CSC242: Intro to AI
Project 2: Propositional Logic
Model Checking and Satisfiability

In this project we will investigate using propositional logic to represent knowledge and do inference. Propositional logic is a formal language for describing the world—a "programming language" for knowledge. It is described in AIMA Sec. 7.4, and we will see it in class, but here is a quick introduction.

- **Propositions** are things that can be true or false, such as "it is raining" or "there is life on Mars" or "$2 + 2 = 5$".
- In propositional logic, we represent propositions using **proposition symbols**. Each proposition symbol corresponds to a proposition. As the texbook says, the names are arbitrary, but are often chosen to have some mnemonic value. For example: $Raining$ for "it is raining." But we could also use symbols like P, Q, R, or anything else.
- Proposition symbols are the **atomic sentences** (expressions) of the language of propositional logic.
- **Complex sentences** are constructed from simpler ones using **connectives** (operators). The textbook describes five common connectives:
  - **negation** ($\neg$, "not")
  - **conjunction** ($\wedge$, "and")
  - **disjunction** ($\vee$, "or")
  - **implication** ($\Rightarrow$, "implies")
  - **biconditional** ($\Leftrightarrow$, "if and only if")

  See AIMA Sec. 7.4.1 for full details of these operators and the syntax and semantics of propositional logic.
- An assignment of true or false to each of the propositions in a sentence or set of sentences is called a **possible world**. That is, such an assignment is one way that the world can be.
- A possible world that makes a sentence or set of sentences is true is called a **model** of the sentence or set of sentences. Note that the textbook confusingly uses "model" to refer to both possible worlds and possible worlds in which the sentence(s) are true.

- A sentence of set of sentences is **satisfiable** if it has at least one model (if it is true in at least possible world). It is **unsatisfiable** if it has no models (it is not true in any possible world).
- **Inference** is the process of determing what follows from our knowledge. Suppose that our knowledge is represented as a set of sentences KB (for "knowledge base"). We say that some other sentence or set of sentences β **follows logically** from our knowledge if whenever KB is true, β must also be true We say that KB **entails** β, written "$KB \models \beta$".
- If $KB \models \beta$, then every model of KB must also be a model of β. So one way of testing entailment is to enumerate all the models of KB and check that they are all also models of β. This is called **model checking**, and is shown in AIMA Fig. 7.10 as the procedure TT-ENTAILS?.
- A restricted form of propositional logic is called **conjunctive normal form (CNF)**.
  - A **literal** is either a proposition symbol (*e.g.*, P) or the negation of a proposition symbol (*e.g.*, $\neg P$). No other operators.
  - A **clause** is a disjunction ("or") of literals (*e.g.*, $P \vee Q \vee \neg R$).
  - A sentence in CNF is a conjunction ("and") of clauses, sometimes written as a set of clauses.
- Every sentence of propositional logic is **logically equivalent to** (has exactly the same models as) some sentence in CNF. The four-step procedure for converting to CNF is shown in AIMA Sec. 7.5.2 on p. 227 (4th ed.).

Requirements

1. (25%) Design and implement a representation of clauses and KBs, including printing them as described below.
2. (50%) Implement the TT-ENTAILS systematic model-checking algorithm and demonstrate it on several examples.
3. (25%) Implement the WalkSAT local search satisfiability testing algorithm and demonstrate it 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.

Without loss of generality, we may assume that our atomic propositions use the symbols x_1, x_2, and so on. If you have a knowledge base 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 (conjunction) 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:

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

Printing KBs and Clauses: Sets must be printed with each set enclosed in square brackets, each pair of elements separated by a comma and a space (standard Java `Set.toString`). For example, the previous KB might be printed as:

```
[[1, 3, -4], [-3, 2], [4]]
```

Note that the elements of a set are **not** ordered, so they may be printed in different orders by different implementations of the set.

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 representing clauses, with appropriate methods.

Go to study session early if you want help with this.

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 KB logically entails another sentence or set of sentences β, or equivalently, whether β follows logically from KB. Since all our sentences will be clauses, we can do model checking with clauses in this project.

I suggest that you represent possible worlds 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 a sentence is true or false in a (complete) possible world. This is called function PL-TRUE in AIMA Fig. 7.10.

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 $P_{2,2}$.

   Continues on next page...

