Using AC-3 to come up with a good title

David Hildering Dennis Lindberg Joel Maxson Helen Sand
Andy S. Tatman

Monday 24$^{\text{th}}$ March, 2025

Abstract

We give a toy example of a report in *literate programming* style. The main advantage of this is that source code and documentation can be written and presented next to each other. We use the listings package to typeset Haskell source code nicely.

Contents

1 Introduction

Many problems in computer science can be written as a set of possible values for each body in the problem instance, and a set of restrictions, or constraints, between pairs of bodies. Using this method, we can then define such an instance as a graph, where each vertex has a set of possible values, or domain, and where each edge between two vertices is a restriction.

This was the basis behind the arc consistency algorithms discussed in [Mac77], including the AC-3 algorithm we use in our implementation. This algorithm aims to efficiently iterate over all of the constraints, and remove any values for which the constraints is invalidated.

Take as an example X with domain $[1, 2]$ and Y with domain $[2, 3]$, with an edge connecting them representing the contraint (==). The AC-3 algorithm will remove the value 1 from X's domain and the value 3 from Y's domain, as there is no value in the other body's domain such that the constraint is true for $X = 1$ or $Y = 3$.

Notably, AC-3 cannot *solve* problems. As an example, take 3 entities X, Y, Z, each with domain $[0, 1]$, and with edges (/ =) between each. The AC-3 algorithm does not purge any values. For each value, the other entities have a value for which the constraint holds. However, if we try to find a solution with backtracking, we find that no solution exists, as the three entities need to all have a unique value.

As such, AC-3's main purpose is to reduce the search space, by pruning values for which no solution can exist. If at least one vertex has no possible values, then there is no solution. Else, we can use backtracking to determine if a solution exists, and if so what said solution is.

We discuss our AC-3 and backtracking implementation in Section 2. We then introduce a number of different computer science problems in Section 3, and show how our implementation attempts to solve them.

2 The skeleton files

These files form the basis of our implementation, which we then use to solve various problems.

2.1 The AC3Solver library

This module contains the main algorithm and definition for our project, the `AC3` type.

```
module AC3Solver where

import Control.Monad.Writer
    ( runWriter, MonadWriter(tell), Writer )
```

To start of, we define the `AC3` instance. For each agent, we have a set of agents of type `a`. An `AC3` instance then constains a list of constraints `constraintAA`, and a list of domains. Each `constraintAA` contains a pair of agents (X,Y), and then a function, such as `(==)`, which is the constraint on the arc from X to Y. [1] Each `Domain` item contains an agent, and then a list of

[1] Note that we only allow for *binary* constraints. The AC-3 algorithm does not allow for ternary (or greater) constraints, and unary constraints can be resolved by restricting that agent's domain. [Mac77] provides other approaches for achieving path consistency, where you may have ternary (or greater) constraints.

values of type `b`.

We may have multiple constraints for a pair of agents (X,Y), such as both `(>)` and `(>=)`. The programme however expects that each agent mentioned in a constraint has exactly 1 (possibly empty) domain specified for it.

Note that we do not define an arbitrary instance for AC3. Instead, we define arbitrary instances for specific problems. (See for example Section 3.2.)

```
data AC3 a b = AC3 {
    -- Constraint should take values from the first & second agents as params x & y resp.
        in \x,y-> x ?=? y.
    -- We should allow for multiple constraints for (X,Y), eg. both (x > y) AND (x < y) in
        the set.
    cons :: [ConstraintAA a b],
    -- Assume we have 1 domain list for each variable. (TODO: Check for this? )
    domains :: [Domain a b] }

type Domain a b = (Agent a, [b])
type ConstraintAA a b = (Agent a, Agent a, Constraint b)
type Constraint a = a -> a -> Bool

type Agent a = a
```

For each constraint (X,Y,f), we want to check for each value in the domain of X whether there is a value in Y's domain such that `f x y` is satisfied. Values of X for which there is no such value in Y are removed from X's domain. We make use of the Writer monad to do an $O(1)$ lookup to see if we removed items from X's domain.

```
-- Return the elements of xs for which there exist a y \in ys, such that c x y holds.
-- Using the writer monad, we also give a O(1) method to check whether we altered x's
    domain after termination.
checkDomain :: [a] -> [a] -> Constraint a -> Writer String [a]
checkDomain [] _ _ = return []
checkDomain (x:xs) ys c = do
    rest <- checkDomain xs ys c
    if not $ null [ y' | y'<-ys, c x y'] then return $ x:rest
    else tell "Altered domain" >> return rest -- This is nicely formatted for readability,
        it could just be something simple such as ".".
```

Each time we call iterate, we start by looking for the domains of agents X & Y for our constraint (X,Y). Once we find these, we are likely to replace the original domain for X with a reduced one. We use `popXy` and `popX` to find the domains for X & Y, and at the same time we also remove the *old* domain for X. Using `popXy`, we do one walk through the list, and save two walks, compared to doing a separate lookup for y, and a separate walk to delete the old x.

```
-- PRE: x is an element of (a:as)
popX :: Eq a => Agent a -> [Domain a b] -> ([b], [Domain a b] )
popX _ [] = undefined -- should not occur.
popX x (a@(aA, aD):as) = if x == aA then (aD,as)
                         else let (x', as') = popX x as in (x', a:as')

-- PRE: x != y; x,y are elements of (a:as).
--       (else, this is not a binary constraint but a unary one.)
popXy :: Eq a => Agent a -> Agent a -> [Domain a b] -> ([b], [b], [Domain a b] )
popXy _ _ [] = undefined -- should not occur.
popXy x y (a@(aA, aD):as)
    | x == aA = let -- we want to REMOVE a from the list.
        -- search through the rest of the list and return y's domain.
        yDomain = head [b' | (a',b')<-as, y==a' ]
        in (aD, yDomain, as)
    | y == aA = let (retX, retAs) = popX x as in (retX, aD, a:retAs)
    | otherwise = let (retX, retY, retAs) = popXy x y as in (retX, retY, a:retAs)
```

We now come to the main part of the algorithm. The iterateAC3 function runs as long as the queue of constraints is not empty, starting with the original set of constraints. We get the domains of X & Y, and remove the *old* domains of X. We then run checkDomain, and add the new domain of X back to the list of domains. If X's domain was altered, then we add all constraints of the form (Y,X) to the back of the queue.

```
ac3 :: (Ord a, Ord b) => AC3 a b -> [Domain a b] -- return a list of domains.
ac3 m@(AC3 c d) = let
    queue = c -- put each constraint into the queue.  -- TODO: implement this better, eg a
        proper queue?
    in iterateAC3 m queue d

iterateAC3 :: (Ord a, Ord b) => AC3 a b -> [ConstraintAA a b] -> [Domain a b]
                -> [Domain a b]
iterateAC3 _ [] d = d
iterateAC3 m@(AC3 fullCS _) ((x,y,c):cs) d = let
    (xDomain, yDomain, alteredD) = popXy x y d
    (newX, str) = runWriter $ checkDomain xDomain yDomain c
        -- In a lens, we could do this with "modify (\ (a,_) -> (a, newX))"
    newDomains = (x, newX) : alteredD
    -- take all constraints of the form (y,x, c)
    z = if null str then cs else cs ++ [c' | c'@(y1,x1,_)<-fullCS, y1/=y && y1/=x, x1==x ]
    in iterateAC3 m z newDomains
```

2.2 The Backtracking library

Using our AC3 instances, we now define a backtracking method to find one or all solutions (where possible) for a given instance. We start by defining the 'output' of our backtracking method, which will be a list `[Assignment a b]`.

```
module Backtracking where

import AC3Solver

type Assignment a b = (Agent a, b)
```

First of all, we can provide a fast method to check that a solution is even *theoretically* possible: if at least 1 agent has an empty domain, then there will never be a legal assignment.

```
--Returns true iff at least 1 agent has an empty domain.
--Post: Returns true -> \not \exist a solution.
--      However, returns false does NOT guarantee that a solution exists.
determineNoSol :: [Domain a b] -> Bool
determineNoSol = any (\(_,ds) -> null ds)
```

Next, we use backtracking to try and find a solution, using backtracking. For our agent X, we iterate over each value in X's domain. For every constraint (X,Y) or (Y,X), where Y has already got an assigned value, we check if this constraint holds. If at least one of these constraints does not hold, then we continue with the next value in X's domain. Else, we continue with the next agent. If we find a valid assignment `Just ...`, then we return this, else we try the next value in X's domain.

