Functional Programming & Constraint Programming

Flip Lijnzaad and Jasper Stammes

Friday 4th February, 2022

Abstract

We provide a Haskell representation of constraint satisfaction problems. Moreover, we implement the AC-3 algorithm that facilitates arc-consistency for constraint satisfaction problems. Lastly, we provide an implementation of two constraint satisfaction problems: sudokus, and 3D interpretation of 2D drawings.

goede omschrijving?

Contents

1 Constraint satisfaction problems

A constraint satisfaction problem (CSP) is a triple $\langle X, D, C \rangle$ where

- X is a set of variables $\{x_1, \ldots, x_n\}$;
- D is a set of domains $\{D_1, \ldots, D_n\}$. Each domain is a set of possible values for a variable;
- C is a set of constraints on the domains of the variables.

A constraint is a pair $\langle scope,\ relation \rangle$, where the *scope* is a tuple of the variables that participate in the constraint, and the *relation* is a set of tuples determining the allowed combinations of values for the variables in the *scope*.

Any constraint with a finite scope can be reduced to a set of binary constraints [RN09, p. 206]; this greatly simplifies the representation of constraints, especially for the purposes of implementing algorithms. We call the scope of a binary constraint an *arc*.

In our Haskell implementation of CSPs, we define `type`s and `newtype`s for variables, values, domains, arcs, constraints and CSPs. In the formal definition of a CSP, a domain $D_i \in D$ corresponds to the variable $x_i \in X$; i.e., which domain corresponds to which variable is indicated by their subscripted indices matching. In our implementation, we chose to represent a domain as a tuple of the variable it pertains to, together with the list of possible values, e.g. $\langle x_1, D_1 \rangle$. Since then the set of domains already contains the variables as well, there is no need for including the set of variables in the definition of a CSP. Therefore, in our implementation a CSP is just the pair $\langle D, C \rangle$.

```haskell
{-# LANGUAGE
  GeneralizedNewtypeDeriving
#-}

module CSP where

newtype Variable = Var { getVar :: Int } deriving (Eq, Ord, Num)
instance Show Variable where
  show x = show (getVar x)
newtype Value    = Val { getVal :: Int } deriving (Eq, Ord, Num)
instance Show Value where
  show x = show (getVal x)
type Domain      = (Variable, [Value])
type Arc         = (Variable, Variable)
type Constraint  = ( Arc, [(Value, Value)] )
data Problem     = CSP { domains     :: [Domain]
                       , constraints :: [Constraint] } deriving Show
```

2 Arc consistency

A variable in a CSP is *arc-consistent* if every value in its domain satisfies the variable's binary constraints. A CSP is arc-consistent if every variable in it is arc-consistent with every other variable. Arc consistency is a desirable property of a CSP, since it restricts and thus minimizes the domains of the CSP's variables. An arc-consistent CSP is not necessarily a solved CSP; a

CSP is solved if all constraints are satisfied by any combination of values the variables can take on. A CSP does not have a solution if one variable has an empty domain (i.e. no possible values it can take on).

The AC-3 algorithm reduces a CSP to its arc-consistent version. The algorithm returns true if such an arc-consistent version exists, and it returns false if at any point a variable has an empty domain, i.e. the CSP has no solution. Note that even if AC-3 returns true, this does not necessarily entail that the CSP has a solution. A CSP can be arc-consistent yet have no solution.

3 The AC-3 algorithm

Our functional implementation of the AC-3 algorithm is based on the imperative pseudocode in [RN09, p. 209].

The implementation makes use of the auxiliary function `getVarDomain`. When given a variable and a list of domains, it returns the domain of that variable. It performs `lookup` on the list of domains and returns not only the second argument of the relevant tuple (which is a list of values), but the whole tuple (the whole domain). Moreover, it uses `fromJust` to strip the domain of its `Just`. Generally, this is not safe to do, but in this implementation we are certain that the `lookup` will never return `Nothing`: the use of `getVarDomain` is restricted to cases where we are certain that the variable to look up is actually present in the list.

```
module AC3 where

import CSP
import Data.List
import Data.Maybe -- for using "fromJust"

getVarDomain :: Variable -> [Domain] -> Domain
getVarDomain var doms = let dom = fromJust $ lookup var doms in (var, dom)
```

The `ac3` function takes as input a tuple containing the full CSP, a Boolean flag and a queue of constraints. The Boolean flag is the true or false that the algorithm returns, as described in Section 2. The queue of constraints more or less functions as a to-do list, containing the arcs of which the consistency still needs to be checked. When first calling the function, this queue contains all constraints of the CSP; when the queue of constraints is empty, the CSP is arc-consistent. If at any point during the recursion the Boolean flag is set to false, this means that the domain of a variable is empty and the CSP has no solution. In this case, there is no point in continuing the recursion, so it is halted.

```
ac3 :: (Problem, Bool, [Constraint]) -> (Problem, Bool, [Constraint])
ac3 (p, False, _) = (p, False, [])
ac3 (p, True, []) = (p, True,  [])
```

In the recursive case of the `ac3` function, the first constraint $C = \langle\langle x, y\rangle, R\rangle$ in the constraint queue is considered. The arc $\langle x, y\rangle$ of C is then 'revised' based on this constraint: the domain of x is restricted to those values that satisfy the constraint C. More specifically, the new domain for x, `newXDomain`, is such that those $x' \in D_x$ are kept for which $\exists y' \in D_y$ such that $\langle x', y'\rangle \in R$.

