CS 486/686—Introduction to Artificial Intelligence

Assignment 1

Fall 2001
Department of Computer Science
University of Waterloo

Due: *Monday, Oct 1 at 12:00pm* (midnight)
Worth: 10% of final grade
(3 questions worth 2%, 4%, and 4%, plus a 2% bonus question.)
Instructors: Dale Schuurmans, DC1310, x3005, dale@cs.uwaterloo.ca
Vlado Keselj, DC2133, x3378, vkeselj@cs.uwaterloo.ca

Note: This assignment is to be submitted electronically by using the Unix "submit" command which is available on the undergraduate machines (contact me if you do not have an undergraduate account). You need to submit three Java classes in files called: Randsat.java, Satcom.java, and Satinc.java, and, if you choose, the answer to the bonus question in two additional files called Graph2sat.java and Sat2gc.java. You can submit all files at once by putting them in a directory, changing to that directory, and typing "`submit cs486 a1 .`". See "`man submit`" for further details.

Similarly to Lisp or Scheme syntax, we use the expression (`and` $expr_1$ $expr_2$ ... $expr_n$) to represent conjunction of the expressions $expr_1$, $expr_2$, ..., and $expr_n$. For implication we use the function `imp`, and for disjunction `or`. The true value is represented by the string `true` and the false value is represented by `false`.

Definition: Propositions in *strict implicative normal form* are of the form:

$$\begin{array}{rcl} sinf & \leftarrow & (\texttt{and}\ impl_1\ impl_2\ \ldots\ impl_k) \\ impl & \leftarrow & (\texttt{imp}\ (\texttt{and}\ \texttt{p}_1\ \texttt{p}_2\ \cdots\ \texttt{p}_k)\ (\texttt{or}\ \texttt{p}_{k+1}\ \texttt{p}_{k+2}\ \cdots\ \texttt{p}_n)\ ) \\ & & \text{where the } \texttt{p}_i \text{ are atomic propositions that appear without duplication.} \end{array}$$

Note that the empty conjunction, `(and)`, and empty disjunction, `(or)`, are both considered legal expressions, and evaluate to `true` and `false` respectively. This means that a contradiction can be represented by `(imp(and)(or))`. This also means that if an *impl* contains the same proposition on both sides, it immediately evaluates to `true` and can be removed from the formula—hence the restriction to no duplication within an *impl*.

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 following signature:

`public static String randsat(int n, int k, int m)`

Given three positive integers n, k and m, the function generates a random compound proposition `c` = `"(and` r_1 r_2 ... r_m`)"` in strict implicative normal form where each implication r_j contains exactly k random primitives from {`p1`, `p2`, ..., `pn`}. In particular, for each implication r_i choose k random primitives, and for each primitive $\mathtt{p}_i$ randomly decide whether to put the it on the "`and`" or the "`or`" side of the implication. Use a generally different seed for the random generator in each run.

Note that one can think of the primitive propositions $\mathtt{p}_i$ as *variables* that need truth values assigned and of the implications r_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).

Part 2 (Complete, systematic search—4%)

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

`public static String satcom(String c)`

The function takes a compound proposition `c` (in strict implicative normal form) and returns a list of truth assignments `"p1=true p2=false ... pn=true"` that makes `c` evaluate to `true`. If there are no such assignments then `satcom` should return `"fail"`.

So, for example:

`Satcom.satcom("(and (imp(and p1)(or p2)) (imp(and p2)(or p3)))")`

should return

`"p1=true p2=true p3=true"`
or `"p1=false p2=true p3=true"`
or `"p1=false p2=false p3=true"`
or `"p1=false p2=false p3=false"`
or some permutation of these, like `"p2=true p3=true p1=false"`.

That is, if there are many satisfying assignments, your function should just return one.
Note that you may also find a *partial* assignment that satisfies a given proposition. For example, the partial assignment `"p1=false p2=false"` also forces the above proposition to evaluate to `true` regardless of the value assigned to `p3`. It is acceptable to return such an assignment as long as this still forces the given proposition to evaluate to `true`.

Similarly:

`Satcom.satcom("(and (imp(and p1)(or)) (imp(and)(or p1)))")` should return `"fail"`.

To receive full marks you must implement the efficient systematic search procedure outlined below. However, if you run out of time you can just implement an inefficient exhaustive search to receive half marks.