If no value in X's domain leads to a valid assignment, we return `Nothing`, and try a different assignment, or return Nothing if no solution exists for this instance.

Notably, while findSolution takes an instance of AC3, we can run `findSolution` *without* having run `ac3`, and so we can compare the runtime of findSolution before and after running `ac3`.

```
findSolution :: Eq a => AC3 a b -> Maybe [Assignment a b]
findSolution (AC3 c d) = helpFS c d []

helpFS :: Eq a => [ConstraintAA a b] -> [Domain a b] -> [Assignment a b] -> Maybe [
    Assignment a b]
helpFS _ [] as = Just as -- Done
helpFS constrs ((x, ds):dss) as = recurseFS ds where
    recurseFS [] = Nothing
    recurseFS (d:ds') = let
        -- we want to try assigning value d to agent x.
        -- Get all constraints (X,Y) and (Y,X), where Y already has a value assigned to it.
        -- Check if x=d works, for all previously assigned values Y.
        checkCons = and $
            [cf d (valY y as) | (x',y,cf)<-constrs, x==x', y `elemAs` as] ++
            [cf (valY y as) d | (y,x',cf)<-constrs, x==x', y `elemAs` as]
        in
        if not checkCons then recurseFS ds' -- easy case, x=d is not allowed.
        else --
            case helpFS constrs dss ((x,d):as) of
                Nothing -> recurseFS ds' -- x=d causes issues later on.
                Just solution -> Just solution -- :)
```

As with `findSolution`, `findAllSolutions` returns the (possibly empty) list of all solutions, again using backtracking.

```
-- find all
findAllSolutions :: Eq a => AC3 a b -> [[Assignment a b]]
findAllSolutions (AC3 c d) = helpFSAll c d []

-- helper function for find all
helpFSAll :: Eq a => [ConstraintAA a b] -> [Domain a b] -> [Assignment a b] -> [[Assignment
     a b]]
helpFSAll _ [] as = [as]  -- Found a complete solution
helpFSAll constrs ((x, ds):dss) as = concatMap recurseFS ds where
    recurseFS d =
        let checkCons = all (\(x', y, cf) -> not (x == x' && y `elemAs` as) || cf d (valY y
             as)) constrs
                     && all (\(y, x', cf) -> not (x == x' && y `elemAs` as) || cf (valY y
                         as) d) constrs
        in if checkCons then helpFSAll constrs dss ((x,d):as) else []
```

Given a solution, verify whether this solution is permissible with the provided constraints.

```
checkSolution :: Eq a => [ConstraintAA a b] -> [Assignment a b] -> Bool
checkSolution [] _ = True
checkSolution ((x,y,f):cs) as = elemAs x as && elemAs y as && let
    xN = valY x as
    yN = valY y as
    in f xN yN && checkSolution cs as
```

Help-functions used by our solution methods.

```
-- Find whether agent Y has an assignment.
elemAs :: Eq a => Agent a -> [Assignment a b] -> Bool
elemAs _ [] = False
elemAs y ((x,_):as) = x==y || y `elemAs` as

-- Find agent Y's assigned value
-- PRE: y \in as.
valY :: Eq a => Agent a -> [Assignment a b] -> b
valY _ [] = undefined -- should not happen.
valY y ((x,b):as) = if x == y then b else valY y as
```

3 The problem files

3.1 The NQueens library

The NQueens module defines a constraint satisfaction problem where we place N queens on an $N \times N$ chessboard so that no two queens attack each other.

```
module NQueens where

import AC3Solver ( ac3, AC3(AC3) )  -- Import AC3 solver
import Backtracking (findSolution, findAllSolutions ) -- Import backtracking solver

notSameQueenMove :: (Int, Int) -> (Int, Int) -> Bool
notSameQueenMove (a1, a2) (b1, b2) =
    not (a1 == b1 || a2 == b2 || abs (a1 - b1) == abs (a2 - b2))

(//=) :: (Int, Int) -> (Int, Int) -> Bool
(a1, a2) //= (b1, b2) = notSameQueenMove (a1, a2) (b1, b2)
```

The `nQueens` function encodes the N-Queens problem as a constraint satisfaction problem. The domain is defined in such a way that exactly one queen must be placed in each row. Constraints are generated using list comprehension together with the custom infix function `(//=)`, which ensures that no two queens share the same row, column, or diagonal.

```
nQueens :: Int -> AC3 Int (Int, Int)
nQueens n = let
    agents = [0 .. n-1] -- Queens as row numbers
    domain = [(row, [(row, col) | col <- [0 .. n-1]]) | row <- agents] -- 1 queen per row
    constraints = [(a, b, (//=)) | a <- agents, b <- agents, a < b]
    in AC3 constraints domain
```

There are two functions available to solve the problem. The function `solveNQueens` finds a single solution using backtracking. Meanwhile, the function `solveAllNQueens` finds all possible solutions.

```
solveNQueens :: Int -> Maybe [(Int, (Int, Int))]
solveNQueens n = findSolution (AC3 constraints (ac3 (nQueens n)))
  where
    AC3 constraints _ = nQueens n

solveAllNQueens :: Int -> [[(Int, (Int, Int))]]
solveAllNQueens n = findAllSolutions (AC3 constraints (ac3 (nQueens n)))
  where
    AC3 constraints _ = nQueens n
```

The function `prettyPrintBoard` is responsible for printing the board. Solutions are displayed using numbers (`0,1,2,...`) to represent queens, while empty spaces are represented by a dot (`.`).

```
prettyPrintBoard :: Int -> [(Int, (Int, Int))] -> IO ()
prettyPrintBoard n solution = do
    let board = [[if (r, c) `elem` map snd solution then show r else "." | c <- [0 .. n-1]]
          | r <- [0 .. n-1]]
    mapM_ (putStrLn . pptHelper) board
    putStrLn ""

pptHelper :: [String] -> String
pptHelper [] = ""
pptHelper [x] = x
pptHelper (x:xs) = x ++ " " ++ pptHelper xs
```

The `nQueensMain` function provides user interaction by asking for an input value N. It then solves the problem and either prints the full solutions or just the count of solutions, depending on whether prettyPrintBoard is enabled. To start the NQueens program, run stack ghci and then nQueensMain, after which you are prompted to give an integer for N.

```
nQueensMain :: IO ()
nQueensMain = do
    putStrLn "Enter board size (N):"
    n <- readLn
    let solutions = solveAllNQueens n
    -- Uncomment for 1 solution instead
    -- let solutions = solveNQueens n
    if null solutions
    then putStrLn "No solution found."
    else do
        putStrLn "Solutions: "
        -- Comment out if only interested in the number of solutions
        -- mapM_ (prettyPrintBoard n) solutions
        putStrLn $ "Found " ++ show (length solutions) ++ " solution(s)"
```

3.2 The Graph Colouring library

Graph colouring is a well-known NP-Complete problem [GJS74]. Its nature as a graph problem lends it well to being modelled as an AC3 instance, and then being solved using our backtracking functions.

