General purpose AC-3 solver with various applications

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

Saturday 29th March, 2025

Abstract

We have implemented the AC-3 algortihm described in [Mac77] in Haskell, as well as a backtracking method to allow us to solve general constraint satisfaction problems. We have also implemented a variety of NP-hard problems, and shown how we can apply the AC-3 algorithm to them, with varying levels of effectiveness.

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 a problem instance as a graph, where each vertex has a set of possible values, or domain, and where each edge between two vertices is a constraint.

This forms the basis behind the arc consistency algorithms discussed in [Mac77], including the AC-3 algorithm we use in our implementation. The AC-3 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. An `AC3` instance constains a list of constraints `Arc`, and a list of domains. Each `Arc` contains a pair of variables (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 variable, and then a list of values of type `b`.

[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 variables's domain. [Mac77] provides other approaches for achieving path consistency, where you may have ternary (or greater) constraints.

We may have multiple constraints for a pair of variables (X, Y), such as both `(>)` and `(>=)`, or an variable may not be in any constraint. The programme however expects that each variable has exactly 1 (possibly empty) domain specified for it.

Note that we do not define an arbitrary instance for AC3. Instead, we can 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 variables 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 :: [Arc a b],
    -- Assume we have 1 domain list for each variable.
    domains :: [Domain a b] }

type Domain a b = (Variable a, [b])
type Arc a b = (Variable a, Variable a, Constraint b)
type Constraint a = a -> a -> Bool

type Variable a = a
```

For each constraint (X, Y, f), we want to check for each value x in the domain of X whether there is at least one value y 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 variables 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)
-- | POST: Output = (X's domain, the original domain with X's domain removed.)
popX :: Eq a => Variable a -> [Domain a b] -> ([b], [Domain a b] )
popX _ [] = error "No domain found for a Variable X in a constraint." -- 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.)
-- | POST: Output = (X's domain [b], Y's domain [b], the original domain d with X's domain
    removed.)
popXy :: Eq a => Variable a -> Variable a -> [Domain a b] -> ([b], [b], [Domain a b] )
popXy _ _ [] = error "No domains found for both variables X & Y in a constraint." -- 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.

```
-- | Given an legal AC3 instance, this function returns a reduced list of domains, where
-- |    each variable's domain now only contains values which are arc-consistent with
-- |    the set constraints.
-- | POST: for each variable x, x's domain in the output \subseteq x's domain originally.
ac3 :: (Ord a, Ord b) => AC3 a b -> [Domain a b] -- return a list of domains.
ac3 m@(AC3 c d) = let
    queue = c
    in iterateAC3 m queue d

iterateAC3 :: (Ord a, Ord b) => AC3 a b -> [Arc 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
    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 = (Variable a, b)
```

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

```
-- |Returns true iff at least 1 variable 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 variable 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 variable. 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 => [Arc 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 variable 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 solutions
findAllSolutions :: Eq a => AC3 a b -> [[Assignment a b]]
findAllSolutions (AC3 c d) = helpFSAll c d []

-- | helper function for findAllSolutions
helpFSAll :: Eq a => [Arc 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 => [Arc 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 variable Y has an assignment.
elemAs :: Eq a => Variable a -> [Assignment a b] -> Bool
elemAs _ [] = False
elemAs y ((x,_):as) = x==y || y `elemAs` as

-- | Find variable Y's assigned value
-- | PRE: y \in as.
valY :: Eq a => Variable a -> [Assignment a b] -> b
valY _ [] = error "Y's value could not be found in the assignment." -- 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
    variables = [0 .. n-1] -- Queens as row numbers
    domain = [(row, [(row, col) | col <- [0 .. n-1]]) | row <- variables] -- 1 queen per
        row
    constraints = [(a, b, (//=)) | a <- variables, b <- variables, 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.

```
module GraphCol where

import Control.Monad (when, foldM_)
import Criterion.Main
import Data.Char (toUpper)
import Data.Graph
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 variables 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)

-- It appears that Haskell graphs do not already have an arbitrary instance.
instance Arbitrary GraphCol where
    arbitrary = sized arbitGraphColN where
        arbitGraphColN n = do
```

```
            nColours <- chooseInt (1, max (n `div` 4) 1) -- we require n to be > 0
            --nColours <- chooseInt (2, max (n `div` 4) 2)
            sizeV <- choose (0, n `div` 3) -- we make vertices 0..sizeV INCLUDING SIZEV!
            --let sizeV = 498
            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

-- | We require show to use the arbritary instance above in QuickCheck.
-- | Note that we cannot actually check that each constraint is a (/=), we must
-- | assume it to be so.
instance Show GraphCol where
  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.

