Parallel KenKen Solver Report

Harry Smith
hs3061
December 18, 2019

1 How it Works

1.1 The KenKen Puzzle

My project solves KenKen puzzles, which are Sudoku-like number puzzles played on a square $N \times N$ grid for any size N. Each row and each column must contain exactly one instance of each number between 1 and N. Additionally, there are units partitioning the game board made of contiguous cells. Each of these units indicate other mathematical constraints that the values entered in the constituent cells must satisfy. In particular, these constraints are adding up to a target sum, subtracting (largest to smallest) to a value, multiplying to a target sum, or dividing (largest to smallest) to a particular quotient. A unit may also contain just one cell, in which case its associated target value is automatically the value that the cell will take in a correct solution. Subject to the constraint of no repeats within any row or column, the same number may appear more than one time in the same unit.

I obtained a corpus of KenKen puzzles from `http://www.mlsite.net/neknek/play.php`. Each puzzle has a single solution. Below, I present an example puzzle.

```
# 7
* 336 A1 A2 A3 A4
+ 8 A5 A6 B6
+ 10 A7 B7 C7
* 210 B1 C1 D1
+ 4 B2 B3
* 20 B4 C3 C4
! 6 B5
! 7 C2
+ 13 C5 D4 D5 E5
+ 13 C6 D6 D7
* 30 D2 E1 E2
+ 7 D3 E3 E4
* 210 E6 E7 F6 F7
! 4 F1
+ 8 F2 G1 G2
+ 8 F3 F4
* 168 F5 G5 G6 G7
-2 G3 G4
```

Observe that each row specifies a unit with a constraining operation, a target value, and the addresses of the constituent cells listed in row major order (i.e. 'B4' refers to row 2, column 4). The ! operator indicates a unit of size 1, which has as its target value the correct assignment for the only cell in this unit. Additionally, the first line of the specification indicates the size of the puzzle (in this case, 7).

1.2 The Code

The majority of the logic for the game of KenKen is contained within the file `src/Kenken.hs`. The code which is responsible for parallelizing the execution of the solver on the list of input puzzles is located in `app/Main.hs`. Each puzzle is individually contained in a file `puzzles/<SIZE>_<INDEX>.txt` with a corresponding solution in `solutions/<SIZE>_<INDEX>.txt`.

1.2.1 Kenken.hs

This module specifies a handful of datatypes used in the puzzle logic, the most interesting of which are `Constraint` and `Partial`. A `Constraint` is a representation of one row of the puzzle. Specifically, it represents an operation which accumulates its members to reach a target value. Although the uniqueness of elements in rows and columns could be represented as a `Constraint`, I chose to represent these restrictions implicitly to maintain that each cell has precisely one `Constraint`.

```
data Constraint = Constraint {members :: [Address]
  , op :: Operation
  , target :: Int
  } deriving (Show, Eq)
```

Additionally, we have the data type `Partial`. This represents a *partial* solution to the KenKen puzzle, while also keeping track of the relationships among cells implied by the underlying constraints. In particular, the `state` field tracks all candidate values for each cell. The field `pPeers` maps each cell to the set of other cells in the same constraining unit (I call them the cells 'peers') and `pUnit` maps each address to its own constraining unit.

```
data Partial = Partial {state :: M.Map Address [Int]
                       , pPeers :: M.Map Address (S.Set Address)
                       , pUnits :: M.Map Address Constraint
                       } deriving (Show, Eq)
```

Lines 23 through 80 of `Kenken.hs` provide the framework for how to parse a puzzle from its String representation into an initial `Partial`, which represents a puzzle with all constraints formalized but without any steps taken to solve it. I leave these functions to be explored in the source code by the reader, since I believe that the underlying logic used to write them is somewhat easily intuitied from the design of the `Constraint` and `Partial` datatypes discussed above.

At this point, I will introduce the basic pipeline of how I solve a KenKen puzzle. The algorithm is broadly a depth-first search, where at each call to the function `search`, the computer attempts to assign a value to a cell chosen from its current available values. Whenever a cell has a value assigned to it or has a potential value eliminated, three basic steps occur. First, the actual change to the `state` of the `Partial` is made, resulting in a new `Partial`. Second, constraints are propagated from the changed cell to eliminate or assign potential values in its peers, resulting in a new `Partial` generated for each deductive change.[1] Finally, inconsistent puzzles result in terminated execution paths.