A problem instance consists of an undirected graph, and an integer $n > 0$. We are asked to assign a colour $0..(n-1)$ to each vertex, where for each edge (u, v), u and v have different colours.

```
--{-# LANGUAGE LambdaCase #-} -- todo remove? if not using data.graph.read...
module GraphCol where

import Control.Monad (when, foldM_)
import Criterion.Main
import Data.Char (toUpper)
import Data.Graph
--import Data.Graph.Read
import Data.Maybe
import Data.List
import Text.Read (readMaybe)
import Test.QuickCheck

import AC3Solver
import Backtracking
import Scheduling (parseInput)
```

We make use of Haskell's Graph library, following in its convention that vertices are numbers, and edges are pairs of vertices.

We define a newtype GraphCol using AC3, where the agents are of type Vertex and the domain is a set of colours $\subseteq$ [0..(n-1)]. All constraints should be of the form (X,Y,(/=)), and this represents an edge (X,Y) in the graph.

We define arbitrary instances for GraphCol using following these conventions.

```

-- We define a newtype, so that we can generate arbitrary instances.
newtype GraphCol = GC (AC3 Vertex Int)

seqPair :: (Gen a, Gen a) -> Gen (a,a)
seqPair (ma, mb) = ma >>= \a -> mb >>= \b -> return (a,b)

-- (Seems graphs don't already have an arbitrary instance...)
instance Arbitrary GraphCol where
    arbitrary = sized arbitGraphColN where
        arbitGraphColN n = do
```

```haskell
            nColours <- chooseInt (2, max (n `div` 4) 2) -- we require n to be > 0
            --sizeV <- choose (0, n `div` 3) -- we make vertices 0..sizeV INCLUDING SIZEV!
            let sizeV = 999
            let eMax = max sizeV $ (sizeV*(sizeV-1)) `div` 4
            sizeE <- chooseInt (sizeV, eMax)
            e <- sequence [seqPair (chooseInt (0, sizeV), chooseInt (0, sizeV)) | _<-[0..
                sizeE]]
            -- we do not want edges (x,x), nor do we want repeat edges
            let nonReflE = nub $ filter (uncurry (/=)) e
            let g = buildG (0, sizeV) nonReflE
            return $ convertGraphToAC3 g nColours --return $ convertGraphToAC3 g n

instance Show GraphCol where
  --show :: GraphCol -> String
  show (GC (AC3 c d)) = let
    strCon = "[" ++ makeShow c ++ "]" where
      makeShow [] = ""
      makeShow ((x,y,_):cs) =
        "(" ++ show x ++ ", " ++ show y ++ ", (/=))"
        ++ if not $ null cs then ", " ++ makeShow cs else ""
    strD = show d
    in "GC (AC3 " ++ strCon ++ " " ++ strD ++ " )"
```

We define a method to convert a graph into an instance of `GraphCol`, and vice versa. Note that graph colouring concerns *un*directed graphs while the Graph library concerns *directed* graphs. As a result, `g == ac3ToGraph $ convertGraphToAC3 g n` (for any $n > 0$) is NOT guaranteed to hold.

```haskell
-- NOTE: The Graph library uses *directed* graphs.
--        We add both (x,y,/=) and (y,x,/=), as graph colouring concerns Undirected graphs.
-- Create an instance with colours [0..(n-1)]
-- PRE: n >= 1
convertGraphToAC3 :: Graph -> Int -> GraphCol
convertGraphToAC3 g n = let
    agents = vertices g
    constr = [(x,y, (/=)) | (x,y)<-edges g]
    in GC $ AC3
        (constr ++ reverseCons constr)
        -- In graph colouring, we want to check both X's domain to Y, and Y's to X.
        ((head agents, [0]) : [(a, [0..(n-1)]) | a<-tail agents])

-- Help function: If we have an edge (x,y), we need both (x,y, /=) and (y,x,/=) as
    constraints.
reverseCons :: [(a,b,c)] -> [(b,a,c)]
reverseCons = map (\ (a,b,c) -> (b,a,c))

ac3ToGraph :: GraphCol -> Graph
ac3ToGraph (GC (AC3 c d)) = let
    v = [a | (a,_)<-d]
    e = nub [ (a,b) | (a,b,_)<-c] -- If we originally had (x,y) AND (y,x) in our graph,
        then c contains each twice.
    in buildG (foldr min 0 v, foldr max minBound v) e
```

We provide a section of code that may optimise the `GraphCol` instance. We assign the colour 0 to the vertex 0, as in graph colouring we can arbitrarily assign a colour to the 'first' vertex.

However, if the graph consists of multiple disconnected components, then we can do such an arbitrary assignment to a vertex in each separate component, thereby reducing the search space.

```haskell
optimiseGC :: GraphCol -> GraphCol
optimiseGC gc@(GC (AC3 c d)) = let
  comps = components $ ac3ToGraph gc
  -- As far as I can find with the tests, if 0 is an element of a component, then
  --  0 is at the root. (Assuming a normal, legal GC instance of course).
  -- We assume this is the case. For each component, we assign the reduced domain [0]
  -- to the root, thereby reducing the search space.
  dChanges = map (\(Node r _) -> r) comps
```

```
  in GC (AC3 c (map (\(a,b) -> if a `elem` dChanges then (a,[0]) else (a,b)) d))
```

The actual main part of the programme, for Graph Colouring:

```
getGraphChoice :: IO Int
getGraphChoice = do
  putStr "Choose one of the following options: \n\
         \1: Read in a graph colouring instance from the terminal \n\
         \2: Read in a graph colouring instance from a file \n\
         \3: Run benchmarks \n"
  choice <- getLine
  case readMaybe choice of
    Nothing -> do
      putStrLn "Invalid choice, please try again."
      getGraphChoice
    Just n ->
      if n > 0 && n < 4 then return n else do
        putStrLn "Invalid choice, please try again."
        getGraphChoice

graphColMain :: IO ()
graphColMain = do
    choice <- getGraphChoice
    case choice of
        1 -> terminalGraph
        2 -> fileGraph
        3 -> benchmarkTests
        _ -> undefined

-- PRE: m <= n.
getEdges :: Int -> Int -> IO [Edge]
getEdges m n
  | m == n = return []
  | otherwise = do
      putStrLn $ "Edge " ++ show m
      x <- parseInput "Enter the first vertex: "
      y <- parseInput "Enter the second vertex: "

      rest <- getEdges (m+1) n
      return $ (x,y) : rest

terminalGraph :: IO ()
terminalGraph = do
  nVertices <- parseInput "Enter the number of vertices: "
  putStrLn $ "Okay, we number the vertices from 0 to " ++ show (nVertices-1)
  -- TODO: Error handling, eg. if nVertices <= 0 ?
  nEdges <- parseInput "Enter the number of edges: "
  eList <- getEdges 0 nEdges
  let graph = buildG (0, nVertices-1) eList
  nColours <- parseInput "Enter the number of colours: "
  let g = convertGraphToAC3 graph nColours
  putStrLn $ "We run AC3 on this instance: " ++ show g
  runGraph g


-- Given a graphcol instance, we run AC3 on it. If we have at least 1 solution (after back
    prop.),
--  show it to the user,  and ask if they want to see all solutions.
runGraph :: GraphCol -> IO ()
runGraph (GC ac3Inst) = do
  let ac3Domain = ac3 ac3Inst
  if determineNoSol ac3Domain
    then putStrLn "AC3 has found an empty domain for at least 1 agent -> No solution"
    else do
      putStrLn "AC3 has at least 1 option for each agent."
      case findSolution ac3Inst of
        Nothing -> putStrLn "There is no solution based on the reduced AC3 input."
        Just sol -> do
          putStrLn $ "We have found a solution: " ++ show sol
          putStrLn "Do you want to find out how many different solutions we have? (Y/N) "
          choice <- getLine
          when (toUpper (head choice) == 'Y') $ do
```

```
            let allSols = findAllSolutions ac3Inst
            if length allSols == 1 then putStrLn "There is only 1 solution."
            else do -- it should not be possible to reach here if allSols = 0.
              putStrLn $ "There are " ++ show (length allSols) ++ " different solutions. \
                  nDo you want to see them? (Y/N) "
              choice2 <- getLine
              when (toUpper (head choice2) == 'Y') $ mapM_ print allSols
```