If this revision did not change the domain of x (i.e., $\langle x, y\rangle$ was already arc-consistent), then

the recursion continues with the CSP unchanged, with the flag still set to `True` (because no domains were changed) and with rest of the queue of constraints.

If the revision *did* change the domain of x, then this may cause changes in the domains of the 'neighbors' of x: the arcs of which x is the second argument. To propagate these changes, the neighbors of x are added to the queue (`newQueue`). The domain list is updated (`newDoms`) by deleting the old domain of x from it and adding the new domain of x to it. The new Boolean flag is whether or not x's new domain is empty.

```
ac3 (p@(CSP doms cons), True, ((x, y), rel):queue) =
  if getVarDomain x doms == newXDomain
    then ac3 (p, True, queue)
    else ac3 (CSP newDoms cons, not $ null $ snd newXDomain, newQueue)
  where
    newXDomain = ( x, [ x' | x' <- xvals, any (\y' -> (x', y') `elem` rel) yvals ] ) where
      xvals = snd $ getVarDomain x doms
      yvals = snd $ getVarDomain y doms
    newDoms  = newXDomain : delete (getVarDomain x doms) doms
    newQueue = queue ++ filter (\(arc, _) -> snd arc == x) cons
```

Since the `ac3` function outputs a full CSP, a Boolean flag and an empty queue of constraints, it is useful for practical applications to have a wrapper function that calls `ac3` and only outputs the list of domains of the CSP. Moreover, it is useful to have this list be sorted, since during execution of the AC-3 algorithm, the list of domains has been scrambled. What's more, this function ensures that if the Boolean flag is false, an empty list of domains is returned.

```
arcConsistentDomain :: Problem -> [Domain]
arcConsistentDomain problem@(CSP _ cons) =
  if succeeded
    then
      sortBy (\(a,_) (b,_) -> compare a b) y
    else []
  where
    (CSP y _, succeeded, _) = ac3 (problem, True, cons)
```

4 Sudokus

Sudokus are a well-known constraint satisfaction problem: each square of the 9×9 grid of digits is constrained by the squares in the same row, the same column, and the same 3×3 block. In order to use the AC-3 algorithm on sudokus, the sudoku first needs to be represented as a (binary) constraint satisfaction problem (see Section 1). In order to do so, the variables, domains and constraints of the problem need to be specified.

```
module Sudoku where

import CSP
import AC3
import Data.Char      -- for using "digitToInt"
import Data.Maybe     -- for using "fromJust"
import Control.Monad -- for using "when"
```

Intermezzo: generating sudokus In order to later test our representation of a sudoku as a CSP and subsequently test our implementation of the AC-3 algorithm (and avoid having to type in sudokus manually), we need a program that generates sudokus in plaintext. We have

found a Python script that does just this, and adapted it to fit our program; see Appendix A for its code and a short explanation of its workings.

Although a representation of a square as its coordinates within the grid is a natural one, our definition of a CSP calls for a variable to be an integer. Therefore, we represent the 81 squares of the grid as numbers between 0 and 80, numbered from left to right from top to bottom. As we will see in a bit, for determining the constraints on a square it is useful to use the coordinate notation.

The domain of each empty square of a sudoku (which we represent in our input as `0`) is $\{1, \ldots, 9\}$; the domain of a square filled with some digit x is $\{x\}$. Since a `Domain` in our `CSP` definition also consists of the variable's `Var Int`, the following code also computes the 'index' of the square as a number between 0 and 80.

```
genSudokuDoms :: [Value] -> [Domain]
genSudokuDoms [] = []
genSudokuDoms (x:xs)
    | x == 0    = (Var (80 - length xs), map Val [1..9]) : genSudokuDoms xs
    | otherwise = (Var (80 - length xs), [x])            : genSudokuDoms xs
```

Arguably the most interesting part now is how the constraints for each variable are generated. To be able to formulate the constraints in an intuitive way, the function `varToCoords` takes a variable and returns a tuple of its x- and y-coordinates within the 9×9 grid. `varToCoords` functions as a wrapper around the `varGrid` to eliminate duplicate code later on.

```
varGrid :: [(Variable, (Int, Int))]
varGrid = zip (map Var [0..80]) [ (i,j) | i <- [0..8], j <- [0..8] ]
varToCoords :: Variable -> (Int, Int)
varToCoords n = fromJust $ lookup n varGrid
```