Progress is made in this search by doing either an assignment or an elimination; however, I have chosen to implement assignment of a chosen value (in `assign`) as repeated elimination of values in this cell that are **not** the chosen value.

```
assign :: Address -> Int -> Partial -> Maybe Partial
assign a v p =
  do assigned <- foldM f p toRemove
     propagateSet assigned a
  where toRemove = filter (/= v) $ (state p) M.! a
        f = \partial value -> eliminate a value partial
```

The elementary operation of this algorithm is thus the *elimination*, presented here:

```
eliminate :: Address -> Int -> Partial -> Maybe Partial
eliminate a v p =
  do removed <- remove a v p
     unitPropagated <- propagateUnit removed a
     return unitPropagated
```

Elimination is responsible for removing a single value as a possibility at a particular address and then propagating the constraint of that address' unit forward. The propagation step here is most robust when handling cells with addition or multiplication as their operations. In these cases, it is quite simple to determine whether any of the remaining value combinations lead to a consistent assignment. When there is no such combination, the search down this game state ceases. This pruning is quite imperative for runtime efficiency. For a time, I attempted to carry the effects of the new assignment forward and proactively remove values that would now be inconsistent. While the process of finding newly inconsistent values was relatively simple and efficient, the extra propagation steps resulted in a significant slowdown due to expensive further eliminations down paths that ended up becoming inconsistent anyway.

The search is terminated in a successful state when every list contained in `state` has exactly one member.

1.2.2 Main.hs

I include in full here a description of `Main.hs`, which provides the logic for solving multiple puzzles from a manifest provided as a command line argument.

```
main :: IO ()
main = do puzzles <- getArgs >>= \[f] -> lines <$> readFile f
          let solutions = parMap rpar solve puzzles
          print $ length $ filter isJust $ solutions
```

Of course, I provide heavy credit to the course notes for providing essentially this exact code. Much like the example provided in class, the file that `Main` expects contains one puzzle description per line. To make this possible, I converted the puzzle format from a line-separated format to a semicolon-separated format. From there, the use of `parMap` with the associated strategy `rpar` results in a dynamically partition set of problems where each call to `solve` is handled by a separate spark.

[1] As an example of this second step, consider the 'trick' of assigning the value 1 to the only empty cell in a row which has all other values $2 \dots N$ assigned.

2 Effects of Parallelism

I'll begin by presenting the raw statistics on how my program performs with increasing numbers of cores. Broadly, then, parallelizing this solver is a success. The main

Table 1:

# cores	Time elapsed (s)	Speedup	Sparks Created	Conversions	GCs	Fizzles
1	5.026	1x	1362	0	258	1104
2	2.856	1.76x	1362	1350	2	10
3	2.054	2.45x	1362	1361	1	0
4	1.820	2.76x	1362	1246	13	103

shortcoming becomes apparent when looking at a Threadscope trace of an execution on multiple cores. Presented below in Figure 1 the visualization for the execution on four cores. Immediately obvious is a load balancing problem, despite the strategy of

