Clue Deduction: an introduction to satisfiability reasoning

Todd W. Neller[*], Zdravko Markov, Ingrid Russell, Dave Musicant[†]

1 Introduction

Clue®[1] is a mystery-themed game of deduction. The goal of the game is to be the first player to correctly name the contents of a case file: the murder suspect, the weapon used, and the room the murder took place in. There are 6 possible suspects, 6 possible weapons, and 9 possible rooms, each of which are pictured on a card. One card of each type is chosen randomly and placed in a "case file" envelope without being revealed to any player. All other cards are dealt out face-down to the players. Each player takes on the identity of one of the suspects.

Each player thus begins with private knowledge that their dealt cards are not in the case file. As the game proceeds, players suggest possible suspect, weapon, and room combinations, and other players refute these suggestions by privately revealing such cards to the suggester. This type of game is called a **knowledge game**, and the basic knowledge of the game may be expressed using **propositional logic**.

In this project, we will study:

- Propositional logic syntax (structure) and semantics (meaning),
- Conjunctive normal form (CNF),
- Resolution theorem proving, and
- Application of a SAT solver to Clue.

The result will be a reasoning engine which yields "expert" deductions for the play of Clue. Additionally, we will discuss deductive learning, inductive learning, knowledge acquisition, and possible advanced projects.

2 Propositional Logic

In this section, we give a brief overview of **propositional logic** (a.k.a. **Boolean logic** and **sentential logic**). For a more complete introduction, see Chapter 7 of your textbook.

Propositional logic is a simple logic based upon the Boolean values *true* and *false*. Many programmers will find propositional logic intuitive from exposure to Boolean types and logical operators which occur frequently in programming.

In discussing a logic, one needs to address both the **syntax** (structure) and the **semantics** (meaning) of the language. Just as the syntax of a programming language defines rules for what constitutes a well-formed program, the syntax of a logical language defines rules for what constitutes a well-formed

[*]Corresponding author: tneller@gettysburg.edu, Gettysburg College, Department of Computer Science, Campus Box 402, Gettysburg, PA 17325-1486

[†]Made modifications for Python, swapped in cryptominisat for zchaff, other smaller changes

[1]A.k.a. Cluedo® in other parts of the world

sentence of the logic. Just as programming language specifications help the programmer understand the expected behavior of a program, the semantics of a logical language define the meaning of well-formed sentences of the logic. We will present the syntax and semantics of propositional logic one simple piece at a time.

The **atomic sentence** is the simplest fundamental unit of the language. The syntax of an atomic sentence is either the constant *true*, the constant *false*, or a symbol. Each symbol is associated with a statement which can be true or false. For example, $\mathrm{pi}_{\mathrm{wh}}$ may symbolize the statement that "The Mrs. White player is holding the Lead Pipe card." This statement may be true or false. In general, c_p will symbolize the statement "The card c is in place p."

Often, the symbols used for atomic sentences are capital alphabetic characters $A, B, C, ..., Z$, but we are free to choose whichever symbols are convenient. However, just as with commenting variables in a program, it is important to attach meanings to these symbols to ground them in the reality they represent.

> **Example Problem:** Suppose that liars always speak what is false, and truth-tellers always speak what is true. Further suppose that Amy, Bob, and Cal are each either a liar or truth-teller. Amy says, "Bob is a liar." Bob says, "Cal is a liar." Cal says, "Amy and Bob are liars." Which, if any, of these people are truth-tellers?

The atomic sentences for this problem are as follows:

- A - Amy is a truth-teller.
- B - Bob is a truth-teller.
- C - Cal is a truth-teller.

Negation ($\neg$, "not"): More complex sentences may be built from these. The **negation** of A, denoted $\neg A$ (read "not" A), means the *opposite* of A, and thus means "Amy is *not* a truth-teller." Given our constraint that Amy is either a truth-teller or liar, this also means "Amy is a liar." The negation operator $\neg$ operates on a single sentence, negating its meaning. Thus for propositional logic sentence s, $\neg s$ has the opposite truth value. We may thus express the semantics of $\neg$ in a truth table:

s	$\neg s$
false	true
true	false

A **literal** is an atomic sentence or its negation. For example, A, $\neg A$, B, $\neg B$, C, and $\neg C$ are all literals.

