CSC242: Intro to AI
Project 2: Model Checking and Satisfiability Testing

In this project we will investigate using propositional logic to represent knowledge and do inference.

To simplify things, we will assume that knowledge is represented as *clauses* (a.k.a. conjunctive normal form or CNF). You should know that this is not a limitation, since any sentence of propositional logic can be converted into an equivalent set of clauses (although this may result in an exponential increase in the number of sentences).

Requirements

1. Design and implement a representation of clauses.

2. Implement the TT-ENTAILS model-checking algorithm and demonstrate it on several examples.

3. Implement the GSAT and/or WalkSAT local search satisfiability testing algorithms and demonstrate it/them on several examples.

More details are in each of the following sections.

Part 1: Representing Clauses

Representing knowledge using propositional logic begins with identifying the *atomic propositions*: basic, irreducible features of the world that are either true or false. These are also called "propositional variables" because they can take on (or be assigned) the values true and false.

Without loss of generality, we may assume that our atomic propositions (propositional variables) use the symbols x_1, x_2, and so on. If you have clauses that use some other symbols, you can rename them.

A clause is a disjunction ("or") of literals, where a literal is either an atomic proposition or the negation of an atomic proposition. For example, the following sentence in CNF:

$$(x_1 \lor x_3 \lor \neg x_4) \land (x_4) \land (x_2 \lor \neg x_3)$$

is equivalent to the following set of three clauses:

$$\{x_1 \lor x_3 \lor \neg x_4, \ x_4, \ x_2 \lor \neg x_3\}$$

I suggest that you represent clauses as sets of integers, where the number i means that x_i is in the clause, and the number $-i$ means that $\neg x_i$ is in the clause. With this encoding, the previous clauses would be represented as follows:

$$\{1, 3, -4\}, \{4\}, \{2, -3\}$$

This is easy to code up and easy to work with.

DIMACS CNF File Format

The DIMACS CNF file format is a simple text format for describing sets of clauses. Appendix A describes the format. Your program must be able to read a DIMACS CNF file and create the corresponding set of clauses. Don't worry—it's not hard if you represent your clauses as described above.

Several examples of DIMACS CNF files are provided with the project description, and you can find many, many more on the internet. Just don't look for code if you go looking for problems to test **YOUR** code on.

Part 2: Model Checking

Model checking is a way of testing whether a sentence or set of sentences α *logically entails* another sentence or set of sentences β. Since all our sentences are clauses, we will do model checking with clauses in this project.

A *possible world* is an assignment of true or false to each of the atomic propositions (propositional variables).[1]

I suggest that you represent assignments as arrays, lists, or vectors of boolean values. The element at index i in the array is the value assigned to atomic proposition x_i. Note that if you need to represent *partial* assignments, then you have to be able to distinguish the values true, false, and "not assigned." In Java, this means using `Boolean` rather than `boolean`.

It is then easy to lookup the value assigned to an atomic proposition using its index i (or $-i$) as stored in clauses. It is also easy to write a function that tests whether an assignment satisfies a clause (makes it true).

With this you can implement the TT-ENTAILS algorithm for propositional entailment from AIMA Fig. 7.10.

Your project must show your model checker running on at least the following problems:

1. Show that $\{P, P \Rightarrow Q\} \models Q$.

2. The Wumpus World example from AIMA 7.2 formalized in 7.4.3, solved in 7.4.4.
   Background knowledge:

   R1: $\neg P_{1,1}$
   R2: $B_{1,1} \Leftrightarrow P_{1,2} \vee P_{2,1}$
   R3: $B_{2,1} \Leftrightarrow P_{1,1} \vee P_{2,2} \vee P_{3,1}$
   R7: $B_{1,2} \Leftrightarrow P_{1,1} \vee P_{2,2} \vee P_{1,3}$

   The agent starts at $[1, 1]$ (Fig. 7.3(a)). Add perception:

   R4: $\neg B_{1,1}$

   Show that this knowledge base entails $\neg P_{1,2}$ and $\neg P_{2,1}$, but not $P_{2,2}$ or $\neg P_{2,2}$. The agent doesn't know enough to conclude anything about $P_{2,2}$.