PROBABLY TODO REMOVE (but nice to break up)

```
readGraphFromFile :: String -> IO GraphCol
readGraphFromFile filename = do
  filecon <- readFile filename
  let fileInput = words filecon
  let nVertices = read $ head fileInput
  let nEdges = read $ head (tail fileInput)
  let edgeList = makeEdges nEdges (drop 2 fileInput) -- includes nColours
  let nColours = read $ last fileInput
  let graph = buildG (0, nVertices-1) edgeList
  let g = convertGraphToAC3 graph nColours
  return g

fileGraph :: IO ()
fileGraph = do
  putStrLn "We expect the file to be of the following format: \n\
            \[number of vertices] \n\
            \[number of edges] \n\
            \for each edge: [vertex 1] [vertex 2] of the edge. \n\
            \[the number of colours > 0.]\n"
  putStrLn "Provide the file name:"
  filename <- getLine
  {-
  filecon <- readFile filename
  let fileInput = words filecon
  let nVertices = read $ head fileInput
  let nEdges = read $ head (tail fileInput)
  let edgeList = makeEdges nEdges (drop 2 fileInput) -- includes nColours
  let nColours = read $ last fileInput
  let graph = buildG (0, nVertices-1) edgeList
  let g = convertGraphToAC3 graph nColours
  -}
  g <- readGraphFromFile filename
  putStrLn $ "We run AC3 on this instance: " ++ show g
  runGraph g

-- PRE: n >= 0.
makeEdges :: Int -> [String] -> [Edge]
makeEdges 0 _ = []
-- if n > 0, but we have run out of edges -> fail
makeEdges _ [] = undefined
makeEdges _ [_] = undefined
makeEdges n (x:y:es) = (read x, read y) : makeEdges (n-1) es


graphFileFormat :: GraphCol -> IO ()
graphFileFormat (GC (AC3 c d)) = do
  --let nVertices = (fst . last . sort) d
  -- nVertices
  -- succ, as we want vertices from (0..n-1)
  (print . succ . fst . maximum) d
  --let nEdges = length c
  -- nEdges
  (print . length) c
  -- print each edge
  foldM_ ( \_ (x,y,_) -> putStrLn $ show x ++ " " ++ show y) () c
  -- print nColours
  print . succ $ foldr (\(_,ds) x -> foldr max x ds) 0 d
```

PROBABLY TODO REMOVE (but nice to break up)

```haskell
-- A method to note the difference before & after running AC3.
getTotalDomainOptions :: [Domain a b] -> Int
getTotalDomainOptions = foldr (\(_, ds) prev -> length ds + prev) 0

testFiles :: [String] --TODO: Automate this for all files in /lib
testFiles = map ("graphcolInstances/"++)
  ["n10e16nc14.txt", "n10e18nc9.txt", "n10e22nc2.txt", "n15e16nc2.txt",
   "n15e38nc6.txt", "n15e44nc4.txt",
   "n20e96nc20.txt", "n20e188nc6.txt",
   "n25e110nc15.txt", "n25e134nc22.txt"]

-- Specific files with loose vertices / 2 separate components
testFilesComps :: [String]
testFilesComps = map ("graphcolInstances/"++)
  ["n10e26nc7_Single1.txt", "n20e26nc7_Single2.txt", "n20e52nc7_Comps.txt"]

testFilesComps3 :: [String]
testFilesComps3 = map ("graphcolInstances/"++)
  ["n10e26nc3_Single1.txt", "n20e26nc3_Single2.txt", "n20e52nc3_Comps.txt"]

benchmarkTests :: IO ()
benchmarkTests = mapM_ runBenchmark $ testFiles ++ testFilesComps ++ testFilesComps3 ++ ["
    graphcolInstances/n10e40nc3_Neg.txt"]

runBenchmark :: String -> IO ()
runBenchmark filename = do
  gc@(GC inst) <- readGraphFromFile filename
  let origNOpts =  getTotalDomainOptions $ domains inst
  let newD = ac3 inst
  let newNOpts = getTotalDomainOptions newD

  let (GC optiInst) = optimiseGC gc
  let newOptiD = ac3 optiInst
  let newNOptiD = getTotalDomainOptions newOptiD

  putStrLn $ "Filename: " ++ filename
  putStrLn $ "Pre AC-3:        " ++ show origNOpts
  putStrLn $ "Post AC-3:       " ++ show newNOpts
  putStrLn $ "OptimiseGC:      " ++ (show . getTotalDomainOptions . domains) optiInst
  putStrLn $ "OptimiseGC AC-3: " ++ show newNOptiD

  -- Benchmark Criterion bit
  defaultMain [
    bgroup filename [ bench "pre AC-3" $ whnf findSolution inst
                    , bench "post AC-3" $ whnf findSolution (AC3 (cons inst) (ac3 inst))
                    , bench "OptimiseGC, no AC-3" $ whnf (\(GC oi) -> findSolution oi) (
                      optimiseGC gc)
                    , bench "OptimiseGC, + AC-3 " $ whnf (\(GC oi) -> findSolution (AC3 (
                      cons inst) (ac3 oi))) (optimiseGC gc)
                    ]
    ]
```

```haskell
-- | Given a graph, we want to duplicate it so that we have 2 components,
-- | where every vertices 1,3,... form 1 copy of the graph, and 0,2,4.. the other
duplicateGraph :: Graph -> Graph
duplicateGraph g = let
  mappingEven = zip (vertices g) [0 :: Int, 2..]
  mappingOdd = zip (vertices g) [1 :: Int, 3..]
  newVcount = 2*length (vertices g)  - 1
  newEdges = concatMap (\(a,b) -> [ (lookupJust a mappingEven, lookupJust b mappingEven) ,
       (lookupJust a mappingOdd, lookupJust b mappingOdd)]) (edges g)
  in buildG (0, newVcount) newEdges
```