Conjunction ($\wedge$ "and"): We have seen how "Amy is a liar." can be represented as $\neg A$. Similarly, "Bob is a liar." can be represented as $\neg B$. To say "Amy and Bob are liars." is to say that both $\neg A$ and $\neg B$ are true. We denote this **conjunction** of sentences as $\neg A \wedge \neg B$ (read "not A and not B"). If either or both of the two subsentences $\neg A$ or $\neg B$ are false, then the conjunction is false as well. The semantics of $\wedge$ for sentences s_1 and s_2 are thus expressed in this truth table:

s_1	s_2	$s_1 \wedge s_2$
false	false	false
false	true	false
true	false	false
true	true	true

Disjunction ($\vee$ "or"): Suppose that you are playing Miss Scarlet and Mrs. White suggests that Colonel Mustard committed the murder with the knife in the hall. Mr. Green refutes this suggestion by privately showing one of the cards Mustard, Knife, or Hall privately to Mrs. White. You hold the Mustard card. Thus you know that Mr. Green has either the Knife card or the Hall card. We may represent this knowledge as the **disjunction** $\mathrm{kn}_{\mathrm{gr}} \vee \mathrm{ha}_{\mathrm{gr}}$ (read "$\mathrm{kn}_{\mathrm{gr}}$ or $\mathrm{ha}_{\mathrm{gr}}$"). If either or both of the two subsentences $\neg A$ or $\neg B$ are true, then the disjunction is true as well. The semantics of $\vee$ for sentences s_1 and s_2 are thus expressed in this truth table:

s_1	s_2	$s_1 \vee s_2$
false	false	false
false	true	true
true	false	true
true	true	true

Conditional ($\Rightarrow$ "implies"): In our truth-teller and liar example, Amy says "Bob is a liar.". If Amy is a truth-teller (A), then we know that Bob is a liar ($\neg B$). We may represent this knowledge as the conditional (a.k.a. implication) $A \Rightarrow \neg B$ (read "A implies not B"). If the first subsentence s_1, the **antecedent**, is true, and the second subsentence s_2, the **consequent**, is false, then the conditional is false. Otherwise, it is true. The semantics of $\Rightarrow$ for sentences s_1 and s_2 are thus expressed in this truth table:

s_1	s_2	$s_1 \Rightarrow s_2$
false	false	true
false	true	true
true	false	false
true	true	true

Biconditional ($\Leftrightarrow$ "if and only if"): In the previous example, $A \Rightarrow \neg B$ is not the only fact we represent from Amy saying that Bob is a liar. We also know that if we find that Bob is, in fact, a liar, then Amy must be a truth-teller. That is $\neg B \Rightarrow A$. Combining these facts in a conjunction, we fully represent the knowledge as $(A \Rightarrow \neg B) \wedge (\neg B \Rightarrow A)$. This conjunction of two conditionals with antecedents and consequents reversed has a more compact representation as the **biconditional** $A \Leftrightarrow \neg B$. If the first subsentence s_1 and the second subsentence s_2 have the same truth value, the biconditional is true. Otherwise, it is false. The semantics of $\Leftrightarrow$ for sentences s_1 and s_2 are thus expressed in this truth table:

s_1	s_2	$s_1 \Leftrightarrow s_2$
false	false	true
false	true	false
true	false	false
true	true	true

As with arithmetic expressions, there is an order of precedence of the operators. The negation operator ($\neg$) has highest precedence, followed by $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$. Thus,

$$\begin{array}{rcl} \neg A \wedge B & \text{means} & (\neg A) \wedge B \\ A \vee B \Leftrightarrow C & \text{means} & (A \vee B) \Leftrightarrow C \\ \neg A \vee B \wedge C & \text{means} & (\neg A) \vee (B \wedge C) \\ & \textit{etc.} & \end{array}$$

Here, we've used parentheses to clarify operator grouping, but parentheses may also be used to group operations as desired (e.g. $\neg(A \wedge B)$), or improve readability (e.g. $(A \wedge B) \vee (C \wedge D) \vee (E \wedge F)$).

