SATisfiability

1 Australia

Consider this map of Australian States. It is coloured, with four colours, but two adjacent states (states that share a common border) have the same colour. We want a better colouring.

The four-colour theorem says that any planar map can be coloured with four colours, so that no two adjacent states have the same colour. Here we have not coloured the sea and if we use four colours to colour the land, we would need a fifth colour to colour the sea, as they all border the sea. However, adding a node (representing the sea) that has an edge to all existing nodes, the graph remains planar. This means we can still colour the map (with the sea) using four colours.

Furthermore, since each Australian state meets the sea, there must be a 3-colouring of the states. Of course, it is easy to find one, but we want to use this problem to make a connection with logic. We first throw away some irrelevant detail by converting this to a problem about graphs. A (directed) graph is just a binary relation E on a set N of nodes; we call this an adjacency relation. If $E(a, b)$ we say there is an edge $\langle a, b\rangle$ from a to b. We can draw a diagram to visualise the adjacency relation.[1]

For example, a triangle has three nodes and three edges, a tetrahedron has four nodes and six edges, and a cube has eight nodes and twelve edges.

1. Draw these three graphs. Can you draw them so that no edges cross each other? (If you can, they are *planar* graphs.) Can you find an example of a non-planar graph?

[1] For graph colouring the direction of the links is irrelevant. It suffices to have one link for each adjacent pair.

2. In this question you will develop a Haskell program to test whether a colouring of the map of Australia is a legal 3-colouring.

   (a) To present our example as a graph, let the nodes be the seven states, and let X and Y be adjacent iff they share a common border. Draw the graph.
   Use the names of the state capitals as names for the seven nodes,
   `Melbourne,Sydney,Hobart,Darwin,Perth,Adelaide,Brisbane`.

   (b) How many edges are there in your Australia graph?

We can now replace the map colouring problem by the graph colouring problem. Can we colour each node so that no pair of adjacent nodes have the same colour — or equivalently, so that any two adjacent nodes have different colours? Note that nodes **can** be coloured with multiple colours.

3. Suppose we have three colours, $C = \{r, a, g\}$ (red, amber, green). Without thinking about logic (for a brief moment) can you count the number of three-colourings of your graph?
   Hint: If you arbitrarily fix the colouring of two adjacent nodes, can you extend your colouring to the entire graph?

To express the problem in logic, we need to specify two properties of a colouring:

A each city must have a colour
$\forall x \in \texttt{cities}.\ \exists a \in \texttt{colours}.\ \texttt{colouring}\ x\ a$

B adjacent cities cannot have the same colour
$\forall x, y \in \texttt{cities} \mid x\texttt{`adj`}y.\ \forall a \in \texttt{colours}.\ \neg(\ \texttt{colouring}\ x\ a\ \wedge\ \texttt{colouring}\ y\ a)$

we can rephrase this using deMorgan
$\forall x, y \in \texttt{cities} \mid x\texttt{`adj`}y.\ \forall a \in \texttt{colours}.\ (\neg\ \texttt{colouring}\ x\ a\ \vee\ \neg\texttt{colouring}\ y\ a)$

Consider these two requirements, A and B. Do they capture your informal understanding of the colouring problem? Notice that we do not say that we may only apply a single colour to each city.
Should we say this? What difference would it make to the possible solutions? For the purposes of this tutorial we will **not** impose this additional constraint.

4. How many distinct colourings are there of this graph?

Now we will represent the problem in Haskell.

```
data City   = Perth | Brisbane | Hobart | Adelaide | Sydney | Darwin | Melbourne
            deriving (Ord, Eq, Show)
data Colour = Red | Green | Amber       deriving (Ord,  Eq, Show)
cities  = [ Perth, Brisbane, Hobart, Adelaide, Sydney, Darwin, Melbourne ]
colours = [Red, Green, Amber]

adj :: City -> City -> Bool
adj x y = (x,y) `elem`
          [ (Perth,Darwin), (Perth,Adelaide), (Darwin,Adelaide), (Darwin,Brisbane)
          , (Melbourne,Sydney), (Adelaide,Sydney), (Adelaide,Melbourne)
          , (Brisbane,Sydney), (Brisbane,Adelaide)]

type Colouring = City -> Colour -> Bool

eachCityHasAColour :: Colouring -> Bool
eachCityHasAColour colouring =
  and [ or[colouring city paint | paint <- colours ] | city <- cities ]
```

Here we have set up the types for this problem, and written a Haskell implementation of the first requirement, A, so we can test whether a given `paint :: Colouring`, does assign a Colour to each City. We could have written this condition as:

```
every cities (\city -> some colours (\colour ->  paint city colour))
```

However, for our present purposes it is helpful to use the `and[ or[ ... | ... ] | ... ]` formulation.

