CMPUT 366—Intelligent Systems

Assignment 1

Fall 2004
Department of Computing Science
University of Alberta

Due: *Thursday, September 30 at 23:59:59 local time*
Worth: 10% of final grade
(5 questions worth 2% each, plus a 2% bonus question.)
Instructor: Dale Schuurmans, Ath409, x2-4806, dale@cs.ualberta.ca

---

Note: This assignment is to be submitted electronically by using the Unix "astep" command which is available on the undergraduate machines. You need to submit five Java classes in files called: Randsat.java, Satcom.java, Satcomfast.java, Satinc.java, Satincfast.java—and, if you choose, the answer to the bonus question in an additional file called Satfast.java. You can submit your files by typing "`astep -c c366 -p a1` *file1 file2 ...*". If you need help with "astep" see: http://ugweb.cs.ualberta.ca/~astep

---

Definition: Propositions in *conjunctive normal form* are of the form:

$$\begin{array}{rcl} \mathit{CNF} & \leftarrow & \mathit{clause}_1 \wedge \mathit{clause}_2 \wedge \cdots \wedge \mathit{clause}_m \\ \mathit{clause} & \leftarrow & \mathit{literal}_1 \vee \mathit{literal}_2 \vee \cdots \vee \mathit{literal}_k \\ \mathit{literal} & \leftarrow & \mathtt{p}_i \\ \mathit{literal} & \leftarrow & \neg\mathtt{p}_i \quad \text{where } \mathtt{p}_i \text{ is a primitive proposition} \end{array}$$

Propositions in *strict conjunctive normal form* do not have any repeated primitive propositions in clauses, nor any opposing pair if literals in any clause (that is, both `p` and ¬ `p` never occur in the same clause).

In this assignment you will represent:

Primitive propositions by positive integers, 1, 2, 3, etc.

For example, a primitive proposition $\mathtt{p}_1$ will be denoted by 1, $\mathtt{p}_2$ by 2, etc.

Literals by nonzero integers, 1, -1, 2, -2, etc.

For example, a negated literal $\neg\mathtt{p}_2$ will be denoted by -2, etc.

Clauses by an array of nonzero integers (literals).

For example, a clause $(\mathtt{p}_1 \vee \neg\mathtt{p}_2)$ will be denoted by [1, -2], etc.

CNF formulas by an array of clauses.

For example, a CNF formula $(\mathtt{p}_1 \vee \neg\mathtt{p}_2) \wedge (\mathtt{p}_2 \vee \mathtt{p}_3)$ will be denoted by [[1, -2], [2, 3]].

Note: that a clause that contains an opposing pair of literals, i and -i, immediately evaluates to *true* and can be removed from the CNF formula.

Truth values by 1 and -1, where 1 denotes the value *true* and -1 denotes *false*.

Truth value assignments by an array of 1's, -1's, and 0's. The zero denotes that no truth value is assigned.

For example, a truth value assignment to a set of primitive propositions would be an array [-1, 1, ... , -1]. The assignment to primitive proposition p_i is located in array position i.

Consider a complete truth value assignment to all primitive propositions.

- The assignment *satisfies* a positive literal i, if i is assigned value 1 in the assignment.
- The assignment *satisfies* a negative literal -i, if i is assigned value -1 in the assignment.
- The assignment *satisfies* a clause if it satisfies *at least one* literal in the clause.

  (Thus, a length zero clause is unsatisfiable.)
- The assignment *satisfies* a CNF formula if is satisfies *every* clause in the formula.

  (Thus, a length zero CNF formula is unfalsifiable.)

Part 1 (Generating constraint satisfaction problems—2%)

Write a Java class `Randsat` (in the file `Randsat.java`), which contains a public static function "`randsat`" with the signature

```
public static int[][] randsat(int n, int k, int m)
```

Given three positive integers n, k and m, the function generates a random CNF formula $\texttt{C} = [c_1, c_2, ..., c_m]$ in strict conjunctive normal form, where each clause c_j contains exactly k random literals from {1, -1, 2, -2, ..., n, -n}. In particular, for each clause c_i choose k random primitives from {1, 2, ..., n}, and for each primitive, i, randomly decide whether to negate it or not. Use a different seed for the random number generator in each run.

Note that one can think of the primitive propositions i as *variables* that need truth values assigned and of the clauses c_j as *constraints* that need to be satisfied simultaneously. You will use this function to generate random constraint satisfaction problems.

Interestingly for $k = 3$ it is known that if $m < 3n$ then there are usually not enough constraints to make the problem unsatisfiable (that is, most of the random `C`'s will be satisfiable); whereas if $m > 6n$ then there are usually too many constraints to make the problem satisfiable (that is, most of the random `C`'s will be unsatisfiable). A critical "phase transition" occurs near $m = 4.26n$. At this point half of the random problems are satisfiable. Curiously, random problems generated at the phase transition appear to be the hardest to solve (on average) for any solution technique.

Part 2 (Simple complete search—2%)

Write a Java class `Satcom` (in the file `Satcom.java`), which contains a public static function "`satcom`" with the signature

```
public static int[] satcom(int[][] C)
```

The function takes a CNF formula `C` and returns an array of truth value assignments that satisfies `C`. The array should have size $(n + 1)$ where p_n is the largest-numbered primitive proposition occurring in `C`. (Recall that Java indexes arrays beginning with 0 and we want to index variables by their name.) Any array positions corresponding to primitives *not occurring* in `C` should be initialized to 0. If there are no assignments that satisfy `C` then `satcom` should return `NULL`.