This is whole of the syntax and semantics may thus be summarized with the following Backus-Naur Form (BNF) grammar, where the *Symbol* set is chosen and defined by the logician:

$$\begin{array}{rcl} \mathit{Sentence} & \rightarrow & \mathit{Atomic\_Sentence} \mid \mathit{Complex\_Sentence} \\ \mathit{Atomic\_Sentence} & \rightarrow & \mathbf{true} \mid \mathbf{false} \mid \mathit{Symbol} \\ \mathit{Complex\_Sentence} & \rightarrow & \neg \mathit{Sentence} \mid \mathit{Sentence} \wedge \mathit{Sentence} \mid \mathit{Sentence} \vee \mathit{Sentence} \\ & & \mid \mathit{Sentence} \Rightarrow \mathit{Sentence} \mid \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \end{array}$$

Above, the BNF "$\rightarrow$" roughly means "is a" and the BNF "|" means "or", so that the first syntax rule (or **production**) means "A sentence is either an atomic sentence or a complex sentence."

The set of sentences that represent our knowledge is called our **knowledge base.** For example, the knowledge base for our liars and truth-tellers problem is:

$$\{A \Leftrightarrow \neg B,\ B \Leftrightarrow \neg C,\ C \Leftrightarrow \neg A \wedge \neg B\}$$

Note that these sentences could all be combined with conjunction in a single sentence which expresses all knowledge of the knowledge base:

$$(A \Leftrightarrow \neg B) \wedge (B \Leftrightarrow \neg C) \wedge (C \Leftrightarrow \neg A \wedge \neg B)$$

If we think of our atomic sentence symbols as boolean variables, then an assignment to all variables would be called a **truth assignment**. A truth assignment which makes a sentence s true is said to **satisfy** s. This satisfying truth assignment is also called a **model** of s. Another way to think about it is that each truth assignment is a "world", and that the models of s are "possible worlds". If and only if s has a model, s is said to be **satisfiable**. If and only if s has no model, s is said to be **unsatisfiable**. If and only if every truth assignment is a model of s, then s is said to be **valid** or a **tautology**.

Now let us consider two sentences s_1 and s_2. We say that s_1 **entails** s_2, denoted $s_1 \models s_2$, if and only if every model of s_1 is also a model of s_2. Sometimes this is expressed by saying that "s_2 logically follows from s_1". Two sentences s_1 and s_2 are **logically equivalent** if and only if $s_1 \models s_2$ and $s_2 \models s_1$.

3 Conjunctive Normal Form (CNF)

Propositional knowledge which is expressed as a *conjunction of disjunctions of literals* is said to be in **conjunctive normal form** (CNF). Recall that a literal is an atomic sentence or a negated atomic sentence. A disjunction (or) of literals is often referred to as a **clause**. Conjunctive normal form is a conjunction (and) of such clauses.

Any sentence can be converted to CNF. We will describe the steps of this process and show how the liar and truth-teller example knowledge base may be converted to CNF.

1. **Eliminate $\Leftrightarrow$.** Replace each occurrence of $s_1 \Leftrightarrow s_2$ with the equivalent $(s_1 \Rightarrow s_2) \wedge (s_2 \Rightarrow s_1)$. Thus, the knowledge base

$$\{A \Leftrightarrow \neg B,\ B \Leftrightarrow \neg C,\ C \Leftrightarrow \neg A \wedge \neg B\}$$

may be rewritten as

$$\{(A \Rightarrow \neg B) \wedge (\neg B \Rightarrow A),\ (B \Rightarrow \neg C) \wedge (\neg C \Rightarrow B),\ (C \Rightarrow \neg A \wedge \neg B) \wedge (\neg A \wedge \neg B \Rightarrow C)\}$$

We noted before that the set of knowledge base sentences could all be combined with conjunction in a single equivalent sentence. At any time, we can use this equivalence in the opposite direction to separate conjunctions into separate sentences of the knowledge base:

$$\{A \Rightarrow \neg B,\ \neg B \Rightarrow A,\ B \Rightarrow \neg C,\ \neg C \Rightarrow B,\ C \Rightarrow \neg A \wedge \neg B,\ \neg A \wedge \neg B \Rightarrow C\}$$

2. **Eliminate $\Rightarrow$.** Replace each occurrence of $s_1 \Rightarrow s_2$ with the equivalent $\neg s_1 \vee s_2$. With such replacements, our knowledge base becomes:

$$\{\neg A \vee \neg B,\ \neg\neg B \vee A,\ \neg B \vee \neg C,\ \neg\neg C \vee B,\ \neg C \vee \neg A \wedge \neg B,\ \neg(\neg A \wedge \neg B) \vee C\}$$