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

R5: $B_{2,1}$

Show that this knowledge base entails $P_{2,2} \lor P_{3,1}$, 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 clauses (CNF). 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. For each query, you **must**

   - Print the KB of clauses on a single line (perhaps a long line, see below).
   - Print the query clause on a single line.
   - Print `true` if the query is entailed or `false` if it is not entailed.

5. Your program(s) should print informative messages so that we know what is going on when we run them. **It is your responsability to make it clear what is happening.**

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.

You know from Unit 1 that local search is an alternative to computationally expensive exhaustive search. The local search approach to testing satisfiability is described in AIMA Sec. 7.6.2. The original algorithm was GSAT (Selman, Levesque, and Mitchell, 1992; see Appendix B). WalkSAT (Selman and Kautz, 1993), shown in AIMA Fig. 7.18, is based on GSAT (see Appendix C).

For this project, **you MUST implement the WalkSAT algorithm with random restarts**.

States are truth assignments (possible worlds). They use a complete state formulation, so states are complete truth assignments (a value for every atomic proposition symbol). Actions are picking an atomic proposition symbol 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 easy to check. WalkSAT does hillclimbing for at most max_flips steps per run.

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. Two quick notes:

- You must be able to randomly select a "clause from [the set of] $clauses$ that is false in [the] $model$." This is easy to do.
- You must be able to randomly select a "symbol from [a] $clause$." This is also easy to do.
- You must be able to find "whichever $symbol$ in $clause$ maximizes the number of satisfied clauses." This is a bit harder to compute.

For all of these, clear and correct implementation is more important than fast execution for this course and project.

You **MUST** also use the "Random Restarts" strategy. But that is easy: in a loop for 1 up to some number max_tries, call WalkSAT until it returns a solution.

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 WalkSAT with $p = 0.5$, $max\_flips = 100$, and $max\_tries = 100$ solves $N = 11$ pretty quickly (it depends), but is still working on $N = 12\ldots$

3. Pigeonhole problems: assign n pigeons to m holes with no more than 1 (or, in general, k), pigeons per hole. Try as big as your checker can handle in a reasonable time. My implementation of WalkSAT with $p = 0.5$, $max\_flips = 100$, and $max\_ttries = 100$ can solve problems with $n = m = 15$ in about 30 seconds (it depends).

DIMACS files for these 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. Print the KB on a single line as described above.
3. Ask the user for p, 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.
4. Call your satisfiability checker with those parameters and print the result (satisfying assignment, or that no solution was found).
5. Print informative messages so that we know what is going on. 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.

Additional Requirements and Policies

The short version:

- You may **only** use Java 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.

Detailed information follows...

Programming Requirements

- You may use Java or Python for this project.
  - I **STRONGLY** recommend that you use Java.
  - Any sample code we distribute will be in Java.
- You **must** use good object-oriented design.
  - You **must** have well-designed classes (and perhaps interfaces).
  - Yes, even in Python.
- No giant `main` methods or other unstructured chunks of code.

  - Yes, even in Python.

- Your code should use meaningful variable and function/method names and have plenty of meaningful comments.
  - I can't believe that I even have to say that.
  - And yes, even in Python.

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: 20.0.2 (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 **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: 3.11.4 (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 or pandas. 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:

- Teams of no more than 3 students, all currently taking CSC242.
- Any team member **must** be able to explain anything they or their team submits, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the team should submit code on the team's behalf in addition to their writeup. Other team members **must** submit a README (only) indicating who their collaborators are.
- All members of a collaborative team 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.

B GSAT

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 from (Selman, Levesque, and Mitchell, 1992)

C From GSAT to WalkSAT

The following is the description of GSAT from (Selman and Kautz, 1993):

Procedure GSAT
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
        Flip any variable in T that results in greatest
                decrease (can be 0 or negative)
                in the number of unsatisfied clauses
        end for
    end for
    return "No satisfying assignment found"
end

What later became known as WalkSAT is then described as the following "Random Walk Strategy" in (Selman and Kautz, 1993):

Random Walk Strategy
With probability p, pick a variable occuring in some
    unsatisfied clause and flip its truth assigment.
With probability $1 - p$, follow the standard GSAT scheme,
    i.e., make the best possible local move

Compare these to the AIMA Fig. 7.18 description of WalkSAT, which is what you need to implement for this project (with random restarts).