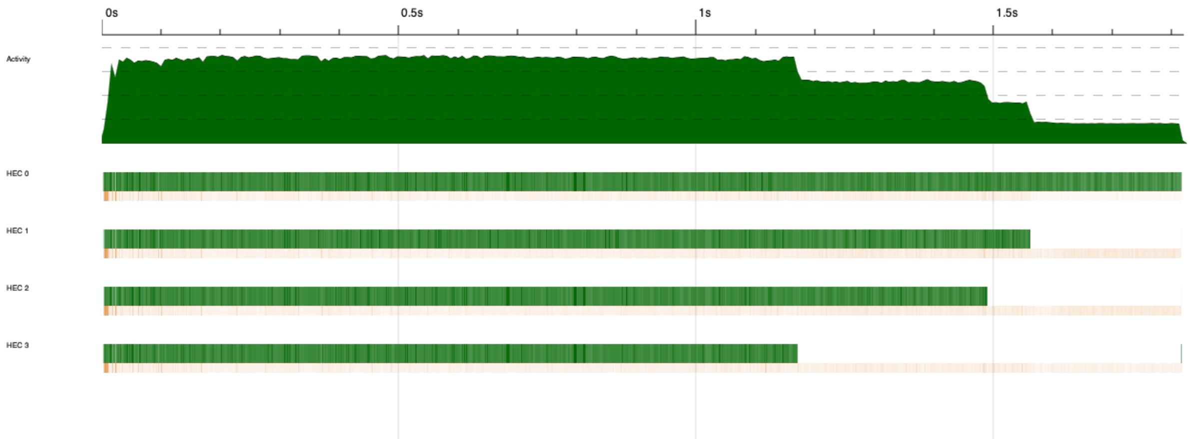

Figure 1: Execution Viz for Four Cores on Puzzles Sizes 3-7

dynamic balancing that I attempted. This is a result of puzzles of size 7 being relatively slow for my solver to handle and the fact that these puzzles begin to have more variance in how long they take to solve. Among the first ten puzzles of size 7, there are some that take less than aa tenth of a second and others that take over half of a second. So while this is obviously detrimental to runtime, this is a shortcoming of the underlying sequential code rather than the parallelization strategy taken. If we compare with Figure 2, an execution on puzzles of up to size 6 which are much quicker to solve, we see that the load balancing issue is much less significant. Indeed, on these smaller puzzles, we get a speedup of $3.2\times$ when running the solver on four cores $(0.69s)$ over a single core $(2.21s)$. To improve this going forward, it would be helpful to parallelize the search step within the solving algorithm as well. This would allow the machine to follow along multiple independent execution paths at once, perhaps speeding up execution even further.

