Functional Programming & Constraint Programming

Flip Lijnzaad and Jasper Stammes

Friday 28th January, 2022

Abstract

This is where the abstract should go

Contents

1 Constraint satisfaction problems

```
module CSP where

import Data.List
import Data.Char

-- type definitions based on formal definition of CSP in book
type Variable   = Int
type Value      = Int
type Domain     = (Variable, [Value])
type Arc        = (Variable, Variable)
type Constraint = ( Arc, [(Value, Value)] )
data Problem    = CSP { vars :: [Variable]
                      , doms :: [Domain]
                      , cons :: [Constraint] } deriving Show
```

2 The AC-3 algorithm

```
module AC3 where

import CSP
-- implementation of the AC-3 function, recursive version of the pseudocode in
-- the book; calls ‘revise‘ helper function. the book version passes a queue
-- of arcs; we use a list of constraints, since those contain the arcs
ac3 :: (Problem, Bool, [Constraint]) -> (Problem, Bool, [Constraint])
-- if the Bool flag is False, the CSP has no solution, so stop the recursion
ac3 (p, False, _) = (p, False, [])
-- if the arc queue is empty, stop the recursion and return True
ac3 (p, True, []) = (p, True,  [])
-- else, perform body of the ‘while‘ loop
ac3 (p@(CSP vars doms cons), True, ((varX, varY), rel):xs) =
  if strongLookup varX doms == newXDomain
    -- if after revising, the domain of x stays the same,
    -- continue with the next arc in the queue and pass whether newXDomain is nonempty
    then ac3 (p, not $ null newXDomain, xs)
    -- if the domain of x has changed, need to add x's neighbors to queue
    else ac3 (CSP vars newDoms cons, True, newQueue)
  where
    newXDomain = revise ((varX, varY), rel) (strongLookup varX doms) (strongLookup varY
        doms)
    -- delete x's old domain and add x's new domain to the list of domains
    newDoms    = newXDomain : delete (strongLookup varX doms) doms
    -- append to the arc queue xs the neighbors of x by filtering on (_, x)
    newQueue   = xs ++ filter (\(arc, rel) -> snd arc == varX) cons

-- perform lookup and drop the Maybe
strongLookup :: Variable -> [Domain] -> Domain
strongLookup x v = let (Just y) = lookup x v in (x,y)

-- implementation of the revise function of the pseudocode in the book
revise :: Constraint -> Domain -> Domain -> Domain
-- trivial case: if there are no constraints, pass a domain with empty list of values
revise (_ , []) (varX,  _) _ = (varX, [])
-- if the domain for x is empty, pass domain with empty list of values for x
revise (_, rel) (varX, []) _ = (varX, [])
-- else, perform body of the ‘for each‘ loop
revise (arc, rel) (varX, x:xs) (varY, ys) =
  if any (\y -> (x, y) ‘elem‘ rel) ys
    -- if there is a value y in ys that satisfies the contraint between x and y,
    -- add x to the domain and continue
    then prependToSnd x (revise (arc, rel) (varX, xs) (varY, ys))
    -- if there is none, continue without adding x
    else revise (arc, rel) (varX, xs) (varY, ys)
-- test case : revise ((100,101),[(x,y)| x<-[1..4], y<-[1..4], x==y]) (100,[1..3])
    (101,[2..4])
```

```
-- prepend a value to the value list of a domain (the second argument of the tuple)
prependToSnd :: Value -> Domain -> Domain
prependToSnd x (varX, xs) = (varX, x:xs)
```

3 Sudokus

```
module Sudoku where

import AC3
-- given a variable we create the arcs that correspond to all other variables in the same
    square, row or column
generateSudokuConstraints :: [Variable] -> [Constraint]
generateSudokuConstraints [] = []
-- given an arc, all pairs with type (Value, Value) with different digits are allowed
generateSudokuConstraints (n:xs) = map (\x -> ((n,x), [(y1,y2) | y1 <-[1..9], y2 <- [1..9],
     y1 /= y2]))
-- the other variables in the square are found by finding the x-axis and y-axis position of
     the current variable in its square
    -- eg if the y position is the middle row of the 3x3 square we have (n div 9) mod 3 ==
        1, and so we find the other square variables by also looking the row above (j = -1)
         and below (j = 1)
    (filter (/=n) (nub ([n + i + 9*j | i <- [- (n `mod` 3) .. 2- (n `mod` 3)], j <- [- (n `
        div` 9 `mod` 3) .. 2- (n `div` 9 `mod` 3)]] ++
        -- the variables in its row are found by subtracting until we get a multiple of 9
            and by adding until the next one
        [n + i | i <- [- (n `mod` 9) .. 8 - n `mod` 9]] ++
        -- and the same action for the column are found by taking the y position
        [n + 9*i | i <- [- (n `div` 9 `mod` 9) .. 8- (n `div` 9 `mod` 9)]])))
        ++ generateSudokuConstraints xs

-- we input the sudoku we want to solve as a string where empty cells are zeroes
generateSudokuDomains :: [Value] -> [Domain]
generateSudokuDomains [] = []
-- a zero means the starting domain can be any thin in [1..9], if the cell is given its
    domain has just that element
generateSudokuDomains (x:xs) | x == 0    = (80 - length xs, [1..9]):generateSudokuDomains
    xs
                             | otherwise = (80 - length xs, [x]):generateSudokuDomains xs

sudokuVars :: [Variable]
sudokuVars = [0..80]

-- test: ac3 (CSP sudokuVars (generateSudokuDomains sudoku1) (generateSudokuConstraints
    sudokuVars), True, generateSudokuConstraints sudokuVars)
-- test: ac3domain sudokuVars (generateSudokuDomains sudoku1) (generateSudokuConstraints
    sudokuVars)

-- since ac3 outputs a CSP including all of the constraints, we use this to return only the
     domain. Note that the problem has a unique solution if all problems have size 1

ac3domain :: [Variable] -> [Domain] -> [Constraint] -> [Domain]
ac3domain vars doms cons = let (CSP _ y _, _, _) = ac3 (CSP vars doms cons, True, cons) in
    sortBy (\(a,_) (b,_) -> compare a b) y

-- prints a sudoku
printSudoku :: [Domain] -> IO ()
-- base case recursion: done printing
printSudoku [] = putStr ""
printSudoku ((n, val@(value:_)):xs) =
  do
    putStr (if val == [value] then show value else "_")
    if n `mod` 3 == 2
       -- put spaces between different blocks
      then putStr " "
      else putStr ""
    if n `mod` 9 == 8
       -- put newlines at the end of rows
      then putStr "\n"
```

```
      else putStr ""
    if n `mod` 27 == 26
       -- put extra newlines to vertically separate blocks
      then putStr "\n"
      else putStr ""
    do printSudoku xs

-- solves sudoku in "sudoku.txt" in current directory
solveSudokuFromFile :: IO ()
solveSudokuFromFile = do
  sudokuString <- readFile "./sudoku.txt"
  -- make the string into a list of Ints
  let values = map digitToInt sudokuString
  -- solve the sudoku and print it
  do printSudoku $ ac3domain sudokuVars (generateSudokuDomains values) (
      generateSudokuConstraints sudokuVars)
```

References