3.3 The Scheduling library

```haskell
module Scheduling where
```

```haskell
import Text.Parsec
import Text.Parsec.String
import Control.Monad (replicateM)
import Data.List (elemIndex)
import AC3Solver
import Backtracking (findSolution)

type ClassAssignment = (Int, Int, Int)
dayNames :: [String]
dayNames = ["monday", "tuesday", "wednesday", "thursday", "friday"]

parseInt :: Parser Int
parseInt = do
  spaces
  n <- many1 digit
  return (read n)

parseInput :: String -> IO Int
parseInput prompt = do
  putStrLn prompt
  read <$> getLine

getNames :: String -> Int -> IO [String]
getNames prompt n = do
  putStrLn prompt
  replicateM n getLine

getUserInputs :: IO (Int, Int, Int, [String], [String], [String])
getUserInputs = do
  numClasses <- parseInput "Enter the number of classes:"
  classNames <- getNames "Enter class names:" numClasses
  numRooms <- parseInput "Enter the number of rooms:"
  roomNames <- getNames "Enter room names:" numRooms
  numTimeSlots <- parseInput "Enter the number of time slots per day:"
  timeSlotNames <- getNames "Enter time slot names:" numTimeSlots
  return (numClasses, numRooms, numTimeSlots, classNames, roomNames, timeSlotNames)

testUserInputs :: IO ()
testUserInputs = do
  (numClasses, numRooms, numTimeSlots, classNames, roomNames, timeSlotNames) <-
      getUserInputs

  putStrLn "\nCollected Inputs:"
  putStrLn $ "Number of Classes: " ++ show numClasses
  putStrLn $ "Class Names: " ++ show classNames
  putStrLn $ "Number of Rooms: " ++ show numRooms
  putStrLn $ "Room Names: " ++ show roomNames
  putStrLn $ "Number of Time Slots per Day: " ++ show numTimeSlots
  putStrLn $ "Time Slot Names: " ++ show timeSlotNames

checkSameDay :: ClassAssignment -> ClassAssignment -> Bool
checkSameDay (x,_,_) (y,_,_) = x == y

checkBefore :: ClassAssignment -> ClassAssignment -> Bool
checkBefore (x,x2,_) (y,y2,_) = x == y && x2 + 1 == y2

checkAfter :: ClassAssignment -> ClassAssignment -> Bool
checkAfter (x,x2,_) (y,y2,_) = x == y && x2 == y2 + 1

checkDay :: Int -> ClassAssignment -> Bool
checkDay a (x,_,_) = x == a

checkTime :: Int -> ClassAssignment -> Bool
checkTime a (_,x,_) = x == a

checkRoom :: Int -> ClassAssignment -> Bool
checkRoom a (_,_,x) = x == a

filterDomains :: [Domain Int ClassAssignment] -> [(Agent Int, ClassAssignment -> Bool)] ->
    [Domain Int ClassAssignment]
filterDomains domainList conditions =
    [(agent, [v | v <- values, all (\(a, f) -> (a /= agent) || f v) conditions]) | (agent,
        values) <- domainList]
```

```
getConstraint :: [String] -> IO (Maybe [ConstraintAA Int ClassAssignment])
getConstraint classNames = do
  putStrLn "Enter a constraint (e.g., 'class1 is before class2' or 'class1 is the same day
      as class2'). Type 'Done' to finish:"
  input <- getLine
  if input == "Done" then return Nothing else do
    let parts = words input
    case parts of
      [class1, "is", "before", class2] -> Just <$> processConstraints class1 class2 "is
          before" classNames
      [class1, "is", "the", "same", "day", "as", class2] -> Just <$> processConstraints
          class1 class2 "is the same day as" classNames
      _ -> do
        putStrLn "Invalid input format"
        getConstraint classNames

processConstraints :: String -> String -> String -> [String] -> IO [ConstraintAA Int
    ClassAssignment]
processConstraints class1 class2 keyword classNames = do
  case (elemIndex class1 classNames, elemIndex class2 classNames) of
    (Just i, Just j) -> case keyword of
      "is before" -> return [(i, j, checkBefore), (j, i, checkAfter)]
      "is the same day as"  -> return [(i, j, checkSameDay), (j, i, checkSameDay)]
      _           -> error "Invalid keyword"
    _ -> error "Invalid class names"

collectConstraints :: [String] -> IO [ConstraintAA Int ClassAssignment]
collectConstraints classNames = do
  let loop acc = do
        constraint <- getConstraint classNames
        case constraint of
          Nothing -> return acc
          Just cs  -> loop (cs ++ acc)
  loop []

getStartingValues :: [String] -> [String] -> [String] -> IO (Maybe (Agent Int,
    ClassAssignment -> Bool))
getStartingValues classNames roomNames timeslotNames = do
  putStrLn "Enter known values (e.g., 'class1 is in room3', 'class1 is at 11am' or 'class1
      is on monday'). Type 'Done' to finish:"
  input <- getLine
  if input == "Done" then return Nothing else do
    let parts = words input
    case parts of
      [class1, "is", "in", room] -> Just <$> processStartingValues class1 room "is in"
          classNames roomNames
      [class1, "is", "at", time] -> Just <$> processStartingValues class1 time "is at"
          classNames timeslotNames
      [class1, "is", "on", day] -> Just <$> processStartingValues class1 day "is on"
          classNames dayNames
      _ -> do
        putStrLn "Invalid input format"
        getStartingValues classNames roomNames timeslotNames

processStartingValues :: String -> String -> String -> [String] -> [String] -> IO (Agent
    Int, ClassAssignment -> Bool)
processStartingValues class1 value keyword classNames valueNames = do
  case (elemIndex class1 classNames, elemIndex value valueNames) of
    (Just i, Just j) -> case keyword of
      "is in" -> return (i, checkRoom j)
      "is at" -> return (i, checkTime j)
      "is on" -> return (i, checkDay j)
      _           -> error "Invalid keyword"
    _ -> error "Invalid name"

collectStartingValues :: [String] -> [String] -> [String] -> IO [(Agent Int,
    ClassAssignment -> Bool)]
collectStartingValues classNames roomNames timeslotNames = do
  let loop acc = do
        value <- getStartingValues classNames roomNames timeslotNames
        case value of
          Nothing -> return acc
```

```haskell
            Just svs   -> loop (svs : acc)
  loop []

printSolution :: [String] -> [String] -> [String] -> [String] -> [(Agent Int,
    ClassAssignment)] -> IO ()
printSolution classNames days roomNames timeSlotNames list = putStrLn $ concat
  [classNames !! agent ++ " is scheduled on " ++ days !! dayId ++
    " in " ++ roomNames !! roomId ++ " at " ++ timeSlotNames !! timeId ++ ".\n"
  | (agent, (dayId, roomId, timeId)) <- list]

schedulingMain :: IO ()
schedulingMain = do
  (numClasses, numRooms, numTimeSlots, classNames, roomNames, timeSlotNames) <-
      getUserInputs
  constraints <- collectConstraints classNames

  let uniquenessConstraints = [(i, j, (/=)) | i <- [0..numClasses-1], j <- [0..numClasses
      -1], i /= j]
  let allConstraints = constraints ++ uniquenessConstraints
  let classDomains = [(i, [(d, t, r) | d <- [0..5], t <- [0..numTimeSlots-1], r <- [0..
      numRooms-1]]) | i <- [0..numClasses-1]]
  domainConditions <- collectStartingValues classNames roomNames timeSlotNames
  let filteredDomains = filterDomains classDomains domainConditions

  let possibleSolutions = ac3 AC3 { cons = allConstraints, domains = filteredDomains }

  let solution = findSolution AC3 { cons = allConstraints, domains = possibleSolutions }

  case solution of
    Nothing -> putStrLn "No solution found."
    Just sol -> printSolution classNames dayNames roomNames timeSlotNames sol
```

```haskell
module Sudoku where

-- General imports
import Data.List ( intercalate )
import Data.Time.Clock ( getCurrentTime, diffUTCTime )
import Text.Printf ( printf )

-- Import AC3 solver and backtracking algorithm
import AC3Solver ( AC3 (..), ac3, ConstraintAA, Domain )
import Backtracking ( findSolution )
```

This file implements Sudoku in a suitable format for our AC3 and backtracking algorithms.