Now, the function `genSudokuCons` (generate sudoku constraints) takes the list of all variables of the sudoku, and returns the list of constraints for the sudoku. It creates this list of constraints by working through the list of variables one by one and generating all constraints for each variable. As said before, each square on the grid is constrained by its row, column and 3×3 block. So a variable n is a member of all arcs $\langle n, x \rangle$ where x is a variable in the same row, column or block. The allowable values for the pair $\langle n, x \rangle$ are then all $y_1, y_2 \in \{1, \ldots, 9\}$ such that $y_1 \neq y_2$.

```
genSudokuCons :: [Variable] -> [Constraint]
genSudokuCons [] = []
genSudokuCons (n:xs) =
  map (\x -> ( (n,x), [(y1,y2) | y1 <- map Val [1..9], y2 <- map Val [1..9], y1 /= y2] ) )
```

The row, column and block constraints are dependent on the position of the variable n within the grid. The following code fragment determines the variables x with which n is participating in a constraint. The variables in the same row as n have the same x-coordinate, and the variables in the same column as n have the same y-coordinate. To obtain the variables in the same 3×3 block as n, we check if the x-coordinates of n and m are the same when divided by 3; we do the same for the y-coordinates.

```
    (
      -- rows
         [m | m <- map Var [0..80], m /= n,
              fst (varToCoords m) == fst (varToCoords n)]
      -- columns
      ++ [m | m <- map Var [0..80], m /= n,
```

```
            snd (varToCoords m) == snd (varToCoords n)]
    -- blocks
    ++ [m | m <- map Var [0..80], m /= n,
            fst (varToCoords m) /= fst (varToCoords n),
            snd (varToCoords m) /= fst (varToCoords n),
            fst (varToCoords m) `div` 3 == fst (varToCoords n) `div` 3,
            snd (varToCoords m) `div` 3 == snd (varToCoords n) `div` 3]
  )
    ++ genSudokuCons xs
```

The list comprehension contains the Boolean condition $m \neq n$ to ensure that there will not be an arc $\langle n, n \rangle$ in the constraints, since there will be no assignment that satisfies the constraint $n \neq n$. Moreover, the list comprehension for the block constraints ensures that variables in the same row or column are ignored, since those have already been taken into account.

The `printSudoku` function takes the list of `Domains` of a sudoku and prints the (partially) solved sudoku in a readable format using spaces and newlines. If the list of possible values for a variable only contains one element, this element may be printed; if it does not, then the value of that variable is as of yet undetermined and an underscore is printed to indicate this.

```
printSudoku :: [Domain] -> IO ()
printSudoku [] = putStr ""
printSudoku ((n, val@(value:_)):xs) =
  do
    -- put the number there if determined, else _
    putStr (if val == [value] then show value else "_")
    -- put spaces between different blocks
    when (getVar n `mod` 3  == 2)  (putStr " ")
    -- put newlines at the end of rows
    when (getVar n `mod` 9  == 8)  (putStr "\n")
    -- put extra newlines to vertically separate blocks
    when (getVar n `mod` 27 == 26) (putStr "\n")
    do printSudoku xs
-- (to avoid warning about non-exhaustive pattern-matching)
printSudoku _ = putStr ""
```

Now the function `ac3SudokuFromFile` ties this all together: when called it reads in a sudoku, applies the AC-3 algorithm and prints the result.

The code that generates an unsolved sudoku (see Appendix A) writes a string of digits to the file `sudoku/sudoku.txt`. These digits are in the same order as the squares are numbered with variables. This input is converted to a list of `Vals`, and `ac3domain` is called using the functions for generating domains and constraints for this particular sudoku. If the sudoku has a solution[1], the result is printed with `printSudoku`; if not (if the domain returned by `ac3domain` is empty), a message on the screen will indicate this fact.

```
ac3SudokuFromFile :: IO ()
ac3SudokuFromFile = do
  sudokuString <- readFile "sudoku/sudoku.txt"
  let values = map (Val . digitToInt) sudokuString
  do
    let sudoku = arcConsistentDomain (CSP (genSudokuDoms values) (genSudokuCons (map Var
        [0..80])))
    if not $ null sudoku
    then printSudoku sudoku
    else putStrLn "This sudoku has no solution."
```

[1] which a sudoku produced by the Appendix A code always will have

5 Objects

This chapter is based on a task described in Chapter 12 of [Win92]: *Symbolic Constraints and Propagation*. It describes how a computer can interpret a 2D image of lines as a 3D object. The task consists of deciding whether a given set of lines define an object without curved faces and of classifying the lines as either convex, concave or boundary edges. There is no shading involved, but an object can partially block the view of itself or other objects. The vertices at which the lines meet are restricted in such a way that they have a maximum of three adjacent faces. The viewpoint of the "observer" doesn't give any "illusions" like lines or vertices that overlap. This is not a heavy restriction, since in a real world scenario one could accomplish this by a small change of prespective.

```
module Objects where

import CSP
import AC3
import Data.List
import Data.Tuple

type LineID     = Variable
type JunctionID = Variable
-- We are going to give lines negative IDs and junctions non-negative IDs

data Junction   = L JunctionID JunctionID JunctionID
                | Fork JunctionID JunctionID JunctionID JunctionID
                | T JunctionID JunctionID JunctionID JunctionID
                | Arrow JunctionID JunctionID JunctionID JunctionID

type Object     = [Junction]
type Outline    = [(JunctionID,JunctionID)]
```

When a real object has its lines labeled correctly, there are only 18 possible junctions, as identified in Figure 12.14 of [Win92, p.259]. Since classifying a junction as either an L, Fork, T or Arrow junction can be done without any global information, we will assume that we know that property of each junction. This could also be done as a preprocessing step by another script, but that is not part of our task here.

