution = inferValuesSeq solutionArray (rowPlacements, colPlacements)

       -- Update completions
       (newCompletedRows, newCompletedCols) = updateCompletions inferredSolution

       -- Update placements
       (updatedFlag, newPlacements) = updatePlacements inferredSolution (rowPlacements, colPlacements) newCompletedRows newCompletedCols
       (newRowPlacements, newColPlacements) = newPlacements
   in
       if not updatedFlag
       then (inferredSolution, (newRowPlacements, newColPlacements), newCompletedRows, newCompletedCols)
```

```haskell
       else iterativeSolveSeq inferredSolution (newRowPlacements, newColPlacements)
rowArgs colArgs newCompletedRows newCompletedCols

iterativeSolvePar :: PartialSolution
              -> PlacementsDict       -- Row and column placements
              -> [Constraint]         -- Row constraints
              -> [Constraint]         -- Column constraints
              -> Set Int              -- Completed localRows
              -> Set Int              -- Completed columns
              -> (PartialSolution, PlacementsDict, Set Int, Set Int)
iterativeSolvePar solutionArray (rowPlacements, colPlacements) rowArgs colArgs
completedRows completedCols =
   let
       -- Infer values and update completions
       inferredSolution = inferValuesPar solutionArray (rowPlacements, colPlacements)

       -- Update completions
       (newCompletedRows, newCompletedCols) = updateCompletions inferredSolution

       -- Update placements
       (updatedFlag, newPlacements) = updatePlacements inferredSolution
(rowPlacements, colPlacements) newCompletedRows newCompletedCols
       (newRowPlacements, newColPlacements) = newPlacements
   in
       if not updatedFlag
       then (inferredSolution, (newRowPlacements, newColPlacements), newCompletedRows,
newCompletedCols)
       else iterativeSolvePar inferredSolution (newRowPlacements, newColPlacements)
rowArgs colArgs newCompletedRows newCompletedCols

--------------------------------------------------------------------------------------
------------------------------------------------------------------

-- Helper to print a solution in a grid format
printSolution :: PartialSolution -> IO ()
printSolution solution = do
   let ((r1, c1), (r2, c2)) = bounds solution
   mapM_ (putStrLn . concatMap show) [[solution ! (r, c) | c <- [c1..c2]] | r <-
[r1..r2]]
   putStrLn "" -- Add a blank line between solutions
```

```
-- Generalized solveNonogramFromFile function
solveNonogramFromFile :: (Int -> [Int] -> [[Int]])            -- computeBlocks
function
                     -> ([[Int]] -> [Int] -> Int -> [[Int]]) -- generateBlocks
function
                     -> (PartialSolution -> PlacementsDict -> [Constraint] ->
[Constraint] -> Set Int -> Set Int
                         -> (PartialSolution, PlacementsDict, Set Int, Set Int)) --
iterativeSolve function
                     -> FilePath                            -- File path to Nonogram
                     -> IO ()
solveNonogramFromFile computeBlocksFunc generateBlocksFunc iterativeSolveFunc filePath
= do
   -- Parse Nonogram from provided file path
   nonogram <- parseNonogram filePath

   -- Extract row and column arguments
   let rowArgs = rows nonogram
       colArgs = columns nonogram

   putStrLn $ "\nTitle: " ++ title nonogram

{-     putStrLn $ "\nTitle: " ++ title nonogram
   putStrLn "Row Hints (rowArgs):"
   mapM_ print rowArgs
   putStrLn "Column Hints (colArgs):"
   mapM_ print colArgs -}

   -- Compute row and column vector lengths
   let rowVectorLen = length colArgs
   let colVectorLen = length rowArgs

   -- Compute placements using passed-in functions
   let rowPlacements = listArray (0, length rowArgs - 1) $
           map (\arg -> Set.fromList (generateBlocksFunc (computeBlocksFunc
rowVectorLen arg) arg rowVectorLen)) rowArgs

   let colPlacements = listArray (0, length colArgs - 1) $
           map (\arg -> Set.fromList (generateBlocksFunc (computeBlocksFunc
colVectorLen arg) arg colVectorLen)) colArgs
```

```
   -- Initialize partial solution grid
   let partialSolution = array ((0, 0), (length rowArgs - 1, length colArgs - 1))
                       [((r, c), -1) | r <- [0..length rowArgs - 1], c <- [0..length
colArgs - 1]]

   let completedRows = Set.empty
   let completedColumns = Set.empty

   -- Step 1: Iterative solving
   -- putStrLn "\nTesting iterativeSolve...\n"
   let placementsDict = (rowPlacements, colPlacements)
   let (finalSolution, _, _, _) =
           iterativeSolveFunc partialSolution placementsDict rowArgs colArgs
completedRows completedColumns