Pseudo code for an efficient systematic search

```
function satcom (c)

  if c = (and), return "" (the empty assignment)

  else if c contains the contradiction (imp(and)(or)), return "fail"

  else if c contains a proposition p that occurs only in "and" parts or
          as the only proposition in some impl (and in the "and" part)
          v := false
          r := satcom(assign p false in c)

  else if c contains a proposition p that occurs only in "or" parts or
          as the only proposition in some impl (and in the "or" part)
          v := true
          r := satcom(assign p true in c)
  else
          pick any proposition p in c
          v := false
          r := satcom(assign p false in c)
          if r = "fail"
            v := true
            r := satcom(assign p true in c)
  endif

  if r = "fail"
          return "fail"
  else
          return r+ "p=v"


function "assign p false in c"
          Remove all impls from c that contain p in "and" part
          Remove p from remaining "or" parts


function "assign p true in c"
          Remove all impls from c that contain p in "or" part
          Remove p from remaining "and" parts
```

Part 3 (Incomplete, unsystematic search—4%)

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

`public static String satinc(String c, int maxRestarts, int maxSteps)`

The function takes a compound proposition `c` (in strict implicative normal form) and searches for a list of truth assignments `"p1=true p2=false` ... `pn=true"` that makes `c` evaluate to `true`. 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 has been reached and returns `"fail"`. The numbers `maxRestarts` and `maxSteps` are two positive parameters. In particular, implement the following procedure:

function `satinc` (`c`, `maxRestarts`, `maxSteps`)

- **for** $r := 1$ to `maxRestarts`
  - $a :=$ random initial assignment
  - **if** a satisfies every implication 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 implications in `c`.
    - $a :=$ assignment obtained by best flip (**)
    - **if** a satisfies every implication in `c`
      - **return** a
- **return** `"fail"`

(**) 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.)

Bonus! (Constraint satisfaction—2%)

A useful application of satisfiability search procedures is that they can be used to solve hard constraint satisfaction problems. A good example of this is the *map coloring problem*: given a planar graph (where vertices correspond to countries and edges correspond to borders) assign a color to the vertices so that no two adjacent vertices have the same color. A famous theorem asserts that a planar graph can always be colored with at most 4 colors (proved first in 1976 by Appel and Haken with the help of a computer). Since then many polynomial time algorithms have been developed for 4-coloring planar graphs (e.g. Robinson et al. 1996 propose a quadratic time algorithm). However, the problem of deciding whether a planar graph is actually 3-colorable is NP-complete. In this question you will use your satisfiability search procedures to find 3-colorings of graphs (if one exists). For this question graphs will be represented by a list of vertices `"v1 v2 ... v3"` and a list of undirected edges `"(v1 v3) (v2 v8) ... "`.

(a) Write a Java class `Graph2sat` (in the file `Graph2sat.java`), which contains a public static function "`graph2sat`" with the following signature:
`public static String graph2sat(String vlist, String elist)`
The function takes a vertex list and an edge list as arguments and returns a compound proposition `(and` r_1 r_2 ... r_m `)` that represents the graph 3-coloring problem as a propositional satisfiability problem.
The variables in the satisfiability problem are defined as follows. For every vertex `vi` in the graph, create three propositional letters `viR`, `viB`, `viY` to stand for the three assertions that the vertex `vi` is red, the vertex `vi` is blue, and the vertex `vi` is yellow, respectively. (Thus, asserting `viR` = `true` means that vertex `vi` is colored red.) The rules $r_1, ..., r_m$ in the final compound proposition represent the constraints on the assignment of colors to vertices. The first class of constraints is that each vertex can have at least and at most one color. This is expressed by the constraints ($\texttt{viR} \vee \texttt{viB} \vee \texttt{viY}$) and $\neg(\texttt{viR} \wedge \texttt{viB})$, $\neg(\texttt{viR} \wedge \texttt{viY})$, and $\neg(\texttt{viB} \wedge \texttt{viY})$ for each vertex `vi`. The next class of constraints is that adjacent vertices cannot have the same color. This is represented by the constraints $\neg(\texttt{viR} \wedge \texttt{vjR})$, $\neg(\texttt{viB} \wedge \texttt{vjB})$, and $\neg(\texttt{viY} \wedge \texttt{vjY})$ for any pair of vertices `vi` and `vj` that are connected by an edge.
To write "graph2sat" simply convert each one of the above constraints into a rule r_k in implicative normal form and compose the final conjunction `(and` r_1 r_2 ... r_m `)` by listing all the rules.

(b) If a satisfying assignment can be found to the propositional satisfiability problem, then you still have to convert the solution back to a coloring of the original graph. Write a Java class `Sat2gc` (in file `Sat2gc.java`) that includes the function:
`public static String sat2gc(String a)`
The function takes a truth assignment
`"v1R=true v1B=false v1Y=false v2R=false` $\cdots$ `vnY=true"`
as input and returns an assignment of colors to the vertices
`"v1=R v2=B` $\cdots$ `vn=Y"`.
Your job will be made easy by realizing that a satisfying assignment must have exactly one of the variables `viR`, `viB`, or `viY` set to true.