For the labeling of a junction we use the orientation used in Figure 12.14 of [Win92, p.259], and first label the JunctionID in the rightmost direction. We then go anticlockwise labeling the next (two) junction(s). The final JunctionID is that of the junction itself. So e.g. Arrow (Var 8) (Var 4) (Var 5) (Var 1), would be an Arrow junction with JunctionID Var 1, and the junction in the direction of the "shaft" of the arrow has JunctionID Var 4. For a complete explanation the reader should read the beforementioned chapter.

To solve this task we add not one, but two variables for lines between two connected junctions. These lines are connected to eachother and one to each junction at the endpoints. When one of these lines is labeled as a "+" (convex edge) or "-" (concave), the other will be as well. For the boundary arrows we use a different label for each: when an arrow should go from junction 1 to junction 2, we label the line variable adjacent to junction 1 as "outgoing" and label the other line variable as "incoming".

We can use arc consistency to solve most of these problems: for many junctions knowledge of the labels of two of its adjacent lines will tell us which junction we are dealing with, and then we know the label of the other line. Propagating this process will add more and more labels,

until (hopefully) all lines are labeled.

```
-- We are going to work with 4-tuples of this type, these projections make that a bit
    easier
myfst, myfrth :: (JunctionID,LineID,LineID,JunctionID) -> JunctionID
mysnd, mytrd  :: (JunctionID,LineID,LineID,JunctionID) -> LineID
myfst  (a, _, _, _) = a
mysnd  (_, b, _, _) = b
mytrd  (_, _, c, _) = c
myfrth (_, _, _, d) = d

-- The reader might find some of the following functions a bit too resembling of imperative
     programming.
-- We would have to agree, but like to point out that most perfecting work has gone into
    the AC-3 algorithm.

-- Cycle through list of junctions, and add two lines between all connected junctions.
lineGeneration :: Object -> LineID -> [(JunctionID,LineID,LineID,JunctionID)]
lineGeneration [] _                              = []
lineGeneration ((L i j juncID):xs)       lineID = lineCreation juncID [i,j] lineID   ++
    lineGeneration xs (Var (getVar lineID-2*length (lineCreation juncID [i,j]   lineID)))
lineGeneration ((Fork i j k juncID):xs)  lineID = lineCreation juncID [i,j,k] lineID ++
    lineGeneration xs (Var (getVar lineID-2*length (lineCreation juncID [i,j,k] lineID)))
lineGeneration ((T i j k juncID):xs)     lineID = lineCreation juncID [i,j,k] lineID ++
    lineGeneration xs (Var (getVar lineID-2*length (lineCreation juncID [i,j,k] lineID)))
lineGeneration ((Arrow i j k juncID):xs) lineID = lineCreation juncID [i,j,k] lineID ++
    lineGeneration xs (Var (getVar lineID-2*length (lineCreation juncID [i,j,k] lineID)))

lineCreation :: JunctionID -> [JunctionID] -> LineID -> [(JunctionID,LineID,LineID,
    JunctionID)]
lineCreation _ [] _ = []
lineCreation juncID (i:is) lineID = if getVar juncID < getVar i then (juncID, lineID, Var (
    getVar lineID-1), i):lineCreation juncID is (Var (getVar lineID-2)) else lineCreation
    juncID is lineID
```

