

# COMP4075/G54RFP Coursework Part III

Benjamin Charlton — psybc3 — 4262648

16<sup>th</sup> January 2019

## 1 Project Overview

### 1.1 Motivation

The original basis for this project comes from a series of lab exercises from the G52AIM module, Artificial Intelligence Methods. The lab exercises involved implementing a variety of AI methods to solve some basic optimisation problems namely MAX-SAT problems.

Many of the methods implemented involved combining smaller functions together to create the desired effect. This could effectively translated into a functional programming setting. I thought it would be interesting to try and reimplement these some of these methods in Haskell to see the benefits of the FP paradigm to these AI methods.

### 1.2 Aims of the Project

The original coursework took place over several lab sessions, incorporating a variety of topics and theoretical questions, as well as originally relying heavily of a java based framework. Due to this only part of the lab exercises will be looked into and some parts of the java framework will have to be remade in Haskell.

Here is what I intend to create in this project:

- MAX-SAT problem generator
- Naive solvers
- Genetic Algorithm Solver
- MAX-SAT evaluator
- Hill Climbing solvers

While creating this components, real world functional programming techniques will be applied to make this work efficient and maintainable.

### 1.3 Technical Background

#### 1.3.1 MAX-SAT

MAX-SAT is an optimisation problem which is NP-Hard. Given a logic formula in conjunctive normal form, the aim is to maximise the number of clauses which are true after variable assignment. A logical problem in conjunctive normal form has the following formal grammar.

$CNF \rightarrow (C) \mid (C) \ \& \ CNF$        $L \rightarrow V \mid !V$   
 $C \rightarrow L \mid L + C$        $V \rightarrow 1 \mid 2 \mid 3 \mid \dots$

**CNF Formula** A CNF formula is made up of clauses which are logically and'd together. If each clause in a CNF problem is true then the whole formula will return true. Although in MAX-SAT we aren't aiming for all clauses to be true only to maximise the number that are true.

**Clauses C** Each clause contains a series of literals, which are logically or'd together. This means so long as any literal can evaluate to true the whole clause will be true.

**Literals L** A literal is a variable which is either positive or negative, this is achieved by applying a logical not or not doing so.

**Variables V** Finally each literal is assigned a variable which are later used to evaluate a solution. There can be any number of variables so integers are used here in this example to represent the different variables.

**Solution for CNF** A solution for CNF will involve all of the variables in the problem being given a boolean value. From here you can evaluate each clause and then find how many clauses evaluate to true. One benefit of how this problem works is that any solution (granting that all variables required have an assignment) will be a valid solution to the problem.

### 1.3.2 Mathematical Optimisation

The MAX-SAT problem is a form of optimisation problem that can be solved by a optimisation technique. An optimisation problem means that there is an objective function that you wish to maximise the value of by changing the input. In the case of MAX-SAT the objective function is the number of clauses that evaluate to true given our solution as input. The goal of any optimisation problem is to find the solution that gives the best possible value for the objective function.

In MAX-SAT there may be several optimal solutions and it is hard to calculate if you do have an optimal solution. Apart from the trivial case of all clauses being true there is no guarantee that an optimal solution has been found, there is also no guarantee that all clauses can be true either.

While you could devise a way to generate all possible solutions and evaluate them all to find the optimal, due to the NP-hardness of this problem this method will take none polynomial time to run which is very inefficient. The field of mathematical optimisation aims to find techniques to more efficiently find optimal or near optimal solutions to these types of problems. While the techniques used for this project don't guarantee optimal solutions the achieve near optimal solutions that have been shown to produce solutions very close to the true optimum. Although MAX-SAT doesn't have any immediately obvious applications it is very similar to problems such as scheduling where near optimal solutions may be sufficient so the application of these algorithms have real world importance.

## 2 Implementaion

In this section the implementation of each component will be discussed including any key decisions and interesting aspects.

### 2.1 Code Structure

The code for this project is split in to 3 main modules CNF, Solvers, and Tests. The idea with this is it separates the code up based upon function and allows the modules to be shared easily depending on what part a user wants. This was also a massive benefit when programming as it was clear what functionality each submodule provided making debugging easier. Splitting everything up also allowed for a high level of code reuse throughout the project.

These features of the Haskell programming language, while not functional and seen in other none functional languages, allow for Haskell to be used like other popular languages used in industry.

### 2.2 MAX-SAT problem generator

In the java framework there was the ability to generate a MAX-SAT problem in a suitable form, before any solvers could exist this component and decisions of how the problem will appear also had to be made.

#### 2.2.1 Creating new types

To help define the problem more succinctly new types and data were created in the type system, by the code below. These types are very similar to the formation of the grammar presented in section 1.3.1 and thus was very simple to implement in Haskell from the grammar. All of the types also derived show so that it could be used to print out the problems to console.

```
newtype Problem = P [Clause]
newtype Clause  = C [Literal]
data    Literal = Positive Var | Negative Var
newtype Var     = V Int
```

#### 2.2.2 Problem Generation

While in theory a CNF problem can have clauses of differing sizes, it was chosen to make all the clauses contain the same number of literals. This restriction means that the generator creates a MAX-kSAT where each clause has exactly k literals. It should be noted that this restriction only applies to the generation of the problem and isn't used by the solvers meaning that the solvers will work on any MAX-SAT problem not just MAX-kSAT.

This allows any MAX-kSAT to be defined by 3 values, v the number of variables, c the number of clauses, and r the number of variables per clause. From there it was simple to create a series of functions that would generate a problem, which can be found in the CNF.Generator module.

The implementation is broken into the generation of a problem, clause, literal and variable. For an individual variable a random number is picked between 0 and the number of variables in the problem, it is then used in the literal generation by randomly choosing if it will be negated. For the problem and clause generation a list of the

desired size is created by using `replicateM`. As randomness is used this means that the problem generated has to be wrapped up in the IO monad.

## 2.3 MAX-SAT evaluator

The final submodule in the CNF module is the evaluator. This required defining the type of a solution which was simply a mapping of variable numbers to a boolean value.

```
newtype Solution = S [(Int , Bool)] deriving (Show, Eq)
```

An evaluator would therefore have the following type.

```
evaluateProblem :: Problem -> Solution -> Int
```

This is quite trivial to then implement with list comprehensions and pattern matching. A full implementation can be found in the `CNF.Evaluator` module.

## 2.4 Naive solvers

3 naive solvers are grouped together in the `Solvers.Naive` module. These were created to allow for a quick way to check if the generator/evaluator was working correctly but also served as a comparison point for the other algorithms. One thing all of these solvers have in common is that they don't take the problem into account when creating the solution just the number of variables needed for it to be valid.

### 2.4.1 All True/False

The most basic of ways to create a valid solution is to make all the variables true or false, which is what these 2 functions did. While these solutions are valid they often don't produce good results.

```
allTrueSolution , allFalseSolution :: Int -> Solution
allTrueSolution v = S [(i, True) | i <- [1..v]]
allFalseSolution v = S [(i, False) | i <- [1..v]]
```

### 2.4.2 Random Solution

The other naive way to create a valid solution is to randomly assign boolean values to variables. This produces an answer which is valid but again not very good, but it does provide a good starting point for the other methods to iterate from.

```
randomSolution :: Int -> IO Solution
randomSolution v = do bs <- replicateM v (randomRIO (True, False))
                    return $ S $ zip [1..v] bs
```

## 2.5 Hill Climbing solvers

## 2.6 Genetic Algorithm Solver

# 3 Learnt stuff — Needs another name

A section reflecting upon what was learned from the project and your thoughts around the project topic from a real-world programming perspective.