**CSC242: Intro to AI**
**Project 2: Automated Reasoning**

In this project, you will implement at least two inference methods for Propositional Logic and demonstrate them on some example problems.

# Representational Preliminaries

You should think about how you would represent Propositional Logic sentences in your program. This is one of the things that makes AI programming interesting. It's not rocket science, but it does force you to think clearly about different levels of representation: Java as programming language, Propositional Logic as "programming language", the abstraction hierarchy of your classes, *etc.*. And let me just say that doing it correctly in Java has some subtleties (for example, be careful with the `equals` and `hashCode` methods if you're using hashtables). Doing it relatively efficiently is an additional challenge.

But give it a try. I will provide sample code for representing Propositional Logic sentences in Java in case you get stuck. You may use these as a foundation for building the rest of your implementation. But again: **try it yourself first**, then peek at mine.

It is also possible to write parsers that read various textual representations of Propositional Logic sentences and create the internal representations (data structures) used by your programs. These include, for example, a bracketed, prefix Lisp- or Scheme-like notation, an infix notation like we've used in class, and things like the DIMACS encoding of SAT problems. This is useful but not necessary for this project. For this project you may create the knowledge bases yourself in code.

# Part I: Basic Model Checking

The first technique you must implement is the "truth-table enumeration method" described in AIMA Figure 7.10. The algorithm is conceptually straightforward—you may not really even need their pseudo-code. If you do follow their example, you'll need to translate their Lisp-like `FIRST`/`REST` method for traversing a list into something you can do in Java (or whatever). You could certainly implement `FIRST` using `get` and `REST`

using `subList`. Or you might use an array or list and pass the index of the next element as parameter in the recursive calls.

You will also need to represent a "model" (a.k.a. possible world, or assignment of truth values). And then you need to use that representation in the implementation of the algorithm. This is a very important design point. I suggest that you do something clear and correct but not necessarily efficient first, then improve it as time permits and your problems demand.

You must test your implementation on at least three (for undergraduates) or four (for graduate students) of the problems described below.

# Part II: Advanced Propositional Inference

For the second part of this assignment, you must implement one of the other techniques for Propositional Inference described in class and in the textbook. Your choices are:

- The DPLL algorithm for checking satisfiability (AIMA Figure 7.17)
- The WalkSAT algorithm, also for SAT checking (AIMA Figure 7.18)
- A resolution-based theorem prover (AIMA Figure 7.12)
- A theorem prover based on other inference rules (if you do this, be sure to explain what you're doing in detail in your writeup)

Note that the forward-chaining algorithm (AIMA Figure 7.15) can only be used if your knowledge base consists solely of definite clauses. That might be ok for some problems...

You can use the same classes for representing Propositional Logic sentences as you did for Part I (mine or your own).

Several of these techniques require the knowledge base to be converted to clauses (conjunctive normal form, CNF). You can write a program that converts arbitrary sentences to CNF. In fact, I will give you code that does this, but **you should think about how to do it yourself** before you look at mine. Or you can arrange for the problems that you solve to be expressed as clauses in the first place.

As with Part I, you must test your implementation on at least three (for undergraduates) or four (for graduate students) of the problems described below. Use the same problems and compare the techniques in your writeup. If you can't use the same problems, explain why in your writeup.

# Sample Problems

The following problems are all amenable to solution using Propositional Logic using both theorem proving and satisfiability testing approaches.

All students must do problems (1) and (2) with both of their implementations. They are very simple.

Undergraduates must then choose **one** other problem to do (with both implementations).

Graduate students must choose **two** other problems to do (with both implementations).

You may do additional problems for extra credit, up to 10% each at the grader's discretion, up to a total of 20% extra credit.

1. **Modus Ponens**: Show that $\{P, P \Rightarrow Q\} \vDash Q$.

2. **Wumpus World (Simple)**: Consider the following facts using the Wumpus World predicates from AIMA:

   $$\begin{aligned} &\neg P_{1,1} \\ &B_{1,1} \iff (P_{1,2} \vee P_{2,1}) \\ &B_{2,1} \iff (P_{1,1} \vee P_{2,2} \vee P_{3,1}) \\ &\neg B_{1,1} \\ &B_{2,1} \end{aligned}$$

   Prove whether $P_{1,2}$ is true or not (that is, whether there is a pit at location [1,2]).

   The textbook gives several other examples of knowledge bases and queries using the Wumpus World that you can also try.

3. **Horn Clauses** (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?

4. **Liars and Truth-tellers** (adapted from OSSMB): Three people, Amy, Bob, and Cal, are each either a liar or a truth-teller. Assume that liars always lie, and truth-tellers always tell the truth.

   (a) OSSMB 82-12:

   - Amy says, "Cal and I are truthful."
   - Bob says, "Cal is a liar."
   - Cal says, "Bob speaks the truth or Amy lies."

   What can you conclude about the truthfulness of each?

   (b) OSSMB 83-11:

   - Amy says, "Cal is not honest."
   - Bob says, "Amy and Cal never lie."
   - Cal says, "Bob is correct."

   What can you conclude about the truthfulness of each?

5. **More Liars and Truth-tellers** (adapted from JRM14 392): At Liars Anonymous, there was a gathering of liars in extreme denial and fully reformed truth-tellers. Each will profess his/her truthfulness. However, when they are all together, their statements about each other reveal the truth:

   - Amy says, "Hal and Ida are truth-tellers."
   - Bob says, "Amy and Lee are truth-tellers."
   - Cal says, "Bob and Gil are truth-tellers."
   - Dee says, "Eli and Lee are truth-tellers."
   - Eli says, "Cal and Hal are truth-tellers."
   - Fay says, "Dee and Ida are truth-tellers."
   - Gil says, "Eli and Jay are liars."
   - Hal says, "Fay and Kay are liars."
   - Ida says, "Gil and Kay are liars."

