Assignment 3

Date of Submission - 10th April, Friday, 11:55pm

1 Introduction

You would have realized by now that you can think of computation in several ways. In this assignment we shall consider a completely different way of solving a computational problem. As an example, consider the 4-queens problem. To solve this problem, we think of 16 boolean variables $\{x_{11}, x_{12}, x_{13}, x_{14}, x_{21}, x_{22}, \ldots, x_{41}, x_{42}, x_{43}, x_{44}\}$, each variable corresponding to a square in the 4×4 chessboard. A boolean variable has two values $true$ (or `#t` in Racket representation) and $false$ (or `#f`). In addition, the values of variables have a certain interpretation that is relevant to the given problem. For our example, if x_{ij} is $true$, it means that there is a queen at x_{ij}. Needless to say, the x_{ij}'s cannot have arbitrary value. Their values must be chosen so that the constraints of the 4-$queens$ problem are satisfied. This is done by constructing a boolean formula, such that the only values of the x_{ij}'s that satisfy the formula are the ones that satisy the constraints of the 4-$queens$ problem. Here is how:

1. First satisfy the constraint that the first row has exactly one queen. This can be conveniently expressed as the conjunction of two constraints:
   (a) The first row has at least one queen. This is expressed by the formula: $\{x_{11} \vee x_{12} \vee x_{13} \vee x_{14}\}$, and
   (b) The first row has at most one queen. This is expressed as $\{\neg x_{11} \vee \neg x_{12}\} \wedge \{\neg x_{11} \vee \neg x_{13}\} \wedge \{\neg x_{11} \vee \neg x_{14}\} \wedge \{\neg x_{12} \vee \neg x_{13}\} \wedge \{\neg x_{12} \vee \neg x_{14}\} \wedge \{\neg x_{13} \vee \neg x_{14}\}$
2. Similarly express the constraint that the other three rows have exactly one queen.
3. Next express the fact that each column has at most one queen.
4. Finally express that each diagonal has at most one queen

We can that the conjunction of all these formulas to get one grand formula, say F. Notice carefully that the set of values of the variables x_{ij} that makes the formula F true are the only solutions of the 4-queens problem.

So how does one find a set of values of the x_{ij}s that makes the formula true? Programs which, given a formula, find an assignment that makes the formula true are called SAT-solvers. This assignment is about making an elementary SAT-solver.

2 Representation of boolean formulas

First, note that our formulas will be in conjunctive normal form, i.e. they will be a conjunction of disjunctions. Here is an example:

$(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3)$

Each of the components of the conjunction is called a *clause*. For example $(\neg x_1 \vee \neg x_2)$, $(x_1 \vee \neg x_2)$ and $(\neg x_1 \vee \neg x_3)$ are clauses. Each of the components of a clause is called a *literal*. $\neg x_1$, $\neg x_2$, x_1, and $\neg x_3$ are literals in the formula. A literal of the form x is a *positive literal* and a literal of the form $\neg x$ is a *negative literal*.

Second, the user will input the formula in the following form:

```
(list '(-1 -2) '(1 -2) '(-1 -3))
```

I hope you see the connection. Finally, formulas will be represented in Racket using the `struct`

```
(struct Var (lit) #:transparent)
(struct And (x y) #:transparent)
(struct Or (x y) #:transparent)
(struct Not (e) #:transparent)
(struct Const (bool) #:transparent)
```

So that the above formula woud be represented as:

```
(And
 (Or (Not (Var 1)) (Not (Var 2)))
 (And
  (Or (Var 1) (Not (Var 2)))
  (Or (Not (Var 1)) (Not (Var 3)))))
```

We shall provide the function to convert the input to a formula using the above struct.

3 The Sat-Solving Algorithm

Here is a summary of the SAT solving method called the Davis-Putnam-Logemann-Loveland (DPLL) procedure. Given a formula F, call the procedure being described below as $dpll(F)$. Here is a summary of the procedure:

1. **Unit propagation:** If the formula contains a unit clause (a clause consisting of a single literal), drop all clauses containing this literal and drop the negation of the literal from the remaining clauses. Applying this step on $(\neg x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge x_3$ because of x_3 gives $\neg x_1 \wedge (x_1 \vee \neg x_2)$.

2. **Literal Elimination:** Identify a literal that has the same polarity in every clause that it appears in. Eliminate all such clauses. Applying this on $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3)$ because of x_2 gives $(\neg x_1 \vee \neg x_3)$.

3. At this point if the formula has no clauses then return true. Else if it has an empty clause, then return false.

4. The previous steps can be considered to be the base steps of the DPLL procedure.

   (a) Now choose any literal, l, set its value to true. Remove all clauses containing l and remove the negation of the literal from all clauses. Call the resulting formula F'. If the result of $dpll(F')$ is true, return true.

   (b) Set the value of the same literal l to false. Remove all clauses containing the negation of the literal l and remove l itself from all clauses containing it. Call the resulting formula F'. Return $dpll(F')$.

4 A model implementation

Like last time, I am going to provide you with a model implementation that is easy to use. Find a tar-zipped file called `model-implementation-6.7.tgz`. Its contents are

```
model-implementation --: test-model-implementation.rkt
                        : compiled--: satSolver_rkt.zo
                                     : satSolver_rkt.dep
```

Run drracket with `test-model-implementation.rkt`. This file has a list of test cases that you could try. The model implementation works for drracket 6.7.

Separately, I shall provide a file called `utilities.rkt` containing the structs required for representing boolean formulas and the function `parseExp`.