We have created two line variables between every pair of connected junctions. The constraint corresponding to an arc of a junction and adjacent line depends on the type of the junction and the direction of that line in it. Hence the pretty convoluted function.

```
-- Values:  Lines: 0 = "-", 1 = "+", 2 = "incoming <", 3 = "outgoing >"
--    Junctions: 0-5 = "L junction", 10-12 = "Fork", 20-23 = "T junction", 30-33 = "Arrow",
     ordered from high to low as in Figure 12.15.
-- Val 12 corresponds to the third, fourth and fifth Fork junction, since the rotational
    symmetry of the Fork junction makes them indistinguishable.
-- This gives the most lenient constraint. We hypothesize that this will result in some
    unsolved cases, which may be resolved by imposing constraints between the lines, or by
    path consistency.
junctionConstraints :: Object -> [(JunctionID,LineID,LineID,JunctionID)] -> ([Domain], [
    Constraint])
junctionConstraints [] _ = ([],[])
-- We consider the junctions one by one. We have already chosen the IDs of the lines
    between two connected junctions, so we use to the lineto lambda to find that correct ID
junctionConstraints ((L i j juncID):xs) lineInfo       = ((juncID,[Val 0, Val 1, Val 2, Val
     3, Val 4, Val 5]):doms,[
  ((juncID,lineto i), [(Val 0,Val 2), (Val 1,Val 3), (Val 2,Val 3), (Val 3,Val 1), (Val 4,
      Val 2), (Val 5,Val 0)]),
  ((juncID,lineto j), [(Val 0,Val 3), (Val 1,Val 2), (Val 2,Val 1), (Val 3,Val 2), (Val 4,
      Val 0), (Val 5,Val 5)])
  ] ++ cons) where
  (doms, cons) = junctionConstraints xs lineInfo
  lineto = \n -> head (map mysnd (filter (\line -> myfst line == juncID && myfrth line == n
      ) lineInfo) ++ map mytrd (filter (\line -> myfst line == n && myfrth line == juncID)
      lineInfo))
    -- Either (juncID, correctline, _ , i) or (i, _, correctline, juncID) appears in the
        output of lineGeneration, so head will always work.
junctionConstraints ((Fork i j k juncID):xs) lineInfo  = ((juncID,[Val 10, Val 11, Val 12])
    :doms,[
  ((juncID,lineto i), [(Val 10,Val 1), (Val 11,Val 0), (Val 12,Val 0), (Val 12,Val 2), (Val
      12,Val 3)]),
  ((juncID,lineto j), [(Val 10,Val 1), (Val 11,Val 0), (Val 12,Val 0), (Val 12,Val 2), (Val
      12,Val 3)]),
  ((juncID,lineto k), [(Val 10,Val 1), (Val 11,Val 0), (Val 12,Val 0), (Val 12,Val 2), (Val
```

```
      12,Val 3)])
  ] ++ cons) where
  (doms, cons) = junctionConstraints xs lineInfo
  lineto = \n -> head (map mysnd (filter (\line -> myfst line == juncID && myfrth line == n
      ) lineInfo) ++ map mytrd (filter (\line -> myfst line == n && myfrth line == juncID)
      lineInfo))
junctionConstraints ((T i j k juncID):xs) lineInfo      = ((juncID,[Val 20, Val 21, Val 22,
    Val 23]):doms,[
  ((juncID,lineto i), [(Val 20,Val 2), (Val 21,Val 2), (Val 22,Val 3), (Val 23,Val 2)]),
  ((juncID,lineto j), [(Val 20,Val 3), (Val 21,Val 3), (Val 22,Val 3), (Val 23,Val 3)]),
  ((juncID,lineto k), [(Val 20,Val 2), (Val 21,Val 3), (Val 22,Val 1), (Val 23,Val 0)])
  ] ++ cons) where
  (doms, cons) = junctionConstraints xs lineInfo
  lineto = \n -> head (map mysnd (filter (\line -> myfst line == juncID && myfrth line == n
      ) lineInfo) ++ map mytrd (filter (\line -> myfst line == n && myfrth line == juncID)
      lineInfo))
junctionConstraints ((Arrow i j k juncID):xs) lineInfo  = ((juncID,[Val 30, Val 31, Val
    32]):doms,[
  ((juncID,lineto i), [(Val 30,Val 3), (Val 31,Val 1), (Val 32,Val 0)]),
  ((juncID,lineto j), [(Val 30,Val 2), (Val 31,Val 1), (Val 32,Val 0)]),
  ((juncID,lineto k), [(Val 30,Val 1), (Val 31,Val 0), (Val 32,Val 1)])
  ] ++ cons) where
  (doms, cons) = junctionConstraints xs lineInfo
  lineto = \n -> head (map mysnd (filter (\line -> myfst line == juncID && myfrth line == n
      ) lineInfo) ++ map mytrd (filter (\line -> myfst line == n && myfrth line == juncID)
      lineInfo))

-- The Object type is nice and relatively readable, this function generates the correct (
    not so readable) CSP.
-- It preserves the JunctionIDs used in the Object def. Be aware! Junctions must have non-
    negative IDs to prevent becoming indistinguishable from lines.
cspGenerator :: Object -> Problem
cspGenerator object = CSP (allJuncDomains ++ allLineDomains) allConstraints where
  lineInfo                          = lineGeneration object (Var (-1)) -- Lines have
      negative variables, starting with Var -1
  (allJuncDomains, allJuncConstraints)  = junctionConstraints object lineInfo
  allLineDomains                    = concat [ [(Var (2*line+1),[Val 0, Val 1, Val 2,
      Val 3]),(Var (2*line),[Val 0, Val 1, Val 2, Val 3])] | line <- [-length lineInfo
      ..(-1)] ]
  allLineConstraints                = [ ((Var (2*line)+1, Var (2*line)), [(Val 0,Val 0)
      , (Val 1,Val 1), (Val 2,Val 3), (Val 3,Val 2)]) | line <- [-length lineInfo..(-1)] ]
  allConstraints                    = allJuncConstraints ++ allLineConstraints ++ map
      reflectConstraint allJuncConstraints ++ map reflectConstraint allLineConstraints
  reflectConstraint                 = \(arc, rel) -> (swap arc, map swap rel) -- we
      need arcs to be included in both directions.
```