3 Code Listing

Kenken.hs

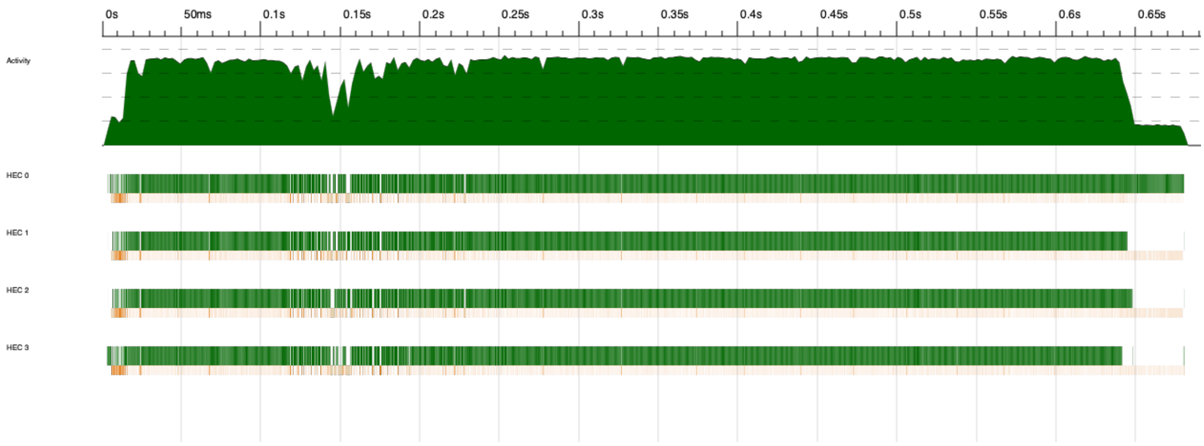

Figure 2: Execution Viz for Four Cores on Puzzles Sizes 3-6

```haskell
{-# LANGUAGE NamedFieldPuns #-}
module Kenken where

import            Data.Char       (ord)
import qualified Data.Map        as M
import qualified Data.Set        as S
import qualified Data.List       as L
import            Data.List.Split (splitOn)
import            Control.Monad  (guard, foldM, msum, fmap)
import            System.IO      (IOMode (ReadMode), hGetContents, openFile)

type Address = (Int, Int)
data Constraint = Constraint {members :: [Address], op :: Operation, target :: Int} deriving (Show, Eq)
data Operation = Add | Sub | Mul | Div | Asrt deriving (Show, Eq)

data Partial = Partial {state :: M.Map Address [Int]
                       , pPeers :: M.Map Address (S.Set Address)
                       , pUnits :: M.Map Address Constraint
                       } deriving (Show, Eq)


crossProd :: [a] -> [b] -> [(a, b)]
crossProd as bs = (,) <$> as <*> bs

translate :: String -> Address
translate (r : c : []) = (ord r - 64, read $ c : "")
translate _            = undefined

symToOp :: String -> Operation
symToOp "+"    = Add
symToOp "-"    = Sub
symToOp "*"    = Mul
symToOp "\\/" = Div
symToOp "!"    = Asrt
symToOp _   = undefined

readKenkenFile :: String -> IO (String)
readKenkenFile fname = do
                    h <- openFile fname ReadMode
                    hGetContents h
```

```haskell

showUnits :: Address -> IO ([Constraint], Int) -> IO (Maybe Constraint)
showUnits addr gameIO = do game <- gameIO
                           let us = units game
                           return $ M.lookup addr us

showPeers :: Address -> IO ([Constraint], Int) -> IO (Maybe (S.Set Address))
showPeers addr gameIO = do game <- gameIO
                           let ps = peers . units $ game
                           return $ M.lookup addr ps

readKenken :: String -> ([Constraint], Int)
readKenken kkStr = let spec : groups = splitOn ";" kkStr
                       translated = do
                                      (op : target : group) <- map words groups
                                      return $ Constraint {op = symToOp op,
                                                           target = read target,
                                                           members = map translate group}
                       [_, size] = words spec in
                    (translated, read size)

units :: ([Constraint], Int) -> M.Map Address Constraint
units (constraints, _) = M.fromList associations
          where associations = [(a, c) |
                                 c <- constraints,
                                 a <- members c
                                ]

peers :: M.Map Address Constraint -> M.Map Address (S.Set Address)
peers m = M.fromList [(addr, neighbors) | (addr, c) <- M.toList m,
                                     let duplicateSet = S.fromList $ members c
                                         neighbors = duplicateSet `S.difference` S.singleton addr]

parsePuzzle :: ([Constraint], Int) -> Partial
parsePuzzle cs@(constraints, size) = Partial {state, pPeers, pUnits} where
                                  pUnits = units cs
                                  pPeers = peers pUnits