- Jay says, “Amy and Cal are liars.”
- Kay says, “Dee and Fay are liars.”
- Lee says, “Bob and Jay are liars.”

Which are liars and which are truth-tellers?

6. **The Doors of Enlightenment** (from CRUX 357): There are four doors $X$, $Y$, $Z$, and $W$ leading out of the Middle Sanctum. At least one of them leads to the Inner Sanctum. If you enter a wrong door, you will be devoured by a fierce dragon. Well, there were eight priests $A$, $B$, $C$, $D$, $E$, $F$, $G$, and $H$, each of whom is either a knight or a knave. (Knights always tell the truth and knaves always lie.) They made the following statements to the philosopher:

   - $A$: $X$ is a good door.
   - $B$: At least one of the doors $Y$ or $Z$ is good.
   - $C$: $A$ and $B$ are both knights.
   - $D$: $X$ and $Y$ are both good doors.
   - $E$: $X$ and $Z$ are both good doors.
   - $F$: Either $D$ or $E$ is a knight.
   - $G$: If $C$ is a knight, so is $F$.
   - $H$: If $G$ and $I$ (meaning $H$) are knights, so is $A$.

   (a) Smullyan’s problem: Which door should the philosopher choose?

   (b) Liu’s problem: The philosopher lacked concentration. All he heard was the first statement ($A$’s) and the last statement ($H$’s) plus two fragments:

   - $C$: $A$ and . . . are both knights.
   - $G$: If $C$ is a knight, . . .

   Prove that he had heard enough to make a decision.

7. **Wumpus World** (AIMA): Unfortunately, it is quite difficult to encode all the knowledge required to explore the Wumpus World in Propositional Logic. See AIMA Section 7.7 and pay attention to *frame axioms* and *successor state axioms*. These can be generated automatically (for a given size of world), but there are many of them, and if they need to be converted to CNF for your solver, the result is many, many clauses. In particular, I had trouble with the axioms for $WumpusAhead$.

   That said, you may want to try doing inference in the Wumpus World. One approach would be to start by simplifying the world, for example to concentrate on inferring the location of the pits (and then perhaps the gold, and then perhaps

the wumpus). In this case, you wouldn't worry about shooting the wumpus, and you wouldn't need to reason about $WumpusAhead$. You would learn a lot by doing this. You could infer some facts about the world given different sets of perceptions, and/or you could build an agent that explores on the basis of what it infers from its background knowledge and perceptions (AIMA Figure 7.20). You could implement the SATplan algorithm (AIMA Figure 7.22). You could try to scale up and handle the entire problem, as described in the textbook. Whatever you do, you must be prepared to explain in your writeup what you did, what worked and what didn't, and *why*.

8. **Sudoku**: Sudoku is described in AIMA Section 6.2.6 as a constraint satisfaction problem. However we know that there is a deep relationship between CSPs and Propositional Logic. So it's not surprising that you could use logical techniques like SAT solvers to solve sudoku puzzles. It's not *that* interesting a problem though—the challenge is mostly in writing a program that turns a sudoku into a set of PL sentences. That's cool, but not much AI involved. It's also very widely available on the Internet. Therefore you may do this problem for extra credit, but not as your third primary problem.

## 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 and build files for your project (see below)
3. A writeup describing your work in PDF format (see below)

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, you *must* make it so that we can build and run your project without Eclipse. Document exactly what needs to be done in your README (Makefile, `javac` or `gcc` incantations, whatever). Eclipse projects with no build instructions will receive 0.

Your code must build on Fedora Linux using recent versions of Java, Python, or `gcc`.

- This is not generally a problem for Java projects, but don't just submit an Eclipse project without the build and run instructions detailed above.
- Python projects must use Python 3 (recent version, like 3.6.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 must use "`-std=c99 -Wall -Werror`" and have a clean report from `valgrind`. And you'd better test on Fedora Linux or expect problems.

## Writing Up Your Work

As noted above, it is crucial that you present your work clearly, honestly, and in its best light. We will give lots of credit for good writeups. We will not like submissions with sloppy, unstructured, hard to read writeups that were clearly written at the last minute.

Your goal is to produce a technical report of your efforts for an expert reader. You need to convince the reader that you knew what you were doing and that you did it as best you could in the (entire) time available.

Write up what you did (and why). If you didn't get to something in your code, write about what you might have done. (There's always *something* you might have done.) If something didn't work, write about why and what you would do differently. But don't write "I would have started sooner." Your readers already know that.

Your report *must be your own words*. Material taken from other sources must be properly attributed if it is not digested and reformulated in your own words. **Plagiarism is cheating.**

Start your writeup early, at least in outline form. Document your design decisions as you make them. That way half the write-up is done by the time you're running the program. Don't forget to include illustrations of the program in action (traces, screenshots) and some kind of evaluation.

Your report must be typeset (not handwritten). Using LaTeX and BibTeX makes things look nice and is worth learning anyway if you don't know it, but it's not a requirement. Your report must be submitted as a PDF file. If you want to use a word processor, be sure to generate and submit a PDF from it, not the document file.

The length of the report is up to you. For a CSC242 assignment, I would say that 3–5 pages is the *minimum* that would allow you to convince the reader. Be sure to include screenshots and/or traces (which wouldn't count towards the 3-5 pages minimum of actual text).

## Late Policy

Don't be late. But if you are: 5% penalty for the first hour or part thereof, 10% penalty per hour or part thereof after the first.

## 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 their writeup.
- All members of a collaborative group will get the same grade on the coding component of the project.

You may NOT collaborate on your writeup. Your writeup must be your own work. Attribute any material that is not yours. Cite any references used in your writeup. Plagiarism is cheating.