In our formulation:

1. Each cell on the Sudoku board is represented as an *Agent* with its associated domain - An agent is identified by a coordinate (i, j) where i is the row $[1-9]$ and j is the column $[1-9]$ - Each agent maintains a domain of possible values $[1-9]$

2. Sudoku's rules are encoded as binary constraints between agents: - **Row constraint**: All cells in the same row must contain different values - **Column constraint**: All cells in the same column must contain different values - **Box constraint**: All cells in the same 3-by-3 box must contain different values

These constraints are implemented as inequality relations ($\neq$) between cells. For instance, cell $(3, 2)$ and cell $(3, 7)$ are in the same row, thus, a constraint is added to ensure that they do not have the same value.

Below is the definition for a list of all cells, and the conditions for two cells being on the same row and in the same column.

```haskell
allCells :: [(Int, Int)]
allCells = [(i,j) | i <- [1..9], j <- [1..9]]

sameRow :: (Int, Int) -> (Int, Int) -> Bool
```

```
sameRow (x1,_) (y1,_) = x1 == y1

sameCol :: (Int, Int) -> (Int, Int) -> Bool
sameCol (_,x2) (_,y2) = x2 == y2
```

A similar condition can be constructed for two cells being in the same 3-by-3 box.

```
sameBox :: (Int, Int) -> (Int, Int) -> Bool
sameBox (x1,y1) (x2,y2) = (x1 - 1) `div` 3 == (x2 - 1) `div` 3 && (y1 - 1) `div` 3 == (y2 -
    1) `div` 3
```

With these conditions, the constraints can be modelled as a list of inequalities between cells. Two cells receive an inequality if they are distinct and share the same row, column, or box.

```
sudokuConstraints :: [ConstraintAA (Int, Int) Int]
sudokuConstraints =    [(i, j, (/=)) | i <- allCells, j <- allCells, i /= j, i `sameRow` j]
                    ++ [(i, j, (/=)) | i <- allCells, j <- allCells, i /= j, i `sameCol` j]
                    ++ [(i, j, (/=)) | i <- allCells, j <- allCells, i /= j, i `sameBox` j]
```

Below is an example of an empty Sudoku board, that is, the domain of every cell is [1..9].

```
sudokuDomains :: [Domain (Int, Int) Int]
sudokuDomains = [(i, [1..9]) | i <- allCells]

sudokuEmpty :: AC3 (Int, Int) Int
sudokuEmpty = AC3 sudokuConstraints sudokuDomains
```

Using the ‘sudokuConstraints‘ as a backbone we can define our own sudoku puzzle. It is quite tedious because it requires us to specify the initial grid. The example below is a Sudoku puzzle with a unique solution.

```
startingCellsUnique :: [Domain (Int, Int) Int]
startingCellsUnique = [ ((1,3), [1]),
    ((1,5), [6]),
    ((1,9), [4]),
    ((2,1), [8]),
    ((2,4), [1]),
    ((2,6), [4]),
    ((2,8), [6]),
    ((3,2), [3]),
    ((3,7), [8]),
    ((3,9), [5]),
    ((4,1), [7]),
    ((4,3), [8]),
    ((4,5), [2]),
    ((4,9), [3]),
    ((5,2), [6]),
    ((5,3), [3]),
    ((5,7), [1]),
    ((5,8), [2]),
    ((5,9), [9]),
    ((6,5), [1]),
    ((7,1), [3]),
    ((7,7), [2]),
    ((7,9), [8]),
    ((8,1), [1]),
    ((8,3), [4]),
    ((8,5), [5]),
    ((8,6), [9]),
    ((8,7), [3]),
    ((9,2), [7]),
    ((9,4), [8]),
    ((9,6), [3]),
    ((9,7), [5]),
    ((9,8), [9]),
    ((9,9), [1])
    ]
```

```
sudokuExampleDomainUnique :: [Domain (Int, Int) Int]   -- | Complete the initial grid by
    adding empty cells
sudokuExampleDomainUnique = startingCellsUnique ++ [(i, [1..9]) | i <- allCells, i `notElem
    ` map fst startingCellsUnique]

sudokuExampleUnique :: AC3 (Int, Int) Int
sudokuExampleUnique = AC3 sudokuConstraints sudokuExampleDomainUnique
```

Instead of specifying our own sudoku puzzles we can leverage the repository of [Ash], in which 100+ puzzles are available. These puzzles are stored as nine rows separated by a newline character, each row containing nine entries. Empty cells are represented by ".".

```
readSudokuFromFile :: FilePath -> IO [Domain (Int, Int) Int]
readSudokuFromFile filePath = do                   -- | filePath is the path to the sudoku
    puzzle file
    contents <- readFile filePath
    let rows = lines contents
    return (parseSudokuDomains rows)

parseSudokuDomains :: [String] -> [Domain (Int, Int) Int]
parseSudokuDomains rows = cellDomains where
    charToDomain :: Char -> [Int]                  -- | Converts characters to domain values
    charToDomain '.' = [1..9]                      -- | Empty cell
    charToDomain c = if c >= '1' && c <= '9'       -- | Should always be true if c!='.', added
        for safety
                        then [read [c]]            -- | Fixed cell
                        else [1..9]                -- | Empty cell

    cellDomains = [((i, j), charToDomain c) |
                  (i, row) <- zip [1..9] (take 9 rows),
                  (j, c) <- zip [1..9] (take 9 row)]
```

Leveraging these functions we can load a sudoku puzzle by specifying its name and return an AC3 instance.

```
loadSudokuPuzzle :: String -> IO (AC3 (Int, Int) Int)
loadSudokuPuzzle fileName = do                     -- | fileName is the name of the sudoku
    puzzle
    let filePath = "sudokuPuzzles/" ++ fileName ++ ".sud"
    cellDomains <- readSudokuFromFile filePath
    return (AC3 sudokuConstraints cellDomains)
```

With the tools above, we can finally define a few different functions that the user can interact with. To start with, we need a function that loads a sudoku puzzle from its file name, runs AC3, and returns the puzzle with its reduced domains.

```
runAC3OnSudokuFile :: String -> IO (AC3 (Int, Int) Int)
runAC3OnSudokuFile fileName = do
    puzzle <- loadSudokuPuzzle fileName            -- | Load sudoku puzzle from file name
    putStrLn "Initial puzzle:"
    printSudokuPuzzle puzzle                       -- | Display the initial puzzle

    putStrLn "Running AC3..."                      -- | Run AC3 and create a new puzzle with
        reduced domains
    let reducedDomain = ac3 puzzle
    let reducedPuzzle = AC3 sudokuConstraints reducedDomain

    let oldDomain = getDomains puzzle              -- | Display the average domain size before
         and after running AC3
    let newDomain = getDomains reducedPuzzle
    let (beforeAC3, afterAC3) = computeDomainReduction oldDomain newDomain
    putStrLn "Average domain size"
    putStrLn $ "    Before AC3: " ++ beforeAC3
    putStrLn $ "    After AC3:  " ++ afterAC3

    return reducedPuzzle
```