```
-- | 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
    variables = 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 variables, [0]) : [(a, [0..(n-1)]) | a<-tail variables])

-- | 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))

-- | Given a graph colouring instance, return the graph of it.
-- | POST: Edges contains at most 1 Directed edge (x,y) for all vertices x/=y.
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.

```
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))
```

Benchmarking GraphCol

We set out to benchmark `GraphCol` in 2 different ways: given an instance of `GraphCol`, first, we determine the total number of options across all domains in the instance, and we compare this to the number of options once we have run `ac3`. We then do the same having run `optimiseGC`, with and without running `ac3`.

Next, we run `findSolution` on this instance, as well as on the same instance altered using `ac3` and/or `optimiseGC`, and record the time using Haskell's Criterion library [O'S].

```
-- | Given a file name, run the benchmark suite on the graphcol instance in this file.
runBenchmark :: String -> IO ()
runBenchmark filename = do
  gc@(GC inst) <- readGraphFromFile filename
```

Note: Only 1 of the following 2 code blocks should be run, and the other should be hidden.

```
  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 using Criterion
  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)
                    ]
    ]
```

To get the most fair comparison between the instance before and after running `ac3`, we will discusses instances where there is no solution.[2] In such cases, `findSolution` needs to go through the entire search space, and as such `ac3` is most likely to be effective in reducing the run-time.

We highlight one specific example, but all results can be found in the `benchmark` folder.

We take the `example_N200_AC3` case, which has 200 vertices, 11324 constraints or 10548 unique edges, and 3 colours. By running `ac3` on this instance, we can reduce the total number of options in the domain from 598 $(200 * 3 - 2)$ to 553. Then, running `findSolution`, the runtime

[2]Note that the *ac*3 function can change the order of the domain. As a result, comparing runtimes of *findSolution* or *findAllSolutions* may lead to erroneous results.

decreases from 119ms on the original instance to 45ms running on the reduced instance. This is an instance that greatly benefits from running AC-3, as we see the run-time reducing as a result of the reduced search space.

Generally, it is tough to determine whether AC-3 provides much improvement to graph colouring. Initially, we only have a limitation on vertex 0's colour being set to 0. If we have 1 colour, then AC-3 can solve the instance: if we have 1 or more edges, then the variables with not be 1-colourable. If we have 2 colours, then AC-3 can assign colours to each vertex in the same component as 0. If this is not possible, then we can see that the graph is not 2-colourable.

However for larger N colours, or if the graph contains more than 1 component, the only reduction that AC-3 can make is by removing the option for colour 0 for vertices connected to vertex 0. Similar to the example mentioned in Section 1, AC-3 will not be able to remove options if both vertices have more than 1 colour possible.

3.3 The Scheduling library

This module uses the AC3 algorithm to solve timetable scheduling problems. The user can input the variables and contraints by using either a parser or loading the problem from a text file. The programm then prints a day, room and time assignment for each course that needed to be scheduled.

The choice of input method is made by setting the variable filePath to either "Nothing" for the parser or to a "Just String" containing the path of the input file.

```
filePath :: Maybe String
filePath = Nothing
--filePath = Just "filePath"
```

If the user chooses to use the parser they will be prompted to type both the total number and names of the given courses, rooms and time slots. Moreover it is assumed that the courses can be scheduled monday to friday and that each class will occupy one time slot.

```
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
  map (map toLower) <$> 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)
```

If the user loads the problem from a text file the programm will immediatly return a solution

without user prompts. The file needs to contain two delimiters to seperate the variable names from the constraints and starting values. A valid input file might look like this:

class1 class2 class3 class4 room1 room2 room3 room4 9am 10am 11am 12pm 1pm — CONSTRAINTS — class1 is before class2 class2 is at the same time as class3 class3 is in the same room as class4 class1 is not the same day as class3 class2 is not at the same time as class4 class4 is not in the same room as class1 — STARTING VALUES — class1 is in room1 class2 is at 10am class3 is on monday class4 is in room4

```
readTextFile :: FilePath -> IO ([String], [String], [String], [String], [String])
readTextFile path = do
  content <- readFile path
  let linesOfFile = filter (not . null) (lines content)

  let (headerLines, rest1) = break (isPrefixOf "--- CONSTRAINTS ---") linesOfFile
  let (constraintLines, rest2) = break (isPrefixOf "--- STARTING VALUES ---") (tail rest1)

  if null headerLines || null rest1 || null constraintLines || null rest2
    then error "Invalid file format."
    else case headerLines of
      [classNamesLine, roomNamesLine, timeSlotNamesLine] -> do
        let classNames = words classNamesLine
        let roomNames = words roomNamesLine
        let timeSlotNames = words timeSlotNamesLine
        return (classNames, roomNames, timeSlotNames, constraintLines, tail rest2)
      _ -> error "Invalid header format."
```