                                  state = M.fromList [(a, [1..size]) | c <- constraints, a <- members c]

eliminate :: Address -> Int -> Partial -> Maybe Partial
eliminate a v p =
  do removed <- remove a v p
     unitPropagated <- propagateUnit removed a
     return unitPropagated


assign :: Address -> Int -> Partial -> Maybe Partial
assign a v p =
  do assigned <- foldM (\partial value -> eliminate a value partial) p toRemove
     propagateSet assigned a
```

```
  where toRemove = filter (/= v) $ (state p) M.! a

remove :: Address -> Int -> Partial -> Maybe Partial
remove adr val p =
  do candidates <- M.lookup adr $ state p
     let reducedCands = L.delete val candidates
         len = L.length reducedCands
     guard (len /= 0)
     return p{state=M.insert adr reducedCands $ state p}

propagateSet :: Partial -> Address -> Maybe Partial
propagateSet p@Partial{state} adr@(row, col) =
  let  [x] = state M.! adr
       size = fst $ fst $ M.findMax state
       setPeers = [(row, c) | c <- [1..size], c /= col] ++
                  [(r, col) | r <- [1..size], r /= row]
       in foldM (\partial peer -> eliminate peer x partial) p setPeers

propagateUnit :: Partial -> Address -> Maybe Partial
propagateUnit p@Partial{pUnits} adr =
  let constraint = pUnits M.! adr in
  case op $ constraint of
    Add -> propagateAdd p constraint
    Mul -> propagateMul p constraint
    Sub -> propagateSub p constraint
    Div -> propagateDiv p constraint
    _ -> Just p

existsSum ::  Int -> [[Int]] -> Bool
existsSum target []         = target == 0
existsSum target (hd:rest)  = or [existsSum (target - choice) $ rest |
    choice <- L.reverse hd, choice <= target]

existsProd ::  Int -> [[Int]] -> Bool
existsProd target []         = target == 1
existsProd target (hd:rest)  = or [existsProd (target `div` choice) $
    rest | choice <- L.reverse hd, target `mod` choice == 0]

propagateAdd :: Partial -> Constraint -> Maybe Partial
propagateAdd p@Partial{state} constraint =
  let prs = members constraint
      t = target constraint
      possibilities = [(peer, state M.! peer) | peer <- prs]
      fixed = L.filter (\(_,l) -> L.length l == 1) $ possibilities
      free = L.filter (\(_,l) -> L.length l > 1) $ possibilities
      freeVals = map snd free
      fixedVals = map snd fixed
      newTarget = t - (sum $ msum $ fixedVals) in
  if not $ existsSum newTarget $ L.reverse $ L.sort $ freeVals
  then Nothing
  else return p

propagateMul :: Partial -> Constraint -> Maybe Partial
propagateMul p@Partial{state} constraint =
  let prs = members constraint
      t = target constraint
      possibilities = [(peer, state M.! peer) | peer <- prs]
      fixed = L.filter (\(_,l) -> L.length l == 1) $ possibilities
      free = L.filter (\(_,l) -> L.length l > 1) $ possibilities
```

```
      fixedVals = map snd fixed
      freeVals = map snd free
      newTarget = t `div` (product $ msum $ fixedVals) in
  if not $ existsProd newTarget $ freeVals
  then Nothing
  else return p



propagateSub:: Partial -> Constraint -> Maybe Partial
propagateSub p@Partial{state} constraint =
  let prs = members constraint
      t =  target constraint
      possibilities = [(peer, state M.! peer) | peer <- prs]
      fixed = L.filter (\(_,l) -> L.length l == 1) $ possibilities
      free = L.filter (\(_,l) -> L.length l > 1) $ possibilities
      largest:rest = L.reverse $ L.sort $ msum $ map snd fixed