3. **Move $\neg$ inward.** In CNF, negation ($\neg$) only occurs in literals before atomic sentence symbols. All occurrences of $\neg$ can either be negated or "moved" inward towards the atomic sentences using three equivalences:

$$\begin{array}{rl} \text{de Morgan's law} & \neg(s_1 \wedge s_2) \equiv \neg s_1 \vee \neg s_2 \\ & \neg(s_1 \vee s_2) \equiv \neg s_1 \wedge \neg s_2 \\ \text{double } \neg \text{ elimination} & \neg\neg s \equiv s \end{array}$$

Using these equivalences, our knowledge base is rewritten:

$$\{\neg A \vee \neg B,\ B \vee A,\ \neg B \vee \neg C,\ C \vee B,\ \neg C \vee \neg A \wedge \neg B,\ A \vee B \vee C\}$$

The last sentence was rewritten in multiple steps:

$$\neg(\neg A \wedge \neg B) \vee C \equiv (\neg\neg A \vee \neg\neg B) \vee C \equiv (A \vee \neg\neg B) \vee C \equiv (A \vee B) \vee C \equiv A \vee B \vee C$$

4. **Distribute $\vee$ over $\wedge$.** That is, move disjunction ($\vee$) inward while moving conjunction ($\wedge$) outward to form a conjunction of disjunctions. This is accomplished through distributivity of $\vee$:

$$\begin{array}{rcl} s_1 \vee (s_2 \wedge s_3) & \equiv & (s_1 \vee s_2) \wedge (s_1 \vee s_3) \\ (s_1 \wedge s_2) \vee s_3 & \equiv & (s_1 \vee s_3) \wedge (s_2 \vee s_3) \end{array}$$

Thus we can rewrite our knowledge base:

$$\{\neg A \vee \neg B,\ B \vee A,\ \neg B \vee \neg C,\ C \vee B,\ (\neg C \vee \neg A) \wedge (\neg C \vee \neg B),\ A \vee B \vee C\}$$

or

$$\{\neg A \vee \neg B,\ B \vee A,\ \neg B \vee \neg C,\ C \vee B,\ \neg C \vee \neg A,\ \neg C \vee \neg B,\ A \vee B \vee C\}$$

This is sometimes more compactly expressed as a set of a set of literals, where the sets of literals are implicitly understood as clauses (disjunctions) and the set of these is implicitly understood as the knowledge base, a conjunction of known sentences:

$$\{\{\neg A, \neg B\},\ \{B, A\},\ \{\neg B, \neg C\},\ \{C, B\},\ \{\neg C, \neg A\},\ \{\neg C, \neg B\},\ \{A, B, C\}\}$$

It is important to note that a model in CNF is a truth assignment that makes *at least one literal in each clause true.*

4 Resolution Theorem Proving

Consider the two clauses $\{B, A\}$ and $\{\neg B, \neg C\}$ of our example CNF knowledge base. The first reads, "Bob is a truth-teller or Amy is a truth-teller." The second reads, "Bob is not a truth-teller or Cal is not a truth-teller." Consider what happens according to Bob's truthfulness:

- **Bob is not a truth-teller.** The second clause is satisfied. The first clause is satisfied if and only if Amy is a truth-teller.
- **Bob is a truth-teller.** The first clause is satisfied. The second clause is satisfied if and only if Cal is not a truth-teller.

Since one case or the other holds, we know in any model of *both* clauses that Amy is a truth-teller or Cal is not a truth-teller. In other words, from clauses $\{B, A\}$ and $\{\neg B, \neg C\}$, we can **derive** the clause $\{A, \neg C\}$ and add it to our knowledge base. This is a specific application of the **resolution rule**.

In general, we can express the resolution rule as follows: Given a clause $\{l_1, l_2, \ldots, l_i, A\}$ and $\{\neg A, l_{i+1}, l_{i+2}, \ldots, l_n\}$, we can derive the clause $\{l_1, l_2, ..., l_n\}$. If A is false, one of the other first clause literals must be true. If A is true, one of the other second clause literals must be true. Since A must be either true or false, then at least one of all the other n literals must be true.

Not all possible resolution rule derivations are useful. Consider what happens when we apply the resolution rule to the first two clauses of our example knowledge base. From $\{\neg A, \neg B\}$ and $\{B, A\}$, we can derive either $\{\neg A, A\}$ or $\{\neg B, B\}$ depending on which atomic sentence we use for the resolution. In either case, we derive a tautology. The clause $\{\neg A, A\}$ is always true. It reads "Amy is not a truth-teller or Amy is a truth-teller."