The parser for the constraints runs until the user types "Done". Possible constraints are: "is the same day as", "is in the same room as", "is at the same time as" and the corresponding negations. Additionally there is "is before" which defines that two classes are in two adjacent time slots on the same day.

```
getConstraint :: [String] -> IO (Maybe [Arc Int ClassAssignment])
getConstraint classNames = do
  putStrLn "Enter a constraint (e.g., 'class1 is before class2', 'class1 is the same day as
       class2', 'class1 is in the same room as class2' or 'class1 is not at the same time
      as class2'). Type 'Done' to finish:"
  input <- getLine
  let parts = map (map toLower) (words input)
  if input == "Done" || input == "done" then return Nothing else do
    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
      [class1, "is", "at", "the", "same", "time", "as", class2] -> Just <$>
          processConstraints class1 class2 "is at the same time as" classNames
      [class1, "is", "in", "the", "same", "room", "as", class2] -> Just <$>
          processConstraints class1 class2 "is in the same room as" classNames
      [class1, "is", "not", "the", "same", "day", "as", class2] -> Just <$>
          processConstraints class1 class2 "is not the same day as" classNames
      [class1, "is", "not", "at", "the", "same", "time", "as", class2] -> Just <$>
          processConstraints class1 class2 "is not at the same time as" classNames
      [class1, "is", "not", "in", "the", "same", "room", "as", class2] -> Just <$>
          processConstraints class1 class2 "is not in the same room as" classNames
      _ -> do
        putStrLn "Invalid input format"
        getConstraint classNames

processConstraints :: String -> String -> String -> [String] -> IO [Arc Int ClassAssignment
    ]
processConstraints class1 class2 keyword classNames = do
  let class1' = map toLower class1
      class2' = map toLower class2
  case (elemIndex class1' (map (map toLower) classNames), elemIndex class2' (map (map
      toLower) 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)]
```

```
      "is in the same room as"  -> return [(i, j, checkSameRoom), (j, i, checkSameRoom)]
      "is at the same time as"  -> return [(i, j, checkSameTime), (j, i, checkSameTime)]
      "is not the same day as"  -> return [(i, j, checkNotSameDay), (j, i, checkNotSameDay)
          ]
      "is not in the same room as"  -> return [(i, j, checkNotSameRoom), (j, i,
          checkNotSameRoom)]
      "is not at the same time as"  -> return [(i, j, checkNotSameTime), (j, i,
          checkNotSameTime)]
      _           -> error "Invalid keyword"
    _ -> error "Invalid class names"

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

The same constraints can also be loaded from file. They have to be preceded by the delimiter "— CONSTRAINTS —".

```
getFileConstraints :: [String] -> [String] -> [String] -> [String] -> IO [Arc Int
    ClassAssignment]
getFileConstraints classNames _ _ constraints = do
    let processLine line = case words line of
            [class1, "is", "before", class2] -> processConstraints class1 class2 "is before
                " classNames
            [class1, "is", "the", "same", "day", "as", class2] -> processConstraints class1
                 class2 "is the same day as" classNames
            [class1, "is", "at", "the", "same", "time", "as", class2] -> processConstraints
                 class1 class2 "is at the same time as" classNames
            [class1, "is", "in", "the", "same", "room", "as", class2] -> processConstraints
                 class1 class2 "is in the same room as" classNames
            [class1, "is", "not", "the", "same", "day", "as", class2] -> processConstraints
                 class1 class2 "is not the same day as" classNames
            [class1, "is", "not", "at", "the", "same", "time", "as", class2] ->
                processConstraints class1 class2 "is not at the same time as" classNames
            [class1, "is", "not", "in", "the", "same", "room", "as", class2] ->
                processConstraints class1 class2 "is not in the same room as" classNames
            _ -> error "Invalid constraint format"
    concat <$> mapM processLine constraints
```

The user can also enter any room, time and day values that are already known. The possible keywords are: "is in", "is at" and "is on".

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

If the starting values are loaded from file they have to be preceded by the delimiter "— STARTING VALUES —".