So, for example:
`Satcom.satcom( [[-1, 2], [-2, 4]] )`
should return

`[0, 1, 1, 0, 1]` or `[0, -1, 1, 0, 1]`
or `[0, -1, -1, 0, 1]` or `[0, -1, -1, 0, -1]`
or `[0, 0, 1, 0, 1]` (p_1 remains unassigned)
or `[0, -1, 0, 0, 1]` (p_2 remains unassigned)

or `[0, -1, -1, 0, 0]` (p_4 remains unassigned).

That is, if there are many satisfying assignments, your function should just return one.

Similarly:
`Satcom.satcom( [[1], [-1]] )` should return `NULL`.

You must implement a simple backtrack search procedure as outlined below.

Pseudo code for a simple backtrack search

function `satcom` (`C`)

`assign` = [0, 0, ..., 0]

if `C` has length zero, **return** `assign`

else if `C` contains a zero length clause, **return** `NULL`

else

pick any primitive proposition i that occurs in `C`
`assign`[i] := -1
r := `satcom`(assign i false in `C`)
if r = `NULL`
`assign`[i] := 1
r := `satcom`(assign i true in `C`)

endif

if r = `NULL`
return `NULL`
else
return `assign`

function "assign i false in `C`"
Temporarily remove all clauses from `C` that contain $-i$
Temporarily remove i from remaining clauses

function "assign i true in `C`"
Temporarily remove all clauses from `C` that contain i
Temporarily remove $-i$ from remaining clauses

Note: all clauses and literals removed from `C` have to be restored to their previous state after each backtrack.

Part 3 (Faster complete search—2%)

Write a Java class `Satcomfast` (in the file `Satcomfast.java`), which contains a public static function "`satcomfast`" with the signature

```
public static int[] satcomfast(int[][] C)
```

As before, the function takes a CNF formula `C` and returns a truth value assignment that satisfies `C`. If there are no such assignments then `satcomfast` should return `NULL`.

You must implement the improved backtrack search procedure outlined below.

Pseudo code for an improved backtrack search

function `satcomfast` (`C`)

Follow the same outline as the simple backtrack search, except add the following two conditions between the first "**else if**" and "**else**":

else if `C` contains a proposition i that occurs only negatively
  or occurs negatively in some clause of length one
  `assign`[i] := -1
  r := `satcomfast`(assign i false in `C`)

else if `C` contains a proposition i that occurs only positively
  or occurs positively in some clause of length one
  `assign`[i] := 1
  r := `satcomfast`(assign i true in `C`)

Part 4 (Simple incomplete search—2%)

Write a Java class `Satinc` (in the file `Satinc.java`), which contains a public static function "`satinc`" with the signature

```
public static int[] satinc(int[][] C, int maxRestarts, int maxSteps)
```

The function takes a CNF formula `C` and searches for a truth value assignment that satisfies `C`. When the function finds such an assignment it is returned as the result. If, however, the function cannot find a satisfying assignment, it simply keeps searching until a time limit is reached and returns `NULL`. In particular, implement the following procedure:

```
function satinc (C, maxRestarts, maxSteps)
  for r := 1 to maxRestarts
    a := random initial assignment to primitive propositions occurring in C
        (assign 0 to array positions not corresponding to a primitive proposition in C)
    if a satisfies every clause in C
        return a
    for s := 1 to maxSteps
        Given a, consider flipping the assignment of each primitive proposition individually.
        Determine which flip results in the fewest unsatisfied clauses in C.
        a := assignment obtained by best flip   (**)
        if a satisfies every clause in C
            return a

  return NULL
```

(**) Break ties randomly. Also, if the best flip results in more unsatisfied clauses than the current assignment, still make the flip. (This, surprisingly, improves performance.)

Part 5 (Faster incomplete search—2%)

Write a Java class `Satincfast` (in the file `Satincfast.java`), which contains a public static function "`satincfast`" with the signature

```
public static int[] satincfast(int[][] C, int maxRestarts, int maxSteps,
                               double walkProb)
```

The function takes a compound proposition `C` and searches for a truth value assignment that satisfies `C`. When the function finds such an assignment it is returned as the result. If, however, the function cannot find a satisfying assignment, it simply keeps searching until a time limit and returns `NULL`. In particular, implement the following procedure:

```
function satincfast (C, maxRestarts, maxSteps, walkProb)
  for r := 1 to maxRestarts
      a := random initial assignment to primitive propositions occurring in C
          (assign 0 to array positions not corresponding to a primitive proposition in C)
      if a satisfies every clause in C
          return a
      for s := 1 to maxSteps
          if flipping the assignment of some proposition falsifies no new clauses   (*)
            a := assignment obtained by making such a flip
          else
            Pick a random unsatisfied clause c in C.
            Consider propositions i occurring in c:
                With probability walkProb, flip the assignment of a random i in c.
                Otherwise, flip the assignment of the best i in c. (**)
          a := assignment that results
          if a satisfies every clause in C
             return a
  return NULL
```

(*) Break ties randomly, but only consider propositions that occur in at least one currently false clause.

(**) The best flip is the one which results in the smallest number of *new* clauses becoming falsified.

Bonus! (Faster search—2%)

Implement either a faster complete or incomplete search (your choice). Submit your improved procedure in a Java class `Satfast` which contains a public static function "`satfast`". To obtain a full bonus your procedure must be significantly faster than a reasonably implemented `satcomfast` or `satincfast`. Clearly document the improved algorithm you implement, along with its Java signature and parameters.