Programming Project #4
3-Satisfiability in Common Lisp

CS 355

Due 11:59 pm, Thursday March 24, 2016

1 Boolean Satisfiability

The *3-SAT* problem is a classic *decision problem* in computer science. You are given a set of boolean variables and a boolean expression written in *conjunctive normal form* (CNF) with 3 variables per clause. Is there some assignment of *true* and *false* to each variable so that the entire expression is true?

For example, say our variables are $\{A, B, C, D\}$ and our expression is

$$(D \vee A \vee \neg C) \wedge (\neg C \vee D \vee \neg A) \wedge (\neg C \vee D \vee \neg A) \wedge (B \vee \neg C \vee \neg D). \tag{1}$$

The following assignment of boolean variables

$$\{A = \text{true},\ B = \text{true},\ C = \text{false},\ D = \text{true}\} \tag{2}$$

"satisfies" the given expression. Note that above expression consists of a *conjunction* of 4 clauses. Each clause is a *disjunction* of thee terms where each term is a variable or a negation of a variable.

3-SAT is known to be *NP-complete* which means we can efficiently check to see if a solution is correct, but there is no known tractable way to find a solution in general. Therefore, we typically use some sort of heuristic search method to find a solution (if there is one). Interestingly enough, 2-SAT (2 variables per clause) is solvable in polynomial time. You can read more about boolean satisfiability here:

`http://en.wikipedia.org/wiki/Boolean_satisfiability_problem`

2 Lisp functions for solving 3-SAT

You are to write several top-level functions in *Common Lisp* as described below to help solve for instances of 3-SAT.

1. We consider a particular assignment of truth values to our given variables a *state* in a search graph. We will represent a state in Lisp as a list of pairs. Each pair is a list containing the variable name (a symbol in Lisp) and its corresponding truth value `t` (true) or `nil` (false). Write a function `eval-var` that returns the value associated with a particular variable:

```
(defun eval-var (var state)
...
)
```

For example

```
* (setf *state* '((a nil) (b t) (c t) (d nil)))
(A NIL) (B T) (C T) (D NIL))
* (eval-var 'b *state*)
T
* (eval-var 'd *state*)
NIL
```

2. We will represent a single *clause* in Lisp as a list of 3 elements. Each element is either a single variable name (i.e., an *atom*) or a list containing the symbol `not` followed by a variable name. Write the function `eval-clause` that evaluates a clause (i.e., returns `t` or `nil`) for a given variables state:

```
(defun eval-clause (clause state)
...
)
```

For example

```
* *state*
((A NIL) (B T) (C T) (D NIL))
* (setf *clause* '(a (not b) c))
(A (NOT B) C)
* (eval-clause *clause* *state*)
T
```

3. Write a function `get-vars` that returns a list of all the variables in the clause.

```
(defun get-vars (clause)
...
)
```

You do not need to worry about duplicates in this case. For example

```
* (get-vars '(A (NOT B) C))
(A B C)
```

4. Now write a function that returns all the variables contained in a list of variables.

```
(defun get-all-vars (clauses)
...
)
```

You should not have duplicate entries in the list, for example:

```
* (setf *clauses* '((a (not b) c) (a (not b) (not c)) (a (not b) d)))
((A (NOT B) C) (A (NOT B) (NOT C)) (A (NOT B) D))
* (get-all-vars *clauses*)
(C D B A)
```

The built-in `union` function is handy here:

```
* (union '(a b h) '(a b c d e f))
(H A B C D E F)
```

5. Write a function `unsat-clauses` that returns all the unsatisfied clauses in an expressions for a given state:

   ```
   (defun unsat-clauses (clauses state)
   ...
   )
   ```

   For example

   ```
   * *state*
   ((A NIL) (B T) (C T) (D NIL))
   * *clauses*
   ((A (NOT B) C) (A (NOT B) (NOT C)) (A (NOT B) D))
   * (unsat-clauses *clauses* *state*)
   ((A (NOT B) (NOT C)) (A (NOT B) D))
   ```

6. Write a function `flip-var` that "flips" the "truthfulness" a particular variable in a state list.

   ```
   (defun flip-var (var state)
   ...
   )
   ```

   For example

   ```
   * *state*
   ((A NIL) (B T) (C T) (D NIL))
   * (flip-var 'b *state*)
   ((A NIL) (B NIL) (C T) (D NIL))
   * (flip-var 'c *state*)
   ((A NIL) (B T) (C NIL) (D NIL))
   ```

7. We consider S' to be a *neighbor* to state S if S' can be created by flipping the state of exactly one variable in S. We consider S' to be a *better* neighbor if it generates less unsatisfied clauses than S in a given expression. Write the function `get-better-neighbor` that finds *some* neighbor to a given state that yields fewer unsatisfied clauses; This function actually returns a list where the first item is the better state and the rest of the list contains the corresponding unsatisfied clauses.

```
(defun get-better-neighbor (clauses state vars num-unsat)
...
)
```

Argument description:

`clauses` : boolean expression (list of clauses),

`state` : current state,

`vars` : list of variables used to generate neighbors *(hint: this list controls the recursion)*,

`num-unsat` : number of unsatisfied clauses generated by `state`.

If there are no better neighbors (i.e., we have reached the top of a "hill" or a "plateau") return `nil`.

```
*clauses*
((A (NOT B) C) (A (NOT B) (NOT C)) (A (NOT B) D))
* *state*
((A NIL) (B T) (C T) (D NIL))
* (get-better-neighbor *clauses* *state* '(a b c)
   (length (unsat-clauses *clauses* *state*)) )
(((A T) (B T) (C T) (D NIL)))
```

The above is actually a poor example since the returned neighbor is actually a solution and thus there are no corresponding unsatisfied clauses.

8. Now we put all the pieces together to perform *simple hill climbing* in search for a solution. Write the function `simple-hill-climb` that begins at a given start state and continually "climbs" by looking for better neighbors until it finds the solution or has visited a prescribed number of states.

   ```
   (defun simple-hill-climb (clauses state dist unsat)
   ...
   )
   ```

   Argument description:

   `clauses` : list of clauses we are trying to satisfy,

   `state` : starting state,

   `dist` : number of states left to examine before giving up,

   `unsat` : list of unsatisfied clauses generated by `state`. If this is `nil`, then `state` is a solution.

3 What to submit

You will fork an initial version of the project at

`https://gitlab.encs.vancouver.wsu.edu/CS355/3sat`

This will provide an initial skeleton of the project (make sure it is private) and a set of unit tests used by GitLab's Continuous Integration (CI) tools. These tests will be triggered whenever you push your solution back to GitLab and you can see the result of the build there. There is also an initial `README.txt` provided that you should modify.

Each function you write should have no *side effects* (i.e., each function returns freshly calculated values without altering any of the arguments or any global variables). There is no need to perform any iteration – recursion is all that is needed. Test each function individually as you write them. In your source code include comments that give your name, email address, and a brief overview of the problem. You source code should contain the eight top-level functions described in this document.

Push you final solution and make sure both Kyle and I have *REPORTER* access to your repository. Good luck.