      result = largest - (sum rest) in
  case free of
     [] -> if result == t then return p else Nothing
     _  -> return p

propagateDiv:: Partial -> Constraint -> Maybe Partial
propagateDiv p@Partial{state} constraint =
  let prs = members constraint
      t = target constraint
      possibilities = [(peer, state M.! peer) | peer <- prs]
      fixed = L.filter (\(_,l) -> L.length l == 1) $ possibilities
      free = L.filter (\(_,l) -> L.length l > 1) $ possibilities
      largest:rest = L.reverse $ L.sort $ msum $ map snd fixed
      result = largest `div` (sum rest) in
  case free of
     [] -> if result == t then return p else Nothing
     _  -> return p





reduce :: Partial -> Address -> Int -> Maybe Partial
reduce p@Partial{state} (row, col) value =
  let size = fst $ fst $ M.findMax state
      rowUnit = [(row, c) | c <- [1..size]]
      colUnit = [(r, col) | r <- [1..size]] in
  do let rowCandidates = [a | a <- rowUnit, value `elem` state M.! a]
     rowReduced <- case rowCandidates of
                        [] -> Nothing
                        [adr] -> assign adr value p
                        _ -> return p
     let colCandidates = [a | a <- colUnit, value `elem` state M.! a]
     case colCandidates of
          [] -> Nothing
          [adr] -> assign adr value rowReduced
          _ -> return rowReduced


applyAssertions  :: Partial -> Maybe Partial
applyAssertions p@Partial{pUnits} =
  let asserts = M.filter (\c -> op c == Asrt) pUnits
```

```
      assocList = L.map (\(adr, c) -> (adr, target c)) $ M.toList asserts in
  foldM (\partial (a, val) -> assign a val partial) p assocList

solve :: String -> Maybe Partial
solve s = do p <- (applyAssertions . parsePuzzle . readKenken) s
             search p

search :: Partial -> Maybe Partial
search p@Partial{state} =
  let remaining = M.filter (\l -> (tail l) /= []) state in
  if remaining == M.empty then return p
  else let  addresses = (M.toList remaining) :: [(Address, [Int])]
            assocComp = \(_, c1) (_,c2) -> (L.length c1) `compare` (L.length c2)
            (nextAdr, candidateVals) = L.minimumBy assocComp addresses
            searchAssign = \v -> assign nextAdr v p >>= \p2 -> search p2
            assigned = map searchAssign candidateVals in
       do result <- msum assigned
          checkSolution result

checkSolution :: Partial -> Maybe Partial
checkSolution p@Partial{state, pUnits} =
  if allCorrect then return p else Nothing
  where allCorrect = and $ map satisfied $ map snd $ M.toList pUnits
        satisfied Constraint{members, op, target} =
          case op of
            Add -> (sum [head $ state M.! member | member <- members]) == target
            Mul -> (product [head $ state M.! member | member <- members]) == target
            _ -> True

showSolution :: Maybe Partial -> IO ()
showSolution (Just sol) = do putStrLn $ show [v | (_, [v]) <- M.toAscList $ state sol]
showSolution (Nothing) = do putStrLn $ "No Solution Found"

parseSolutionString :: String -> [Int]
parseSolutionString "" = []
parseSolutionString (',':rest) = parseSolutionString rest
parseSolutionString (x:rest) = (read (x:"")) : (parseSolutionString rest)

readSolutionFile :: String -> IO [Int]
readSolutionFile sFile = do sol <- readFile sFile
                            return $ parseSolutionString sol
```