5. Write a Haskell implementation of the second requirement, B. Again we want to match the `and or` pattern, so you just need to replace the dots in the code below:

```
adjacentCitiesNotSameColour :: Colouring -> Bool
adjacentCitiesNotSameColour paint =
  and[ or [ ... ] | a <- cities, b <- cities, adj a b, c <- colours ]
```

Next, you will use `quickCheck` to find a colouring that satisfies these two constraints. A `Colouring` is defined by the values `[ colouring x y | x <- cities, y <- colours ]`.
The code provided (lines 31 onwards) uses the two constraints you will have defined, to build a `quickCheck` property taking 21 Boolean arguments, that is satisfied if there is no 3-colouring of your graph. If this check fails the counterexample represents a 3-colouring.

We have 21 `(City, Colour)` pairs, we use 21 booleans to define a colouring.

```
colourBy a b c d e f g h i j k l m n o p q r s t u =
  let pairs = [(city, colour) | city <- cities, colour <- colours]
      choices :: [((City,Colour),Bool)]
      choices = zip pairs -- pair each pair with one of the 21 booleans
                [ a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u ]
      the [x] = x
  in
   (\city colour -> the [ b | ((x, y), b) <- choices, x == city, y == colour ])
```

To get quickCheck to search for a colouring that satisfies the conditions, we hypothesise that one of our conditions always fails.

```
test_prop a b c d e f g h i j k l m n o p q r s t u =
  let paint = colourBy a b c d e f g h i j k l m n o p q r s t u in
   not (adjacentCitiesNotSameColour paint)
   ||
   not (eachCityHasAColour paint)
```

If you run quickCheck with a large enough number of tests

e.g. `quickCheck (withMaxSuccess 12345 test_prop)`

(but 12345 may not be big enough) you should find a counter-example to our test proposition.

If a counter-example is found, it is printed as 21 Boolean values. You can see what these mean by matching these Booleans against the list `pairs` defined in the code.

6. What proportion of the valuations of 21 boolean variables correspond to legal colourings?

Random search and test is not a good way to solve such problems.

2 A simple language

Our first language, the language of clausal forms, is simple and limited. It formalises the `and` `or` pattern we have used earlier. The language has *atoms* – propositional letters, *literals* – negative or positive atomic assertions, *clauses* – disjunctions of literals, and *forms* – conjunctions of clauses.

We introduce these as Haskell types.

```
data    Literal a = N a | P a       deriving (Ord, Eq, Show)
newtype Clause a  = Or  [Literal a] deriving (Ord, Eq, Show)
newtype Form a    = And [Clause a]  deriving (Eq, Show)
```

We will need a few simple functions, `neg` to negate literals, `<&&>` to conjoin two forms, and `canonical` to remove any duplications, of literals within a clause, and of clauses within a form.

```
-- negation of a Literal
neg :: Literal a -> Literal a
neg (P a) = N a
neg (N a) = P a

-- conjunction of two Forms
(<&&>) :: Form a -> Form a  -> Form a
And xs <&&> And ys = And( xs ++ ys )

canonical :: Ord a => [Clause a] -> [Clause a] -- removes duplicates
canonical cs = uniqueSort $ map (\ (Or xs) -> Or(uniqueSort xs)) cs
```

As a first example, we will express our first property, A, as a `Form` in this language.

The first step is to introduce a suitable type of atoms. Our basic propositions are that we will paint a given city with a given colour.

```
data Paint = Paint City Colour
```

We can then translate the Haskell code for `eachCityHasAColour` into a formal statement in our language of clausal forms. Here is the code from the earlier tutorial.

```
eachCityHasAColour :: Colouring -> Bool
eachCityHasAColour paint =
  and [ or[ paint city colour | colour <- colours ] | city <- cities ]
```

Here is the corresponding formal statement.

```
formalEachCityHasColour :: Form Paint
formalEachCityHasColour =
  And [ Or [ P (Paint city colour) | colour <- colours ] | city <- cities ]
```

The atoms have type `Paint`, for example, `Paint Sydney Red :: Paint`. We replace `and` with `And`, and `or` with `Or` , the boolean expression, `paint city colour` is replaced by `P (Paint city colour)`, a positive assertion of an atom. For an occurrence of `not (paint city colour)` (a negative occurrence) we would have used `N (Paint city colour)`.