---

[1]Possible worlds are also sometimes called "models," but that gets confused with the models of a sentence, so I will save the word "model" for the latter.

The agent moves to $[2, 1]$ (Fig 7.3(b)). Add perception:

R5: $B_{2,1}$

Show that this knowledge base entails $P_{2,2} \vee P_{3,1}$, but not $P_{2,2}$, $\neg P_{2,2}$, $P_{3,1}$, or $\neg P_{3,1}$. The agent knows more, but not enough.

The agent moves to $[1, 2]$ (Fig 7.4(a)). Add perception:

R6: $\neg B_{1,2}$

Show that this knowledge base entails $\neg P_{2,2}$ and $P_{3,1}$.

3. (Russell & Norvig) If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.

   (a) Can we prove that the unicorn is mythical?
   (b) Can we prove that the unicorn is magical?
   (c) Can we prove that the unicorn is horned?

For each problem, you must:

1. Express the knowledge and queries using propositional logic (this is already done for the first two problems).

2. Convert the sentences to conjunctive normal form (clauses). You can do this by hand easily enough for these small problems.

3. Have a method or function that creates the knowledge base of clauses for the problem and then calls your implementation of TT-ENTAILS to test each query.

4. Print informative messages so that we know what is going on.

These are all tiny problems. Your model checker should run quickly and perfectly. You are welcome to try larger, harder knowledge bases and queries if you want.

procedure GSAT
Input: a set of clauses α, MAX-FLIPS, and MAX-TRIES
Output: a satisfying truth assignment of α, if found
begin
    for i := 1 to MAX-TRIES
        T := a randomly generated truth assignment
        for j := 1 to MAX-FLIPS
            if T satisfies α then return T
            p := a propositional variable such that a change
                in its truth assignment gives the largest
                increase in the total number of clauses
                of α that are satisfied by T
            T := T with the truth assignment of p reversed
        end for
    end for
    return "no satisfying assignment found"
end

Figure 1: The procedure GSAT

Part 3: Satisfiability Testing

A set of clauses is *satisfiable* is there is some (at least one) assignment of true and false to the atomic propositions (propositional variables) that makes all of the clauses true. A set of clauses is *unsatisfiable* if there is no assignment that makes them all true.

The local search approach to testing satisfiability was described in (Selman, Levesque, and Mitchell, 1992) and is shown in Figure 1. States are truth assignments. Typically we use a complete state formulation, so states are complete truth assignments (a value for every atomic proposition). Actions are then picking an atomic proposition and flipping its assigned value from true to false or vice-versa.

The heuristic value of a state (assignment) is the number of clauses that it satisfies. An assignment that satisifes all n clauses (that is, a solution) has value n, which is nice. GSAT does hillclimbing for at most MAX-FLIPS steps per run, with random restarts up to MAX-TRIES times.

Two comments from the original GSAT paper (you can read it yourself):

> One distinguishing feature of GSAT, however, is the presence of sideways moves. [. . .] In a departure from standard local search algorithms, GSAT continues flipping variables even when this does not increase the total number of satisfied clauses.

> Another feature of GSAT is that the variable whose assignment is to be changed is chosen at random from those that would give an equally good improvement. Such non-determinism makes it very unlikely that the algorithm makes the same sequence of changes over and over.

You should be able to implement this algorithm relatively easily using your representation of clauses and assignments from the first two parts of the project.

You must test your satisfiability checker on the following problems:

1. The following set of clauses, from the DIMACS file format specification:

$$(x_1 \lor x_3 \lor \neg x_4) \land (x_4) \land (x_2 \lor \neg x_3)$$

2. N-Queens for N at least 4. Try larger values and see what your solver can do. Mine does $N = 12$ pretty quickly, but is still working on $N = 16$. . .

