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). We will see clauses in Lecture 2.3, but you probably shouldn't wait until then to read about them in the textbook so that you can get going on this project.

Ok, so if you've read about clauses in the textbook and perhaps seen them in class, you know that a clause is a disjunction (an "or") of literals, where a literal is either an atomic proposition or the negation of an atomic proposition.

You should also know restricting ourselves to clauses is not a problem 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). The procedure for doing that is in the textbook and we will see it in class.

Requirements

1. (25%) Design and implement a representation of clauses.
2. (50%) Implement the TT-ENTAILS model-checking algorithm and demonstrate it on several examples.
3. (25%) 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.

Note that there are several components to the project, and several ways that you can demonstrate your programs. **You must make this clear and easy for us** so that we can give you the points.

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 we think of "assigning" them 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:

$$(x_1 \vee x_3 \vee \neg x_4) \wedge (x_4) \wedge (x_2 \vee \neg x_3)$$

is equivalent to the following set of three clauses:

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

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. But the design is important. However you do it, you **must** have a class (struct for C) representing clauses, with appropriate methods ("methods" for C).

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 sets of integers.

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, or equivalently, whether β *follows logically from* α. Since all our sentences will be clauses, we can 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 sequences of boolean values. The element at index i in the sequence 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."

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) or not. Similarly, whether an assignment does not satisfy a clause (makes it false) nor not.

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: $B_{1,2} \Leftrightarrow P_{1,1} \vee P_{2,2} \vee P_{1,3}$

Continues on next page. . .

[1] As noted in class, 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 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 $P_{2,2}$.

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. Give us whatever directions are needed to run the programs.
5. Print informative messages so that we know what is going on when we run them.

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.

DPLL

Another systematic propositional model-checking algorithm is DPLL (Davis and Putnam, 1962; Davis, Logemann, and Loveland, 1962). It is described in AIMA Sec. 7.6.1 and Fig. 7.17.

Compared to the basic "entailment by enumeration" method, DPLL does early pruning of inconsistent states, uses several problem- and domain-independent heuristics for selecting clauses and propositions (variables), and performs inference (constraint propagation) to reduce the amount of search. You may recognize these techniques from another class of problem-solving methods seen in this unit of the course. AIMA describes additional methods that can be used to scale the algorithm up to huge problems.

This algorithm is not hard to implement once you have the representation of clauses and the basic methods for clauses and assignments, but it is not required for the project.

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 problem of determining whether a set of clauses is satisfiable, usually referred to simply as "SAT," was the first problem that was proven to be NP-complete (Cook, 1971). This means that it is almost certain that there is no tractable algorithm that is guaranteed to find the answer to any SAT problem. The local search approach to testing satisfiability was described in (Selman, Levesque, and Mitchell, 1992) and is shown in Figure 1.

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

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:

$$(x_1 \vee x_3 \vee \neg x_4) \wedge (x_4) \wedge (x_2 \vee \neg x_3)$$

2. N-Queens for N from 4 to however big your checker can handle in a reasonable amount of time. For example, my implementation of GSAT does $N = 12$ pretty quickly, but is still working on $N = 16\ldots$

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.
   - Pigeonhole problems (assign n pigeons to m holes with no more than 1, or k, pigeons per hole) as big as your checker can handle in a reasonable time. Note that the `hole6.cnf` problem in the repository is *unsatisfiable* according to its comments. You should know why giving an unsatisifiable problem to a local search SAT solver is not a productive thing to do. See below for ways to generate your own (satisfiable) pigeonhole problems.

   These files, including pigeonhole problems, 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. Ask the user for MAX-TRIES and MAX-FLIPS, as well as any other necessary parameters (for example, N for N-Queens). Your programs **must** suggest or provide reasonable default values for these parameters.
3. 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, or at least make it optional.

It is **your responsibility** to make your program clear and easy to use.

For the first problem, you can easily write code to 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 the clauses and then write them to a file, FWIW.

For 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 $1 \leq p \leq n$ and $1 \leq h \leq m$. 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 (disjunction) 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.

Several satisifiable pigeonhole problems are included with the project downloads. I wrote a program to do what is described above and had it write the resulting set of clauses to a file. You could do that also.

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.

Additional Requirements and Policies

The short version:

- You may use Java, C/C++, or Python. I STRONGLY recommend Java.
- You MUST use good object-oriented design (yes, even in Python).
- There are other language-specific requirements detailed below.
- You must submit a ZIP including your source code, a README, and a completed submission form by the deadline.
- You must tell us how to build your project in your README.
- You must tell us how to run your project in your README.
- Projects that do not compile will receive a grade of **0**.
- Projects that do not run or that crash will receive a grade of **0** for whatever parts did not work.
- Late projects will receive a grade of **0** (see below regarding extenuating circumstances).
- You will learn the most if you do the project yourself, but collaboration is permitted in groups of up to 3 students.
- Do not copy code from other students or from the Internet.

Detailed information follows. . .