   -- putStrLn "\nFinal Solution:"
   printSolution finalSolution

solveNonogramBacktrack :: (Int -> [Int] -> [[Int]])            -- computeBlocks
function
                     -> ([[Int]] -> [Int] -> Int -> [[Int]]) -- generateBlocks
function
                     -> (PartialSolution -> Array Int (Set [Int]) -> [Constraint] ->
[Constraint] -> Set Int -> Set Int
                         -> [PartialSolution] -> [PartialSolution]) -- backtrack
function
                     -> FilePath                            -- File path to the
Nonogram
                     -> IO ()
solveNonogramBacktrack computeBlocksFunc generateBlocksFunc backtrackFunc filePath =
do
   -- Parse Nonogram from provided file path
   nonogram <- parseNonogram filePath

   -- Extract row and column arguments
   let rowArgs = rows nonogram
       colArgs = columns nonogram

   putStrLn $ "\nTitle: " ++ title nonogram

{-     putStrLn $ "\nTitle: " ++ title nonogram
   putStrLn "Row Hints (rowArgs):"
```

```haskell
   mapM_ print rowArgs
   putStrLn "Column Hints (colArgs):"
   mapM_ print colArgs -}

   -- Compute row and column vector lengths
   let rowVectorLen = length colArgs
   let colVectorLen = length rowArgs

   -- Compute placements using passed-in functions
   let rowPlacements = listArray (0, length rowArgs - 1) $
           map (\arg -> Set.fromList (generateBlocksFunc (computeBlocksFunc rowVectorLen arg) arg rowVectorLen)) rowArgs

   let colPlacements = listArray (0, length colArgs - 1) $
           map (\arg -> Set.fromList (generateBlocksFunc (computeBlocksFunc colVectorLen arg) arg colVectorLen)) colArgs

   -- Initialize partial solution grid
   let partialSolution = array ((0, 0), (length rowArgs - 1, length colArgs - 1))
                       [((r, c), -1) | r <- [0..length rowArgs - 1], c <- [0..length colArgs - 1]]

   let completedRows = Set.empty
   let completedColumns = Set.empty

   putStrLn "Solving via backtracking..."


   let solutions = backtrack partialSolution rowPlacements rowArgs colArgs completedRows completedColumns []

   putStrLn "Found solutions:"
   mapM_ printSolution solutions



--------------------------------------------------------------------------------------------------------------------------------------------------------


solveSequential :: FilePath -> IO ()
```

```
solveSequential = solveNonogramFromFile computeBlocksSeq generateBlocksSeq iterativeSolveSeq

solveParallelComputeBlocks :: FilePath -> IO ()
solveParallelComputeBlocks = solveNonogramFromFile computeBlocksPar generateBlocksSeq iterativeSolveSeq

solveParallelGenerateBlocks :: FilePath -> IO ()
solveParallelGenerateBlocks = solveNonogramFromFile computeBlocksSeq generateBlocksPar iterativeSolveSeq

solveParallelComputeGenerate :: FilePath -> IO ()
solveParallelComputeGenerate = solveNonogramFromFile computeBlocksPar generateBlocksPar iterativeSolveSeq

solveParallelIterativeSolve :: FilePath -> IO ()
solveParallelIterativeSolve = solveNonogramFromFile computeBlocksSeq generateBlocksSeq iterativeSolvePar

solveFullyParallel :: FilePath -> IO ()
solveFullyParallel = solveNonogramFromFile computeBlocksPar generateBlocksPar iterativeSolvePar
```

7.5 TestNonogramSolver.hs

```
module Main (main) where

import Test.HUnit
import NonogramSolverPar (solveSequential)
import NonogramTypes (Nonogram(..))
import Parser (parseNonogram)
import Data.Array (Array, bounds, (!))
import System.IO.Silently (capture)

-- test single puzzle
testPuzzle :: FilePath -> Test
testPuzzle puzzlePath = TestCase $ do
   -- Parse the puzzle
```

```
   nonogram <- parseNonogram puzzlePath

   -- Check if a goal is provided
   let expectedSolution = goal nonogram
   if null expectedSolution
       then assertFailure $ "No solution provided in puzzle file: " ++ puzzlePath
       else do
           -- Capture console output
           (actualOutput, _) <- capture (solveSequential puzzlePath)

           -- Clean and compare output
           let actualSolution = cleanOutput actualOutput
           assertEqual ("Mismatch for puzzle: " ++ puzzlePath) expectedSolution
actualSolution


-- Helper function to reformat captured output
cleanOutput :: String -> String
cleanOutput = concat . map (filter (`elem` "01")) . lines


-- test cases
tests :: Test
tests = TestList
   [ testPuzzle "test/test_cases/bloop.txt",
       testPuzzle "test/test_cases/spade.txt" ]


-- Run the tests
main :: IO ()
main = do
   testResults <- runTestTT tests
   print testResults
```