3. At least two of the following examples from John Burkhardt's repository:

   - `quinn.cnf`: 16 variables and 18 clauses
   - `par8-1-c.cnf`: 64 variables and 254 clauses
   - `aim-50-1_6-yes1-4.cnf`: 50 variables, 80 clauses.
   - `zebra_v155_c1135.cnf`: A formulation of the "Who Owns the Zebra" puzzle, with 155 variables and 1135 clauses.
   - The pigeonhole problem that I originally listed (`hole6.cnf`) is unsatisfiable according to its comments. So let's say that you should test a few *satisfiable* pigeonhole problems with different numbers of pigeons (and the same number of holes). See below. You should also know why giving an unsatisifiable problem to a local search SAT solver is not a productive thing to do.

   These files are available in BlackBoard along with the project description and submission form.

For each problem, your program must:

1. Have a method or function that creates the set of clauses for the problem, either programmatically (in code) or by reading them from a DIMACS CNF file.

2. Call your satisfiability checker with reasonable parameters and print the results as well as informative messages so that we know what is going on.

For the smaller problems, you can print lots of information about what's happening. For the larger problems you should probably turn off the tracing.

For the first problem, you can easily create the set of clauses by hand.

For N-Queens, you may write code that generates the clauses representing the constraints or you may read them from files included with the project description. I wrote the code to create clauses and then write them to a file, FWIW.

For the Pigeonhole problems, you can write code to create them using the following approach:

- Let n be the number of pigeons and m be the number of holes.

- Variable (proposition) $x_{p,h}$ is true iff pigeon p is in hole h, where $1 \le p \le n$ and $1 \le h \le m$. To turn the pair p, h into a single number i, let $i = (p - 1) \times m + h$.

- Every pigeon must be in some hole. So for each pigeon p there is one clause with the appropriate variables, $x_{p,h}$, one for each hole h.

- No more than one pigeon in any hole. For each hole, and for every combination of two pigeons p_i and p_j, there is a clause saying that it is not the case that p_i and p_j are both in that hole (which isn't a clause if you translate the English directly into logic).

- You only need to try problems where $n = m$ for this project. If $n > m$ then the problem is unsatisfiable, and if $n < m$ then then it's easier to solve.

For other respository problems, you must read them from the DIMACS CNF files included with the project description (or direct from the repository, links above). Just tell us what you did in your README if it's not clear from how your program runs.

WalkSAT

Another local search algorithm for testing satisfiability is described in (Selman and Kautz, 1993).[2] They call it the "Random Walk Strategy" in the paper, but it is now known as WalkSAT and is described in AIMA Fig. 7.15.

WalkSAT is a simple modification of GSAT:

- With probability p, pick a variable occuring in some unsatisifed clause and flip its truth assignment.

- With probability $1 - p$, follow the standard GSAT scheme, *i.e.*, make the best possible move.

That first part means that WalkSAT will sometimes make a change even if it decreases the value of the state, which GSAT will never do. This is the "random walk" aspect of method.

You can read the paper. It describes nicely how GSAT is like a form of simulated annealing. And it describes prior work by Papadimitriou on a pure random walk strategy (which doesn't work for general clauses).

This algorithm is also easy to implement once you have GSAT, but not required.

---

[2]Yes, that "Kautz" is Prof. Kautz from Rochester, although he was at Bell Labs when he did that work.

Project Submission

Your project submission MUST include the following:

1. A README.txt file or PDF document describing:

   (a) Any collaborators (see below)
   (b) How to build your project
   (c) How to run your project's program(s) to demonstrate that it/they meet the requirements

2. All source code for your project. Eclipse projects must include the project settings from the project folder. Non-Eclipse projects must include a `Makefile` or shell script that will build the program per your instructions, or at least have those instructions in your README.txt.

3. A completed copy of the submission form posted with the project description. Projects without this will receive a grade of 0. If you cannot complete and save a PDF form, submit a text file containing the questions and your (brief) answers.

Writeups other than the instructions in your README and your completed submission form are **not** required.

We must be able to cut-and-paste from your documentation in order to build and run your code. **The easier you make this for us, the better grade your will be.** It is your job to make both the building and the running of programs easy and informative for your users.

Programming Practice

Use good object-oriented design. No giant `main` methods or other unstructured chunks of code. Comment your code liberally and clearly.

You may use Java, Python, or C/C++ for this project. I recommend that you use Java. Any sample code we distribute will be in Java. Other languages (Haskell, Clojure, Lisp, *etc.*) by arrangement with the TAs only.