```
getFileStartingValues :: [String] -> [String] -> [String] -> [String] -> IO [(Variable Int,
     ClassAssignment -> Bool)]
getFileStartingValues classNames roomNames timeslotNames startValues = do
  let processLine line = case words line of
            [class1, "is", "in", room] -> processStartingValuesFile class1 room "is in"
                classNames roomNames
            [class1, "is", "at", time] -> processStartingValuesFile class1 time "is at"
                classNames timeslotNames
            [class1, "is", "on", day] -> processStartingValuesFile class1 day "is on"
                classNames dayNames
            _ -> error "Invalid starting value format"
  concat <$> mapM processLine startValues

processStartingValuesFile :: String -> String -> String -> [String] -> [String] -> IO [(
    Variable Int, ClassAssignment -> Bool)]
processStartingValuesFile 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"
```

The main function collects all the variables and constraints. Additionally it adds a uniqueness constraints to make sure that no two courses are in the same room at the same time. The function then generates the possible domains for all variables and filters them by the known values that the user entered. The filtered domains and constraints are passed to the AC3 algorithm which reduces the domains. Finally backtracking is used to either find a possible solution or output an error message if it doesn't exist.

```
schedulingMain :: IO ()
schedulingMain = case filePath of
  Nothing -> 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
```

```
  Just path -> do
    (classNames, roomNames, timeSlotNames, fileConstraints, fileStartingValues) <-
        readTextFile path
    constraints <- getFileConstraints classNames roomNames timeSlotNames fileConstraints
    startingValues <- getFileStartingValues classNames roomNames timeSlotNames
        fileStartingValues
    let numClasses = length classNames
    let numRooms = length roomNames
    let numTimeSlots = length timeSlotNames
    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]]
    let filteredDomains = filterDomains classDomains startingValues
    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
```

3.4 The Sudoku Library

```
module Sudoku where

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

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

Sudoku is a popular puzzle that consists of a 9-by-9 grid, divided into 3-by-3 boxes. The goal is to fill the grid with numbers from 1 to 9, such that each number appears exactly once in each row, column, and box. Parts of the implementation are hidden for sake of brevity, but the full code is available in the repository. We will begin by defining the Sudoku board and its constraints.

In our formulation:

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

2. Sudoku's rules are encoded as binary constraints between variables:

   - **Row constraint**: Two cells in the same row must have different values
   - **Column constraint**: Two cells in the same column must have different values
   - **Box constraint**: Two cells in the same 3-by-3 box must have 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.

```
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 :: [Arc (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
```

3.4.1 Loading Sudoku Puzzles

As opposed to 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)

-- | Input is list of strings such as ["53..7....", "6..195...", ...]
parseSudokuDomains :: [String] -> [Domain (Int, Int) Int]
parseSudokuDomains rows = cellDomains where
    -- | Assigns a character to its corresponding domain values
    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!='.',
        included for safety
                     then [read [c]]            -- ^ Value of 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.

```
-- | Input should be a string such as "easy9"
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 experience greater reduction than harder puzzles. 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.