You can examine the value:

```
> :t formalEachCityHasColour
formalEachCityHasColour :: Form Paint
> formalEachCityHasColour
And [Or [P (Paint Perth Red),P (Paint Perth Green),P (Paint Perth Amber)]
    ,Or [P (Paint Brisbane Red),P (Paint Brisbane Green),P (Paint Brisbane Amber)]
    ,Or [P (Paint Hobart Red),P (Paint Hobart Green),P (Paint Hobart Amber)]
    ,Or [P (Paint Adelaide Red),P (Paint Adelaide Green),P (Paint Adelaide Amber)]
    ,Or [P (Paint Sydney Red),P (Paint Sydney Green),P (Paint Sydney Amber)]
    ,Or [P (Paint Darwin Red),P (Paint Darwin Green),P (Paint Darwin Amber)]
    ,Or [P (Paint Melbourne Red),P (Paint Melbourne Green),P (Paint Melbourne Amber)]]
```

7. Translate your code for `adjacentCitiesNotSameColour` to a formal statement

```
formalAdjacentCitiesNotSameColour :: Form Paint
formalAdjacentCitiesNotSameColour = undefined
```

This problem can be solved by a simple search with backtracking. We code the reduction `(<<)` of a list of clauses by a literal, and the `simple` (Davis - Putnam) algorithm introduced in class.

```
(<<) :: Eq a => [Clause a] -> Literal a -> [Clause a]
cs << x = [ Or (delete (neg x) ys) | Or ys <- cs, not $ x `elem` ys ]

simple :: Ord a => Form a -> [Val a]
simple (And cs) =
  let search cs = case cs of
        []              -> [[]] -- no clauses; trivial solution
        Or []  : _      -> []   -- empty clause; no models
        Or (x : _) : _ ->
          [     x : m | m <- search $ cs << x ]     -- reduce by x
          ++
          [ neg x : m | m <- search $ cs << neg x] -- reduce by neg x
  in
   search (canonical cs)
```

You should run this algorithm

```
colourings = simple (formalEachCityHasColour <&&> formalAdjacentCitiesNotSameColour)
```

8. How many valuations do you find?

9. How does this answer relate to your answer to question 3?
   (This is a bit tricky, and will be explained later if you can't figure it out.)

3 Latin Squares

A Latin Square is an $n * n$ matrix with integer entries from the digits `[1..n]` such that each row and each column includes each digit, and no row or column has any digit repeated – and no entry can have two digits.

10. Complete the following formal specification of an $n \times n$ latin square, by replacing the `undefined` sections in the following template.

```
latin :: Int -> Form (Int,Int,Int)
latin n =
  let rows    = [0..n-1]
      columns = [0..n-1]
      digits  = [1..n]
      everyRowHasEveryDigit =
        And[ Or[ P (r,c,d) | c <- columns ]
             | r <- rows, d <- digits]
      everyColumnHasEveryDigit = undefined
      noSquareHasTwoDigits =
        And[ Or[ N (r, c, d), N (r, c, d') ]
           | r <- rows, c <- columns,
             d <- digits, d' <- digits, d < d']
      noRowHasARepeatedDigit = undefined
      noColumnHasARepeatedDigit = undefined
  in everyRowHasEveryDigit
     <&&>everyColumnHasEveryDigit
     <&&>noSquareHasTwoDigits
     <&&>noRowHasARepeatedDigit
     <&&>noColumnHasARepeatedDigit
```

Here we are using triples of type `(Int, Int, Int)` as atoms. The positive literal `P (i, j, k)` is interpreted to mean that the matrix has entry `k` in row `i` and column `j`.

The following type and function are provided in the code for this tutorial, to make it easier to understand your output.

```
data Square   = Square [[Int]] deriving (Show)
toSquare :: Int -> Val (Int,Int,Int) -> Square
```

11. Try your code out as follows:

```
(toSquare 3. head) $ simple (latin 3)
```

For $n = 3$ we have $2^{27} = 128Mi$ valuations in our search space.

You can also try the same with 4 in place of 3, but this problem is already pushing the limits of our simple search.

12. How many valuations are there for the Latin Square problem when $n = 4$?

    Press control-c when your patience runs out.