The reduction in domain size from running AC3 varies between puzzles. Easier puzzles can experience a reduction of more than 50%, while harder puzzles are more like 40%. However, only getting the domain reduction is unsatisfactory, we want a solution to the sudoku as well. The following function does just that by running the backtracking algorithm over the AC3 reduced puzzle.

```
solveSudokuFromFile :: String -> IO ()
solveSudokuFromFile fileName = do
    reducedPuzzle <- runAC3OnSudokuFile fileName-- | Get the puzzle with reduced domains

    putStrLn "Running backtracking..."           -- | Run backtracking to find a solution
    let solutions = findSolution reducedPuzzle

    let solvedDomain = case solutions of         -- | Check for solutions, extract solved
        domain if found
                Nothing -> []                    -- | No solution found
                Just assignments -> [((row, col), [number]) | ((row, col), number) <-
                    assignments]

    if null solvedDomain
        then putStrLn "No solution was found"
        else do
            let solvedPuzzle = AC3 sudokuConstraints solvedDomain
            putStrLn "Solved puzzle:"
            printSudokuPuzzle solvedPuzzle
```

```
solveSudokuFromFileSilent :: String -> IO ()
solveSudokuFromFileSilent fileName = do
    puzzle <- loadSudokuPuzzle fileName
    let ac3Domain = ac3 puzzle
    let ac3Puzzle = AC3 sudokuConstraints ac3Domain
    let solutions = findSolution ac3Puzzle
    let solvedDomain = case solutions of
                Nothing -> []  -- No solution found
                Just assignments -> [((row, col), [number]) | ((row, col), number) <-
                    assignments]

    if null solvedDomain then putStrLn "No solution found"
    else do
        putStrLn "Solved puzzle!"



solveSudokuFromFileNOAC3 :: String -> IO ()
solveSudokuFromFileNOAC3 fileName = do -- 11:36 not by 11:56
    puzzle <- loadSudokuPuzzle fileName
    putStrLn "Initial puzzle"
    printSudokuPuzzle puzzle

    putStrLn "Running backtracking..."           -- | Run backtracking to find a solution
    let solutions = findSolution puzzle

    let solvedDomain = case solutions of         -- | Check for solutions, extract solved
        domain if found
                Nothing -> []                    -- | No solution found
                Just assignments -> [((row, col), [number]) | ((row, col), number) <-
                    assignments]

    if null solvedDomain
        then putStrLn "No solution was found"
        else do
            let solvedPuzzle = AC3 sudokuConstraints solvedDomain
            putStrLn "Solved puzzle:"
            printSudokuPuzzle solvedPuzzle




-- Compute the average domain size before and after running AC3
-- computeReductionFromFile :: String -> IO (Float, Float)
-- computeReductionFromFile fileName = do
```

```
--    puzzle <- loadSudokuPuzzle fileName
--    return (computeDomainReduction puzzle)
```

The code above relied on some pretty-printing and reduction computation, which are defined below.

```
-- Visualize a Sudoku board based on current domains
visualizeSudoku :: [Domain (Int, Int) Int] -> String
visualizeSudoku domains' = unlines (
    horizontalLine : concatMap formatRow [1 .. 9]
  )
  where
    -- Get the domain for a specific cell
    getDomain i j =
        case [d | ((i', j'), d) <- domains', i' == i, j' == j] of
            (d:_) -> d
            [] -> [1..9]  -- Default domain if not specified

    -- Display a cell: single digit if domain has 1 element, empty otherwise
    cellValue i j =
        let domain' = getDomain i j
        in if length domain' == 1
           then show (head domain')
           else " "

    -- Horizontal line patterns
    horizontalLine = "+---+---+---+---+---+---+---+---+---+"
    thickLine = "+===+===+===+===+===+===+===+===+===+"

    -- Format a row with appropriate separators
    formatRow i =
        let rowStr = "| " ++ intercalate " | " [cellValue i j | j <- [1..9]] ++ " |"
            separator = if i `mod` 3 == 0 && i /= 9 then thickLine else horizontalLine
        in [rowStr, separator]

-- Function to print the initial state of a Sudoku puzzle
printSudokuPuzzle :: AC3 (Int, Int) Int -> IO ()
printSudokuPuzzle (AC3 _ domains') = do
    putStrLn (visualizeSudoku domains')

-- Function to run and print a Sudoku solution
printSudokuSolution :: AC3 (Int, Int) Int -> IO ()
printSudokuSolution puzzle = do
    let solution = ac3 puzzle
    putStrLn (visualizeSudoku solution)
```

```
-- Compute the amount of reduction in domain size before and after applying AC-3

computeDomainReduction :: [[Int]] -> [[Int]] -> (String, String)
computeDomainReduction oldDomain newDomain = (printf "%.2f" oldSizeAverage, printf "%.2f"
    newSizeAverage)
  where
    oldSize = sum (map length oldDomain)
    newSize = sum (map length newDomain)
    oldSizeAverage = fromIntegral oldSize / 81 :: Float
    newSizeAverage = fromIntegral newSize / 81 :: Float


-- computeDomainReduction :: AC3 (Int, Int) Int -> (Float, Float)
-- computeDomainReduction puzzle = (fromIntegral originalSize/81, fromIntegral reducedSize
    /81)
--    where
--      originalSize = sum (map length (getDomains puzzle))
--      reducedSize = sum (map (length . snd) (ac3 puzzle)) -- Using AC-3 algorithm to reduce
     the domains



-- Extract the domains of each cell from a sudoku puzzle
getDomains :: AC3 (Int, Int) Int -> [[Int]]
```

```
getDomains (AC3 _cons dom) = map snd dom
```

```
sudokuMain :: IO ()
sudokuMain = do
    showWelcomeMessage

    putStr "Choose your difficulty: \n\
         \  (1) easy\n\
         \  (2) hard\n\
         \  (3) special\n"

    putStr "\nSelect one of (1, 2, 3): "
    diff <- getLine

    fileName <- case diff of
        "1" -> do
            putStr "Choose a puzzle number between 1 and 50: "
            puzzleNum <- getLine
            return ("easy" ++ puzzleNum)

        "2" -> do
            putStr "Choose a puzzle number between 1 and 95: "
            puzzleNum <- getLine
            return ("hard" ++ puzzleNum)

        "3" -> do
            putStr "Choose a puzzle: \n\
                \  (1) impossible\n\
                \  (2) Mirror\n\
                \  (3) Times1\n"
            putStr "\nYour choice: "
            puzzleName <- getLine
            return $ case puzzleName of
                "1" -> "impossible"
                "2" -> "Mirror"
                "3" -> "Times1"
                _   -> "impossible"

        _ -> do
            putStrLn "Invalid choice. Please try again."
            sudokuMain  -- Restart if invalid choice
            return ""   -- This line is never reached but needed for type checking

    -- Print the initial puzzle and the result after applying AC3
    putStr $ "\nSolving Sudoku puzzle " ++ fileName ++ "...\n"

   -- (beforeAC3, afterAC3) <- computeReductionFromFile fileName
   -- putStrLn "The average domain size:"
   -- putStrLn $ "    Before AC3: " ++ show beforeAC3
   -- putStrLn $ "    After AC3: " ++ show afterAC3

    -- Solve the Sudoku puzzle from the file
    solveSudokuFromFile fileName

-- Display welcome banner
showWelcomeMessage :: IO ()
showWelcomeMessage = do
    putStrLn "------------------------------------------"
    putStrLn "|           SUDOKU AC3 SOLVER            |"
    putStrLn "------------------------------------------"
    putStrLn ""
```

Benchmarking the speed of solving sudoku puzzles with AC3 + backtracking VS. only backtracking.

```
timeAC3 :: String -> IO ()
timeAC3 fileName = do
    start <- getCurrentTime
    solveSudokuFromFile fileName
    end <- getCurrentTime
```

```
    putStrLn $ "Time taken: " ++ show (diffUTCTime end start)
```