An object will now have a corresponding CSP. Just like the Sudoku example is initialized by knowing some of the values, the line-drawing task is initialized by knowing the outline of an object. This outline can in most/all? cases be found by tracing the edge of a face from junction to junction, and finding an impossible junction if we picked an incorrect initial line/face. But there might even be several disconnected objects in our image, so for simplicity's sake we include the outline in the given test. If a real image/coordinates were given as our task, identifying the outline would be easy anyways.

```
-- Restrict the domains of the outline. A line with arrow leaving a vertex will get Val 3,
    an incoming line gets Val 2.
setOutlineArrows :: [Domain] -> [(JunctionID,LineID,LineID,JunctionID)] -> Outline -> [
    Domain]
setOutlineArrows doms _ []              = doms
setOutlineArrows doms lineInfo ((x,y):xs)
                  -- the following is safe if our Outline consists of connected junctions
                      on the object, since:
                  -- if x<y we will have (x,i,i-1,y) in our lineInfo, otherwise we will
                      have (y,i,i-1,x) as an element.
                  | getVar x < getVar y = let Just (_,i,_,_) = find (\line -> myfst line ==
                       x && myfrth line == y) lineInfo in
                        (i,[Val 3]):(Var (getVar i-1),[Val 2]):
                          delete (i,[Val 0, Val 1, Val 2, Val 3]) (delete (Var (getVar i-1)
                              ,[Val 0, Val 1, Val 2, Val 3]) (setOutlineArrows doms
```

```
                        lineInfo xs))
                | otherwise           = let Just (_,i,_,_) = find (\line -> myfst line ==
                    y && myfrth line == x) lineInfo in
                     (i,[Val 2]):(Var (getVar i-1),[Val 3]):
                       delete (i,[Val 0, Val 1, Val 2, Val 3]) (delete (Var (getVar i-1)
                           ,[Val 0, Val 1, Val 2, Val 3]) (setOutlineArrows doms
                           lineInfo xs))

-- Increase readability of output of ac3. We remove the (empty) queue and show the Int
    instead of Var Int.
ac3onObject :: (Object, Outline) -> (Bool,[(Int, [Value])],[((Int,Int),[(Value,Value)])])
ac3onObject (object, outline) = (bool, map (\(var,vals) -> (getVar var, vals)) doms, map
    (\((x,y),pairs) -> ((getVar x, getVar y),pairs)) cons) where
  CSP initdoms initcons = cspGenerator object
  (CSP doms cons, bool, _) = ac3 (CSP (setOutlineArrows initdoms (lineGeneration object
      (-1)) outline) initcons, True, initcons)
```