We need a better algorithm! Here is the skeleton of the dpll algorithm discussed in lectures (for the time being, the unit clause case is commented out):

```
dpll :: Ord a => Form a -> [Val a]
dpll (And cs) =
  let models cs = case prioritise cs of
        []             -> [[]]       -- no clauses; trivial solution
        Or []  : _     -> []         -- empty clause; no models
    --  Or [x] : _     -> undefined -- unit clause
        Or (x : _) : _  ->
          [     x : m | m <- models $ cs << x   ]
          ++
          [ neg x : m | m <- models $ cs << neg x]
  in
   models (canonical cs)
  where prioritise = id
```

13. First replace `id` in `prioritise = id` in the last line so that,

    ```
    prioritise :: [Clause a] -> [Clause a]
    ```

    sorts clauses, of the form `Or lits`, by the length of the list `lits`. You can use the library function `sortOn` to do this.

    Use the resulting function `dpll` to see how for you can get with solving larger latin squares. If you want more detailed timing, you can time your computations in ghci by giving the command `:set +s` before running your code.

14. Then uncomment the unit-clause line and replace the undefined with the correct recursive call.

    Again check how this affects the performance on latin square problems.

4 Sudoku

In tutorial 5 you completed the following code.

```
check :: (Int -> Int -> Int -> Bool) -> Bool
check entered =
  -- every square is filled
  and [ or[entered i j k | k <- [1..9] ]
      | i <- [1..9], j <- [1..9] ]
  && -- no square is filled twice
  and [ or[ not(entered i j k), not (entered i j k') ]
      | i <- [1..9], j <- [1..9], k <- [1..9], k' <- [1..(k-1)]]
  && -- every row contains every digit
  and [ or [entered i j k | j <- [1..9] ]
      | i <- [1..9], k<- [1..9] ]
  -- every column contains every digit
  -- every big square contains every digit
```

You will now convert this to a formal specification of the rules of sudoku. Just as we For this example, we will use triples `(i, j, k) :: (Int, Int, Int)` as atoms. The translation of the first three rules are given below. You should translate the conditions you wrote last week.

```
sudoku :: Form (Int, Int, Int)
sudoku =
  -- every square is filled
  And [ Or[P (i, j, k) | k <- [1..9] ]
      | i <- [1..9], j <- [1..9] ]
  <&&> -- no square is filled twice
  And [ Or[ N (i j k), N (i, j, k') ]
      | i <- [1..9], j <- [1..9], k <- [1..9], k' <- [1..(k-1)]]
  <&&> -- every row contains every digit
  And [ Or [P (i, j, k) | j <- [1..9] ]
      | i <- [1..9], k<- [1..9] ]
  -- every column contains every digit
  -- every big square contains every digit
  --+ no row has repeated digit
  --+ no column has repeated digit
  --+ no bigsquare has repeated digit
```

15. Take the code for sudoku that you developed in tutorial 5 and translate it to formal form.

To make the problem tractable (for our dpll SAT solver), you need to do a bit more. It is fairly easy for us to see that one consequence of the sudoku rules is that no digit occurs twice in any row (and we can say the same about any column or any major square). Since there are 9 squares in a row, if we repeated some digit then some other digit could not occur. There are 9 squares and 9 digits, so each one must occur exactly once. However, if we consider only the five rules, there are many partial attempts at a solution that don't immediately contradict any of these rules, but include repetitions and are doomed to fail.

16. Add code for (at least two of) the extra rules. The Haskell file for this tutorial includes ten example problems. You can try your code on the first of these by running

    ```
    > dpll (entries (head problems) <&&> sudoku)
    [[P (1,8,8),P (1,9,2),P (2,1,6),P (2,4,4),P (4,1,4), ......
    ```

    Running in ghci, you may have to wait twenty seconds, or so, for an answer ...

    The code provided includes functions

```haskell
-- conver 81-digit textual problem to logical form
entries :: String -> Form (Int, Int, Int)
-- convert valuation to textual 81-digit form of solution
showEntries :: [Literal (Int,Int,Int)] -> String
```

and a further function

```haskell
-- convert problem or solution in 81-digit form
-- to a String suitable for putStrLn sudoku display
pretty :: String -> String
```

If we add the three constraints indicated with `--+` we allow our program to avoid these blind alleys. If you're having difficulty coding the last one, you will find that the first two are enough to make the problem tractable, but adding all three is better.

For sudoku, it is straightforward for us to see that these added conditions follow from the rules, so we can add them. However, for more general problems there might be such rules, of which we're unaware. One of the major improvements of mordern SAT solvers over dpll is to build solvers that can learn such rules, and add them automatically to speed up search.

The remaining question is optional.

17. You can try to improve on the dpll code. One opportunity, with some low-hanging fruit, is to look carefully at how we prioritise clauses, and replace the sorting with something simpler to achieve the same end. A skeleton, optimistically named `speedy` is provided in the file for your experiments.

    At this stage you need to compile your code to evaluate its performance.