```
-- | Input should be a string such as "easy9"
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
```

3.4.2 Benchmarking AC3 for Sudoku

We could not include a benchmark for solving Sudoku puzzles with and without AC3, as it took too long without it. When we tried to run backtracking on puzzles without first reducing the domains using AC3, it could not solve easy Sudoku puzzles within an hour. As seen in table 1, before applying AC3, there are on average 10^{49} number of board configurations for an easy Sudoku puzzle, and 10^{57} for a hard puzzle. As backtracking makes random guesses in this vast search space, it becomes unpractical. However, when paired with AC3 it can solve both easy

and hard puzzles within minutes.

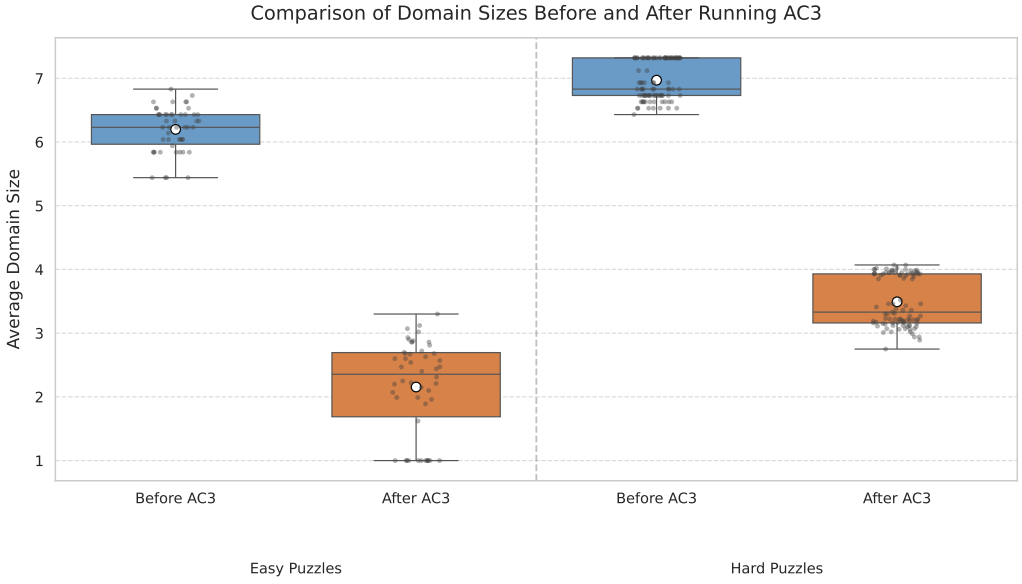
AC3 turns out to be an effective way to significantly reduce the domain size of Sudoku puzzles. It never took more than a couple of seconds to run, and consistently removes 3-4 alternatives from each cell on average, as seen in figure 1. Notably, AC3 manages to *solve* a handful of easy puzzles by reducing the average domain size to one, a surprising feat given that AC3 is not meant to generate solutions. However, this was not the case for any of the harder puzzles.

Table 1: *The Number of Board Combinations Before and After AC3.*

Puzzle Difficulty	Easy		Hard	
	M	SD	M	SD
Before AC3	49.64	3.50	57.44	3.11
After AC3	19.38	11.80	36.95	4.25

Note. Values are given in log base 10. M stands for mean and SD for standard deviation.

Figure 1: *Average Domain Size Before and After AC3.*



3.5 Zebra Puzzle

Other types of problems can be solved using the AC3 algorithm. The Zebra Puzzle, a well known logic puzzle shown below, is an example:

```
There are five houses.
The Englishman lives in the red house.
The Spaniard owns the dog.
Coffee is drunk in the green house.
```

```
The Ukrainian drinks tea.
The Old Gold smoker owns snails.
Kools are smoked in the yellow house.
Milk is drunk in the middle house.
The Norwegian lives in the first house.
The Lucky Strike smoker drinks orange juice.
The Japanese smokes Parliaments.
The green house is immediately to the right of the ivory house.
The man who smokes Chesterfields lives in the house next to the man with the fox.
Kools are smoked in the house next to the house where the horse is kept.
The Norwegian lives next to the blue house
```

Setting up the variables and the domain.

```
houses :: [String]
houses = ["1", "2", "3", "4", "5"]
nationalities :: [String]
nationalities = ["Englishman", "Spaniard", "Ukrainian", "Norwegian", "Japanese"]
colors :: [String]
colors = ["red", "green", "ivory", "yellow", "blue"]
drinks :: [String]
drinks = ["coffee", "tea", "milk", "orange juice", "water"]
smokes :: [String]
smokes = ["Old Gold", "Kools", "Chesterfields", "Lucky Strike", "Parliaments"]
pets :: [String]
pets = ["dog", "snails", "fox", "horse", "zebra"]
agents :: [String]
agents = houses ++ nationalities ++ colors ++ drinks ++ smokes ++ pets
type World = (String, String, String, String, String, String)
type Var = String
permute6 :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [(a, b, c, d, e, f)]
permute6 hl cl drl sl pl nl = [(h, c, dr, s, p, n) | h <- hl, c <- cl, dr <- drl, s <- sl,
    p <- pl, n <- nl]

domainH :: [(Var, [World])]
domainH = [(h, permute6 [h] colors drinks smokes pets nationalities) | h <- houses]
domainC :: [(String, [World])]
domainC = [(c, permute6 houses [c] drinks smokes pets nationalities) | c <- colors]
domainD :: [(String, [World])]
domainD = [(dr, permute6 houses colors [dr] smokes pets nationalities) | dr <- drinks]
domainS :: [(String, [World])]
domainS = [(s, permute6 houses colors drinks [s] pets nationalities) | s <- smokes]
domainP :: [(String, [World])]
domainP = [(p, permute6 houses colors drinks smokes [p] nationalities) | p <- pets]
domainN :: [(String, [World])]
domainN = [(n, permute6 houses colors drinks smokes pets [n]) | n <- nationalities]
d :: [(String, [World])]
d = domainH ++ domainC ++ domainD ++ domainS ++ domainP ++ domainN
```

Only keep the valid parts of the domain. (This is cheating a little as it is not really arc consistency, but it is using that program.)