6 Tests

We now use the HSpec library to test our AC-3 algorithm.

```
module Main where

import CSP
import AC3
import Sudoku
import Objects

import Test.Hspec

main :: IO ()
main = hspec $ do
  describe "AC-3" $ do
    it "A test sudoku should be the same when run through AC-3 once as twice" $
      (arcConsistentDomain sudokuCSP) `shouldBe`
      (arcConsistentDomain (CSP (arcConsistentDomain sudokuCSP) sudokuCons)) where
        sudokuCons :: [Constraint]
        sudokuCons = genSudokuCons (map Var [0..80])
        sudokuCSP :: Problem
        sudokuCSP = CSP (genSudokuDoms sudoku) sudokuCons
    it "All lines and junctions of a cube should be labeled" $
      all (\(_,values) -> length values == 1) cubeDomains `shouldBe` True where
        (_, cubeDomains, _) = ac3onObject cube
    it "An impossible object should get an empty domain" $
      fst ac3onObject testfig18 `shouldBe` False
    it "Another object should get labels" $
      all (\(_,values) -> length values == 1) otherObjectDomains `shouldBe` True where
        (_, otherObjectDomains, _) = ac3onObject cube

sudoku :: [Value]
sudoku = [0,5,0,7,0,3,0,0,9,
          0,0,8,0,6,1,0,0,5,
          7,0,3,5,0,4,8,0,0,
          3,0,0,4,0,0,0,9,0,
          2,0,0,0,0,0,0,0,0,
          0,8,0,0,2,0,3,5,0,
          5,1,6,0,4,2,0,0,0,
          0,0,0,0,0,9,5,8,0,
          0,3,0,0,5,7,0,2,0]

cube :: (Object, Outline) -- As in Winston1992 Fig 12.15. Labeled clockwise starting at the
     upper left junction. Passes our object criteria so should return True.
cube = ([ Arrow (Var 1) (Var 5) (Var 6) (Var 0), L (Var 0) (Var 2) (Var 1), Arrow (Var 3) (
    Var 1) (Var 6) (Var 2), L (Var 2) (Var 4) (Var 3),
        Arrow (Var 5) (Var 3) (Var 6) 4, L (Var 4) (Var 0) (Var 5), Fork (Var 0) (Var 2) (
            Var 4) (Var 6)],
```

```
        [ (Var 0, Var 1), (Var 1, Var 2), (Var 2, Var 3), (Var 3, Var 4), (Var 4, Var 5), (
            Var 5, Var 0) ])

testfig18 :: (Object,Outline) -- As in Fig 12.18. Should return False, since as shown in
    Winston1992 Fig 12.18 there is no possible labeling of the middle junction.
testfig18 = ([ Arrow (Var 1) (Var 2) (Var 3) (Var 0), L (Var 0) (Var 4) (Var 1), L (Var 7)
    (Var 0) (Var 2), Fork (Var 0) (Var 8) (Var 9) (Var 3),
             Arrow (Var 5) (Var 1) (Var 6) (Var 4), L (Var 4) (Var 7) (Var 5), Fork (Var 8)
                 (Var 7) (Var 4) (Var 6),
             Arrow (Var 2) (Var 5) (Var 6) (Var 7), Arrow (Var 3) (Var 6) (Var 9) (Var 8),
                 L (Var 3) (Var 8) (Var 9)],
             [ (Var 0, Var 1), (Var 1, Var 4), (Var 4, Var 5), (Var 5, Var 7), (Var 7, Var
                 2), (Var 2, Var 0) ])

testfig12A :: (Object,Outline) -- Test including a T junction. This is from the perspective
     of observer A in Fig 12.12 in Winston1992.
testfig12A = ([ Arrow (Var 1) (Var 8) (Var 9) (Var 0), L (Var 0) (Var 2) (Var 1), Arrow (
    Var 4) (Var 1) (Var 3) (Var 2), L (Var 4) (Var 2) (Var 3),
              T (Var 3) (Var 5) (Var 2) (Var 4), Arrow (Var 6) (Var 4) (Var 9) (Var 5), L (
                  Var 5) (Var 7) (Var 6),
              Arrow (Var 8) (Var 6) (Var 9) (Var 7), L (Var 7) (Var 0) (Var 8), Fork (Var
                  0) (Var 7) (Var 5) (Var 9) ],
              [ (Var 0, Var 1), (Var 1, Var 2), (Var 2, Var 4), (Var 4, Var 5), (Var 5, Var
                   6), (Var 6, Var 7), (Var 7, Var 8), (Var 8, Var 0) ])
```

Moreover, the algorithm and sudoku implementation can be tested manually by running `python3 generate_sudoku.py` in the `sudoku` subdirectory to generate a new sudoku, and in the main directory running `stack ghci` and then `ac3SudokuFromFile`.

7 Conclusion

In conclusion, the AC-3 algorithm is a fast way to obtain an arc consistent CSP. Moreover, arc consistency is a nice way to simplify and reduce a CSP, and in some cases even solve it. Haskell is a suitable language to represent CSPs and implement the AC-3 algorithm.

7.1 Further improvements and research

Since the AC-3 algorithm is not able to solve all CSPs (for example, some sudokus it was unable to solve), this program could be augmented with other algorithms. For example, the PC-2 algorithm makes a CSP path-consistent. Path consistency is a stronger notion of consistency: it "tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables" [RN09, p. 210]. Taking also this notion of consistency into account and maybe even for example k-consistency, the program could come closer to solving CSPs.

A Appendix: Python script for sudoku generation

The following code is an adaptation of code to be found here; we deleted its GUI-related code, improved its formatting, and added some lines at the end so the output of the program was in a usable format for us and is written to a file.