Main.hs

```
module Main where

import Kenken (solve)
import Control.Parallel.Strategies (parMap, rpar)
import Data.Maybe (isJust)
import System.IO (readFile)
import System.Environment (getArgs)

main :: IO ()
main = do puzzles <- getArgs >>= \[f] -> lines <$> readFile f
```

```
        let solutions = parMap rpar solve puzzles
        print $ length $ filter isJust $ solutions
```

KenkenTest.hs

```
import Test.Hspec
import Test.QuickCheck
import Control.Exception (evaluate)
import qualified Kenken as K
import qualified Data.Map      as M
import qualified Data.Set      as S
import           Data.Maybe (fromMaybe, isJust)
import           Control.Monad (msum)



main :: IO ()
main = hspec $ do
  describe "Kenken" $ do
    describe "translate" $ do

      it "A1 is top left" $ do
        (K.translate "A1") `shouldBe` ((1, 1) :: K.Address)

      it "translate puts letter as col" $ do
        (K.translate "E9") `shouldBe` ((5, 9) :: K.Address)

    describe "readKenken" $ do
      it "simple example" $ do
        (K.readKenken "#\t2;+\t3\tA1 B1;-\t1\tA2 B2") `shouldBe`
          ([K.Constraint {K.members = [(1,1), (2,1)], K.op = K.Add, K.
             target = 3},
            K.Constraint {K.members = [(1,2), (2,2)], K.op = K.Sub, K.
               target = 1}],2)

    describe "units" $ do
      it "simple example units of (1,1)" $ do
        let simplePuzzle = K.readKenken "#\t2;+\t3\tA1 B1;-\t1\tA2 B2"
            constraintMap = K.units simplePuzzle
            actualUnits = M.lookup (1,1) $ constraintMap in
            actualUnits `shouldBe`
              Just K.Constraint {K.members = [(1,1), (2,1)], K.op = K.Add
                 , K.target = 3}
      it "simple example units of (1,2)" $ do
        let simplePuzzle = K.readKenken "#\t2;+\t3\tA1 B1;-\t1\tA2 B2"
            constraintMap = K.units simplePuzzle
            actualUnits = M.lookup (1,2) $ constraintMap in
            actualUnits `shouldBe`
              Just K.Constraint {K.members = [(1,2), (2,2)], K.op = K.Sub
                 , K.target = 1}
      it "simple example units of (2,1)" $ do
        let simplePuzzle = K.readKenken "#\t2;+\t3\tA1 B1;-\t1\tA2 B2"
            constraintMap = K.units simplePuzzle
            actualUnits = M.lookup (2,1) $ constraintMap in
            actualUnits `shouldBe`
              Just K.Constraint {K.members = [(1,1), (2,1)], K.op = K.Add
                 , K.target = 3}
      it "simple example units of (2,2)" $ do
        let simplePuzzle = K.readKenken "#\t2;+\t3\tA1 B1;-\t1\tA2 B2"
```

```
            constraintMap = K.units simplePuzzle
            actualUnits = M.lookup (2,2) $ constraintMap in
            actualUnits `shouldBe`
              Just K.Constraint {K.members = [(1,2), (2,2)], K.op = K.Sub , K.target = 1}

      it "address not found" $ do
        let simplePuzzle = K.readKenken "#\t2;+\t3\tA1 B1;-\t1\tA2 B2"
            constraintMap = K.units simplePuzzle
            actualUnits = M.lookup (3, 4) $ constraintMap in
            actualUnits `shouldBe` Nothing

    describe "trying solver" $ do
      it "some 3s" $ do
        let names = ["puzzles/3_" ++ show num ++ ".txt" | num <- [0..10]]
            puzzles = map readFile names
            solved = sequence $ map (\puzzIO -> fmap K.solve $ puzzIO) puzzles
            res = fmap (all isJust) solved in
            res `shouldReturn` True
      it "some 4s" $ do
        let names = ["puzzles/4_" ++ show num ++ ".txt" | num <- [0..10]]
            puzzles = map readFile names
            solved = sequence $ map (\puzzIO -> fmap K.solve $ puzzIO) puzzles
            res = fmap (all isJust) solved in
            res `shouldReturn` True
      it "some 5s" $ do
        let names = ["puzzles/5_" ++ show num ++ ".txt" | num <- [0..10]]
            puzzles = map readFile names
            solved = sequence $ map (\puzzIO -> fmap K.solve $ puzzIO) puzzles
            res = fmap (all isJust) solved in
            res `shouldReturn` True
      it "some 6s" $ do
        let names = ["puzzles/6_" ++ show num ++ ".txt" | num <- [0..10]]
            puzzles = map readFile names
            solved = sequence $ map (\puzzIO -> fmap K.solve $ puzzIO) puzzles
            res = fmap (all isJust) solved in
            res `shouldReturn` True
      it "some 7s" $ do
        let names = ["puzzles/7_" ++ show num ++ ".txt" | num <- [0..10]]
            puzzles = map readFile names
            solved = sequence $ map (\puzzIO -> fmap K.solve $ puzzIO) puzzles
            res = fmap (all isJust) solved in
            res `shouldReturn` True

    describe "correctness" $ do
      it "a 3" $ do
        let puzzle = "puzzles/3_58.txt"
            solution = "solutions/3_58.txt"
            actualM = fmap K.solve $ readFile $ puzzle
            expectedM = K.readSolutionFile solution in
            do expected <- expectedM
               actual <- actualM
               let res = msum $ map snd $ M.toList $ K.state $ fromMaybe undefined actual
```

```
              res `shouldBe` expected

      it "a 4" $ do
        let puzzle = "puzzles/4_58.txt"
            solution = "solutions/4_58.txt"
            actualM = fmap K.solve $ readFile $ puzzle
            expectedM = K.readSolutionFile solution in
            do expected <- expectedM
               actual <- actualM
               let res = msum $ map snd $ M.toList $ K.state $ fromMaybe undefined actual
               res `shouldBe` expected

      it "a 5" $ do
        let puzzle = "puzzles/5_58.txt"
            solution = "solutions/5_58.txt"
            actualM = fmap K.solve $ readFile $ puzzle
            expectedM = K.readSolutionFile solution in
            do expected <- expectedM
               actual <- actualM
               let res = msum $ map snd $ M.toList $ K.state $ fromMaybe undefined actual
               res `shouldBe` expected

      it "a 6" $ do
        let puzzle = "puzzles/6_58.txt"
            solution = "solutions/6_58.txt"
            actualM = fmap K.solve $ readFile $ puzzle
            expectedM = K.readSolutionFile solution in
            do expected <- expectedM
               actual <- actualM
               let res = msum $ map snd $ M.toList $ K.state $ fromMaybe undefined actual
               res `shouldBe` expected

      it "a 7" $ do
        let puzzle = "puzzles/7_58.txt"
            solution = "solutions/7_58.txt"
            actualM = fmap K.solve $ readFile $ puzzle
            expectedM = K.readSolutionFile solution in
            do expected <- expectedM
               actual <- actualM
               let res = msum $ map snd $ M.toList $ K.state $ fromMaybe undefined actual
               res `shouldBe` expected
```