You may **not** use any non-standard libraries. Python users: that includes things like NumPy. Write your own code—you'll learn more that way.

If you use Eclipse, make it clear how to run your program(s). Setup Build and Run configurations as necessary to make this easy for us. Eclipse projects with poor configuration or inadequate instructions will receive a poor grade.

Python projects must use Python 3 (recent version, like 3.7.x). Mac users should note that Apple ships version 2.7 with their machines so you will need to do something different.

If you are using C or C++, you should use reasonable "object-oriented" design not a mish-mash of giant functions. If you need a refresher on this, check out the C for Java Programmers guide and tutorial. You **must** use "`-std=c99 -Wall -Werror`" **and** have a clean report from `valgrind`. Projects that do not follow both of these guidelines will receive a poor grade.

Late Policy

Late projects will **not** be accepted. Submit what you have by the deadline. If there are extenuating circumstances, submit what you have before the deadline and then explain yourself via email.

If you have a medical excuse (see the course syllabus), submit what you have and explain yourself as soon as you are able.

Collaboration Policy

You will get the most out of this project if you write the code yourself.

That said, collaboration on the coding portion of projects is permitted, subject to the following requirements:

- Groups of no more than 3 students, all currently taking CSC242.

- You must be able to explain anything you or your group submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.

- One member of the group should submit code on the group's behalf in addition to their writeup. Other group members should submit only a README indicating who their collaborators are.

- All members of a collaborative group will get the same grade on the project.

Academic Honesty

Do not copy code from other students or from the Internet.

Avoid Github and StackOverflow completely for the duration of this course.

There is code out there for all these projects. You know it. We know it.

Posting homework and project solutions to public repositories on sites like GitHub is a violation of the University's Academic Honesty Policy, Section V.B.2 "Giving Unauthorized Aid." Honestly, no prospective employer wants to see your coursework. Make a great project outside of class and share that instead to show off your chops.

References

Selman, B., H. Levesque, and D. Mitchell (1992). A New Method for Solving Satisfiability Problems.In *Proceedings of AAAI-92*, pp. 441-446. PDF

Selman, B., H. Kautz, and B. Cohen (1996). Local Search Strategies for Satisfiability Testing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*, Johnson, D.S., and M. A. Trick, eds., pp. 521-532. PDF

A   DIMACS CNF File Format

The CNF file format is an ASCII file format.

1. The file may begin with comment lines. The first character of each comment line must be a lower case letter "c". Comment lines typically occur in one section at the beginning of the file, but are allowed to appear throughout the file.

2. The comment lines are followed by the "problem" line. This begins with a lower case "p" followed by a space, followed by the problem type, which for CNF files is "cnf", followed by the number of variables followed by the number of clauses.

3. The remainder of the file contains lines defining the clauses, one by one.

4. A clause is defined by listing the index of each positive literal, and the negative index of each negative literal. Indices are 1-based, and for obvious reasons the index 0 is not allowed.

5. The definition of a clause may extend beyond a single line of text.

6. The definition of a clause is terminated by a final value of "0".

7. The file terminates after the last clause is defined.

For example, here is the set of clauses given above as the second test problem for satisfiability:
$$(x_1 \lor x_3 \lor \neg x_4) \land (x_4) \land (x_2 \lor \neg x_3)$$
The DIMACS CNF text version of this is the following:

```
c Example CNF format file
c
p cnf 4 3
1 3 -4 0
4 0 2
-3
```

Some odd facts include:

- The definition of the next clause normally begins on a new line, but may follow, on the same line, the "0" that marks the end of the previous clause.

- In some examples of CNF files, the definition of the last clause is not terminated by a final "0".

- In some examples of CNF files, the rule that the variables are numbered from 1 to N is not followed. The file might declare that there are 10 variables, for instance, but allow them to be numbered 2 through 11.

This description is based on "Satisfiability Suggested Format" from the Second DIMACS Implementation Challenge: 1992-1993 site at Rutgers, dated May 8, 1993, and from a page by John Burkardt at FSU.