4 Wrapping it up in an exectuable

TODO

```
module Main where

import Text.Read (readMaybe)

import GraphCol
import Knapsack
import Scheduling
import Sudoku
import NQueens
import ZebraPuzzle

getChoice :: IO Int
getChoice = do
  putStr "Choose one of the following options: \n\
         \1: Graph Colouring \n\
         \2: N-Queens \n\
         \3: Scheduling \n\
         \4: Sudoku \n\
         \5: Zebra Puzzle \n"
  choice <- getLine
  case readMaybe choice of
    Nothing -> do
      putStrLn "Invalid choice, please try again."
      getChoice
    Just n ->
      if n > 0 && n < 6 then return n else do
        putStrLn "Invalid choice, please try again."
        getChoice

main :: IO ()
main = do
  putStrLn "Hello!"
  --print somenumbers
  --print (map funnyfunction somenumbers)
  --myrandomnumbers <- randomnumbers
  --print myrandomnumbers
  --print (map funnyfunction myrandomnumbers)
  --putStrLn "GoodBye"

  -- Get choice
  choice <- getChoice
  case choice of
    1 -> graphColMain
    2 -> nQueensMain
    3 -> schedulingMain
    4 -> sudokuMain
    5 -> zebraPuzzleMain
    _ -> undefined
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

5 The test file(s)

6 AC3 tests

```haskell
module Main where

import AC3Solver
import Backtracking

import Data.Maybe
import Test.Hspec
import Test.QuickCheck

main :: IO ()
main = hspec $ do
  describe "AC3 Tests" $ do
    -- TODO remove
    --it "Example test" $
    --  ac3 exampleAC3 `shouldBe` [(4,[1,2]),(3,[0,1,2]),(2,[0,1,2]),(1,[0,1,2]),(0,[0])]
    it "Positive example (each agent has non-empty domain) - 1" $
      ac3 exampleAC3 `shouldNotSatisfy` determineNoSol
    it "Positive example (each agent has non-empty domain) - 2" $
      ac3 exampleAC3_2 `shouldNotSatisfy` determineNoSol
    it "Positive example (each agent has non-empty domain) - 3" $
      ac3 exampleAC3_GFG `shouldNotSatisfy` determineNoSol

    it "Positive example (at least 1 actual solution) - 1" $ do
      let newD = ac3 exampleAC3
      findSolution (AC3 (cons exampleAC3) newD) `shouldSatisfy` isJust
    it "Positive example (at least 1 actual solution) - 2" $ do
      let newD = ac3 exampleAC3_2
      findSolution (AC3 (cons exampleAC3_2) newD) `shouldSatisfy` isJust
    it "Positive example (at least 1 actual solution) - 3" $ do
      let newD = ac3 exampleAC3_GFG
      findSolution (AC3 (cons exampleAC3_GFG) newD) `shouldSatisfy` isJust

    it "Negative example (has no solution) - 1" $ do
      let newD = ac3 exampleAC3_bad
      findSolution (AC3 (cons exampleAC3_bad) newD) `shouldBe` Nothing
    it "Negative example (has no solution) - 2" $ do
      let newD = ac3 exampleAC3_triv
      findSolution (AC3 (cons exampleAC3_triv) newD) `shouldBe` Nothing
    it "Negative example (has no solution) - 3" $ do
      let newD = ac3 exampleAC3_no_solution
      findSolution (AC3 (cons exampleAC3_no_solution) newD) `shouldBe` Nothing

    -- The --coverage says these cases are never reached, but that is simply not true lol.
    --  It says this even with these cases, but that notwithstanding: according to the test
         report,
    --  we always get the otherwise case, which would seemingly point to us eventually
        reaching the
    --      [] = undefined case
    let xAgent = ("x", [1 :: Int])
    let yAgent = ("y", [2])
    let d = [xAgent, yAgent]
    it "Test popXy x==a" $ do
      popXy "x" "y" d `shouldBe` ([1], [2], [yAgent])
    it "Test popXy y==a" $ do
      popXy "y" "x" d `shouldBe` ([2], [1], [xAgent])
```

```haskell
-- TEST CASES

exampleAC3 :: AC3 Int Int
exampleAC3 = let
    nColours = 3
    nAgents = 5
    -- we assign a specific starting value to an (arbitrary) node. (TODO: for general
        encoding, if a vertex has no edges, assign an arbit colour.)
    in AC3 [ (a, (a+1) `mod` nAgents, (/=)) | a<-[0..nAgents-1]] ((0, [0]) : [ (a, [0..
```

```
        nColours-1]) | a<-[1..nAgents-1]])


-- A graph is 2-colourable iff it is bipartite iff it has no cycles of odd length.
-- (Such as, this example which is a circle of even length.)
exampleAC3_2 :: AC3 Int Int
exampleAC3_2 = let
    nColours = 2
    nAgents = 6
    -- we assign a specific starting value to an (arbitrary) node.
    in AC3 ([ (a, (a-1) `mod` nAgents, (/=)) | a<-[0..nAgents-1]]++[ (a, (a+1) `mod`
        nAgents, (/=)) | a<-[0..nAgents-1]])
           ((0, [0]) : [ (a, [0..nColours-1]) | a<-[1..nAgents-1]])

-- NOT 2-colourable, as it has an odd cycle (circle of len 5).
exampleAC3_bad :: AC3 Int Int
exampleAC3_bad = let
    nColours = 2
    nAgents = 5
    -- we assign a specific starting value to an (arbitrary) node.
    in AC3 ([ (a, (a-1) `mod` nAgents, (/=)) | a<-[0..nAgents-1]]++[ (a, (a+1) `mod`
        nAgents, (/=)) | a<-[0..nAgents-1]])
           ((0, [0]) : [ (a, [0..nColours-1]) | a<-[1..nAgents-1]])

-- NOT 1-colourable, as it has an edge.
exampleAC3_triv :: AC3 Int Int
exampleAC3_triv = let
    nColours = 1 -- can only be 1-colourable iff cons = [].
    nAgents = 5
    -- we assign a specific starting value to an (arbitrary) node.
    in AC3 ([ (a, (a+1) `mod` nAgents, (/=)) | a<-[0..nAgents-1]])
           ((0, [0]) : [ (a, [0..nColours-1]) | a<-[1..nAgents-1]])

-- Example based on https://www.geeksforgeeks.org/3-coloring-is-np-complete/
-- IS 3-colourable.
exampleAC3_GFG :: AC3 String Int
exampleAC3_GFG = let
    nColours = 3 -- can only be 1-colourable iff cons = [].
    agentsA = ["v", "w", "u", "x"]
    agentsB = [s++"'" | s<-agentsA]
    agents = agentsA ++ agentsB -- does NOT include "B"

    bCons = [("B", a, (/=)) | a<-agents]
    outsideCons = [ (a, a++"'", (/=)) | a<-agentsA ]
    reverseCons = map (\ (a,b,c) -> (b,a,c))
    in AC3 (bCons ++ reverseCons bCons ++
                outsideCons ++ reverseCons outsideCons)
           ( ("B", [0]) : [ (a, [0..nColours-1]) | a <- agents])

-- A problem that should have no solutions
exampleAC3_no_solution :: AC3 Int Int
exampleAC3_no_solution = let
    domains_no_sol = [(0, [1,2]), (1, [1,2]), (2, [1,2])]
    constraints_no_sol = [(0, 1, (/=)), (1, 2, (/=)), (0,2, (/=))]
    in AC3 constraints_no_sol domains_no_sol
```

7 Conclusion

Finally, we can see that [LW13] is a nice paper.

References

[Ash] Ben Ashing. Jabenjy/Sudoku: Completed Sudoku solver written in Haskell as part of an assignment as part of a Functional Programming module.

[GJS74] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, page 47–63, New York, NY, USA, 1974. Association for Computing Machinery.

[LW13] Fenrong Liu and Yanjing Wang. Reasoning about agent types and the hardest logic puzzle ever. *Minds and Machines*, 23(1):123–161, 2013.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.