The code generates a finished sudoku, removes some numbers from it and then ensures that the resulting unfinished sudoku has exactly 1 solution.

```
# Sudoku Generator Algorithm - www.101computing.net/sudoku-generator-algorithm/
from random import randint, shuffle
import os

# initialise empty 9 by 9 grid
grid = []
grid.append([0, 0, 0, 0, 0, 0, 0, 0, 0])
grid.append([0, 0, 0, 0, 0, 0, 0, 0, 0])
grid.append([0, 0, 0, 0, 0, 0, 0, 0, 0])
grid.append([0, 0, 0, 0, 0, 0, 0, 0, 0])
grid.append([0, 0, 0, 0, 0, 0, 0, 0, 0])
grid.append([0, 0, 0, 0, 0, 0, 0, 0, 0])
grid.append([0, 0, 0, 0, 0, 0, 0, 0, 0])
grid.append([0, 0, 0, 0, 0, 0, 0, 0, 0])
grid.append([0, 0, 0, 0, 0, 0, 0, 0, 0])

# A function to check if the grid is full
def checkGrid(grid):
  for row in range(0, 9):
    for col in range(0, 9):
      if grid[row][col] == 0:
        return False

  #We have a complete grid!
  return True

# A backtracking/recursive function to check all possible combinations of
# numbers until a solution is found
def solveGrid(grid):
  global counter
  # Find next empty cell
  for i in range(0, 81):
    row = i // 9
    col = i % 9
    if grid[row][col] == 0:
      for value in range (1, 10):
        # Check that this value has not already be used on this row
        if not(value in grid[row]):
          # Check that this value has not already be used on this column
          if not value in \
            (grid[0][col], grid[1][col], grid[2][col], \
             grid[3][col], grid[4][col], grid[5][col], \
             grid[6][col], grid[7][col], grid[8][col]):
            # Identify which of the 9 squares we are working on
            square = []
            if row < 3:
              if col < 3:
                square = [grid[i][0:3] for i in range(0, 3)]
              elif col < 6:
                square = [grid[i][3:6] for i in range(0, 3)]
              else:
                square = [grid[i][6:9] for i in range(0, 3)]
            elif row < 6:
              if col < 3:
                square = [grid[i][0:3] for i in range(3, 6)]
              elif col < 6:
                square = [grid[i][3:6] for i in range(3, 6)]
              else:
                square = [grid[i][6:9] for i in range(3, 6)]
            else:
```

```
            if col < 3:
              square = [grid[i][0:3] for i in range(6, 9)]
            elif col < 6:
              square = [grid[i][3:6] for i in range(6, 9)]
            else:
              square = [grid[i][6:9] for i in range(6, 9)]
          # Check that this value has not already be used on this 3x3 square
          if not value in (square[0] + square[1] + square[2]):
            grid[row][col] = value
            if checkGrid(grid):
              counter += 1
              break
            else:
              if solveGrid(grid):
                return True
      break
  grid[row][col] = 0

numberList = [1, 2, 3, 4, 5, 6, 7, 8, 9]
#shuffle(numberList)

# A backtracking/recursive function to check all possible combinations of
# numbers until a solution is found
def fillGrid(grid):
  global counter
  # Find next empty cell
  for i in range(0, 81):
    row = i // 9
    col = i % 9
    if grid[row][col] == 0:
      shuffle(numberList)
      for value in numberList:
        # Check that this value has not already be used on this row
        if not(value in grid[row]):
          # Check that this value has not already be used on this column
          if not value in \
            (grid[0][col], grid[1][col], grid[2][col], \
             grid[3][col], grid[4][col], grid[5][col], \
             grid[6][col], grid[7][col], grid[8][col]):
            # Identify which of the 9 squares we are working on
            square = []
            if row < 3:
              if col < 3:
                square = [grid[i][0:3] for i in range(0, 3)]
              elif col < 6:
                square = [grid[i][3:6] for i in range(0, 3)]
              else:
                square = [grid[i][6:9] for i in range(0, 3)]
            elif row < 6:
              if col < 3:
                square = [grid[i][0:3] for i in range(3, 6)]
              elif col < 6:
                square = [grid[i][3:6] for i in range(3, 6)]
              else:
                square = [grid[i][6:9] for i in range(3, 6)]
            else:
              if col < 3:
                square = [grid[i][0:3] for i in range(6, 9)]
              elif col < 6:
                square = [grid[i][3:6] for i in range(6, 9)]
              else:
                square = [grid[i][6:9] for i in range(6, 9)]
            # Check that this value has not already be used on this 3x3 square
            if not value in (square[0] + square[1] + square[2]):
              grid[row][col] = value
              if checkGrid(grid):
                return True
              else:
                if fillGrid(grid):
                  return True
      break
  grid[row][col] = 0
```

```
# Generate a Fully Solved Grid
fillGrid(grid)

# Start Removing Numbers one by one
# A higher number of attempts will end up removing more numbers from the grid,
# potentially resulting in more difficult grids to solve!
attempts = 5
counter = 1
while attempts > 0:
  # Select a random cell that is not already empty
  row = randint(0, 8)
  col = randint(0, 8)
  while grid[row][col] == 0:
    row = randint(0, 8)
    col = randint(0, 8)
  # Remember its cell value in case we need to put it back
  backup = grid[row][col]
  grid[row][col] = 0

  # Take a full copy of the grid
  copyGrid = []
  for r in range(0, 9):
     copyGrid.append([])
     for c in range(0, 9):
        copyGrid[r].append(grid[r][c])

  # Count the number of solutions that this grid has (using a backtracking approach
      implemented in the solveGrid() function)
  counter = 0
  solveGrid(copyGrid)
  # If the number of solution is different from 1 then we need to cancel the change by
      putting the value we took away back in the grid
  if counter != 1:
    grid[row][col] = backup
    # We could stop here, but we can also have another attempt with a different cell just
        to try to remove more numbers
    attempts -= 1

# ADAPTATION BY US:
# flatten the grid into one long list of digits (without trailing newline!)
# and print to file "sudoku.txt"
flat_grid = [item for sublist in grid for item in sublist]
flat_grid = ''.join(map(str,flat_grid))
with open("sudoku.txt", 'w') as file:
    file.write(flat_grid)
print("Sudoku printed to sudoku.txt")
```

References

[RN09] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[Win92] P.H. Winston. *Artificial Intelligence*. A-W Series in Computerscience. Addison-Wesley Publishing Company, 1992.