```
nonEqualPairs :: [[String]]
nonEqualPairs = [["Chesterfields", "fox"], ["Kools", "horse"], ["Norwegian", "blue"], ["
    green", "ivory"]]

equalPairs :: [[String]]
equalPairs = [["Englishman", "red"], ["Spaniard", "dog"], ["coffee", "green"], ["Ukrainian"
    , "tea"], ["Old Gold", "snails"], ["Kools", "yellow"], ["milk", "3"], ["Norwegian", "1"
    ], ["Lucky Strike", "orange juice"], ["Japanese", "Parliaments"]]

validity :: World ->  World -> Bool
validity (h, c, dr, s, p, n) _ =   all ((\ x -> null x || (length x == 2)) . (`ints` [h, c,
     dr, s, p, n])) equalPairs
validityN :: World ->  World -> Bool
validityN (h, c, dr, s, p, n) _ =   all ((\ x -> length x /= 2) . (`ints` [h, c, dr, s, p,
    n])) nonEqualPairs
```

```
constraintVal :: [(String, String,   World  -> World -> Bool)]
constraintVal = [(x, y, validity) | x <- agents, y <- agents, x /= y]

constraintValN :: [(String, String,   World   -> World -> Bool)]
constraintValN = [(x, y, validityN) | x <- agents, y <- agents, x /= y]
```

Constraints for equality, uniqueness, and nonequality.

```
constraintEq :: [(String, String,   World   -> World -> Bool)]
constraintEq = [(x, y, (==)) | [x, y] <- equalPairs] ++ [(x, y, (==)) | [y, x] <-
    equalPairs]
constraintUnique :: [(String, String, World -> World -> Bool)]
constraintUnique = [(x, y, unique) | x <- agents, y <- agents, x /= y]
constraintGi :: [(String, String, World -> World -> Bool)]
constraintGi = [("green", "ivory", greenIvory), ("ivory", "green", flip greenIvory)]
constraintCf :: [(String, String, World -> World -> Bool)]
constraintCf = [("Chesterfields", "fox", nextDoor), ("fox", "Chesterfields", nextDoor)]
constraintKh :: [(String, String, World -> World -> Bool)]
constraintKh = [("Kools", "horse", nextDoor), ("horse", "Kools", nextDoor)]
constraintNb :: [(String, String, World -> World -> Bool)]
constraintNb = [("Norwegian", "blue", nextDoor), ("blue", "Norwegian", nextDoor)]
constraintsNeQ :: [(String, String, World -> World -> Bool)]
constraintsNeQ = [(x, y, notEqual) | [x, y] <- nonEqualPairs] ++ [(x, y, notEqual) | [y, x]
     <- nonEqualPairs] ++ constraintGi ++ constraintCf ++ constraintKh ++ constraintNb

constraints :: [(String, String, World -> World -> Bool)]
constraints = constraintEq ++ constraintsNeQ ++ constraintUnique
```

Finally running the code to solve the puzzle.

```
zebraPuzzleMain :: IO ()
zebraPuzzleMain = do
    let validDomain = ac3 (AC3 (constraintVal ++ constraintValN) d)
    let ac3Puzzle = AC3 constraints validDomain
    let ac3solved = ac3 ac3Puzzle
    let solution = ac3solved
    if null solution
    then putStrLn "No solution found."
    else do
        putStrLn "Solution: "
        print solution
```

4 Running the executable

By running `stack exec myprogram`, we can run the main programme. Here, we can choose which problem instance to work on, at which point the user is sent to the main function for the chosen problem.

```
module Main where

import Text.Read (readMaybe)

import GraphCol
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 "Welcome to our AC-3 solver."

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

5 Conclusion & Future work

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.

[O'S] Bryan O'Sullivan. criterion: Robust, reliable performance measurement and analysis.