Programming Requirements

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

  - In Java, C++, or Python, you MUST have well-designed classes.
  - Yes, even in Python.
  - In C, you must have well-designed "object-oriented" data structures (refresher: C for Java Programmers guide and tutorial)
- No giant `main` methods or other unstructured chunks of code.
- Your code should use meaningful variable and function/method names and have plenty of meaningful comments. But you know that...

Submission Requirements

You MUST submit your project as a ZIP archive containing the following items:

1. The source code for your project.
2. A file named `README.txt` or `README.pdf` (see below).
3. A completed copy of the submission form posted with the project description (details below).

Your README **MUST** include the following information:

1. The course: "CSC242"
2. The assignment or project (*e.g.*, "Project 1")
3. Your name and email address
4. The names and email addresses of any collaborators (per the course policy on collaboration)
5. Instructions for building and running your project (see below).

The purpose of the submission form is so that we know which parts of the project you attempted and where we can find the code for some of the key required features.

- Projects without a submission form or whose submission form does not accurately describe the project 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.

Project Evaluation

You **MUST** tell us in your README file how to build your project (if necessary) and how to run it.

Note that we will NOT load projects into Eclipse or any other IDE. We **MUST** be able to build and run your programs from the command-line. If you have questions about that, go to a study session.

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 your grade will be.** It is **your** job to make the building of your project easy and the running of its program(s) easy and informative.

For **Java** projects:

- The current version of Java as of this writing is: 16.0.2 (OpenJDK)
- If you provide a `Makefile`, just tell us in your README which target to make to build your project and which target to make to run it.
- Otherwise, a typical instruction for building a project might be:

  ```
  javac *.java
  ```

  Or for an Eclipse project with packages in a `src` folder and classes in a `bin` folder, the following command can be used from the `src` folder:

  ```
  javac -d ../bin `find . -name '*.java'`
  ```

- And for running, where `MainClass` is the name of the main class for your program:

  ```
  java MainClass [arguments if needed per README]
  ```

  or

  ```
  java -d ../bin pkg.subpkg.MainClass [arguments if needed per README]
  ```

- You **MUST** provide these instructions in your README.

For **C/C++** projects:

- You MUST use at least the following compiler arguments:

  ```
  -Wall -Werror
  ```

- If you provide a `Makefile`, just tell us in your README which target to make to build your project and which target to make to run it.
- Otherwise, a typical instruction for building a project might be:

  ```
  gcc -Wall -Werror -o EXECUTABLE *.c
  ```

  where `EXECUTABLE` is the name of the executable program that we will run to execute your project.
- And for running:

  ```
  ./EXECUTABLE [arguments if needed per README]
  ```

- You **MUST** provide these instructions in your README.
- Your code should have a clean report from `valgrind`. If we have problems running your program and it doesn't have a clean report from `valgrind`, we are going to assume that your code is wrong. If you are a C/C++ programmer and don't know about `valgrind`, look it up. It is an essential tool.

For **Python** projects:

I strongly recommend that you NOT use Python for projects in CSC242. All CSC242 students have at least two terms of programming in Java. The projects in CSC242 involve the representations of many different types of objects with complicated relationships between them and algorithms for computing with them. That's what Java was designed for.

But if you insist...

- The latest version of Python as of this writing is: 3.9.6 (python.org).
- You must use Python 3 and we will use a recent version of Python to run your project.

- You may **NOT** use any non-standard libraries. This includes things like NumPy. Write your own code—you'll learn more that way.
- We will follow the instructions in your README to run your program(s).
- You **MUST** provide these instructions in your README.

For **ALL** projects:

We will **NOT** under any circumstances edit your source files. That is your job.

Projects that do not compile will receive a grade of **0**. There is no way to know if your program is correct solely by looking at its source code (although we can sometimes tell that is incorrect).

Projects that do not run or that crash will receive a grade of **0** for whatever parts did not work. You earn credit for your project by meeting the project requirements. Projects that don't run don't meet the requirements.

Any questions about these requirements: go to study session **BEFORE** the project is due.

Late Policy

Late projects will receive a grade of **0**. You MUST 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

I assume that you are in this course to learn. You will learn the most if you do the projects YOURSELF.

That said, collaboration on 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 MUST submit a README (only) indicating who their collaborators are.
- All members of a collaborative group will get the same grade on the project.

Academic Honesty

I assume that you are in this course to learn. You will learn nothing if you don't do the projects yourself.

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

Cook, S. (1971). The complexity of theorem proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pp. 151–158. doi:10.1145/800157.805047

Davis, M., G. Logemann, and G. Loveland (1962). A Machine Program for Theorem Proving. *Communications of the ACM* 5(7), pp. 394–397. doi:10.1145/368273.368557

Davis, M., and H. Putnam (1960). A Computing Procedure for Quantification Theory. *J. ACM* 7(3), pp. 201–214. doi:10.1145/321033.321034

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:

$$(x_1 \vee x_3 \vee \neg x_4) \wedge (x_4) \wedge (x_2 \vee \neg x_3)$$

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
```

The first two lines are comments. The third line says that the file defines a set of CNF three CNF clauses involving a total of four variables (proposition symbols). The next line defines the first clause. The remaining two clauses are split over the last two lines.

Some odd facts about the DIMACS format include the following:

- 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.