In order to direct our derivations towards a simple goal, we will perform **proof by contradiction**[2]. The basis of this style of proof is the fact that $s_1 \models s_2$ *if and only if* $(s_1 \wedge \neg s_2)$ *is unsatisfiable.* Put another way, every truth assignment that makes s_1 true also makes s_2 true if and only if there exists no truth assignment where s_1 is true and s_2 is false.

The application of this principle is simple. Suppose that sentence s_1 represents our knowledge base, and that we wish to wish to prove that sentence s_2 follows from s_1. We simply add $\neg s_2$ to our knowledge base and seek to derive a contradiction. We take what we believe must follow, and prove it by showing that the opposite causes a contradiction. This approach is not only effective in human argument, but is important for automated theorem proving as well.

Let us use this approach to prove that Cal is a liar ($\neg C$). In addition to the knowledge base, we assume the negation of what we wish to prove ($\neg\neg C$, that is C). We will number our clauses to the left. To the right, we will list the clause numbers used to derive the clause by resolution.

(1)	$\{\neg A, \neg B\}$		Knowledge base
(2)	$\{B, A\}$		
(3)	$\{\neg B, \neg C\}$		
(4)	$\{C, B\}$		
(5)	$\{\neg C, \neg A\}$		
(6)	$\{\neg C, \neg B\}$		
(7)	$\{A, B, C\}$		
(8)	$\{C\}$		Assumed negation
(9)	$\{\neg A\}$	(5),(8)	Derived clauses
(10)	$\{B\}$	(2),(9)	
(11)	$\{\neg C\}$	(3),(10)	
(12)	$\{\}$	(8),(11)	*Contradiction!*

[2]a.k.a. reductio ad absurdum

Recall that at least one literal of each clause must be true for a truth assignment to be a model. This last **empty clause** has no literals at all and represents a clear contradiction. To see why, consider the resolution that led to it. In (10) and (11) we declare a fact and its negation to be true. The resolution rule leaves us no literals to be made true to satisfy these clauses. A clause with no literals is thus logically equivalent to *false*. Since we hold each clause to be true, we have in effect shown that false is true. This is a contradiction. Thus it cannot be the case that $\{C\}$ is true as we had assumed. Thus $\{\neg C\}$ logically follows from our knowledge base.

It should be noted that an unsatisfiable knowledge base will derive an empty clause without any need of assumption. One can thus prove any sentence using proof by contradiction and starting with a contradictory knowledge base.

If one cannot derive a contradiction in this manner, then there is a satisfying truth assignment for the knowledge base and the assumed negation of the hypothesis one sought to prove.

For practice, the reader should perform two resolution proofs by contradiction that

- Amy is a liar, and
- Bob is a truth-teller.

When we derive a sentence s_2 from sentence s_1, we denote it $s_1 \vdash s_2$. A proof procedure that derives only what is entailed is called **sound**. A proof procedure that can derive anything that is entailed is called **complete**. Resolution theorem proving is both sound and complete.

5 pycryptosat

Resolution theorem proving seeks to answer the question "Is the knowledge base plus the negated hypothesis satisfiable?" There is an intense research effort underway for high-performance satisfiability solvers. Each year, there is a satisfiability software competition. We'll be using the Python bindings for cryptominisat, which is one of the many forks of the MiniSAT solver.

Here's how it works. We start off by positive integers to each of our literals, such as:

```
1: Amy is a truth-teller.
2: Bob is a truth-teller.
3: Cal is a truth-teller.
```

If we then want to test the satisfiability of this knowledge base, we can do so as follows:

```
from pycryptosat import Solver
s = Solver()
s.add_clause([1])
s.add_clause([-2])
s.add_clause([-1, 2, 3])
sat, solution = s.solve()
```

In the above, the variable `sat` contains True if the knowledge base represented by the added clauses is satisfiable, and it contains False otherwise. This can be abbreviated as

```
from pycryptosat import Solver
s = Solver()
s.add_clauses([[1],[-2],[-1, 2, 3]])
sat, solution = s.solve()
```

You can also temporarily test an additional literal without adding it the solver object persistently, by adding it as an argument to `solve`, like this:

```
from pycryptosat import Solver
s = Solver()
s.add_clauses([[1],[-2],[-1, 2, 3]])
sat, solution = s.solve([3])
```

More details can be found on the Cryptominisat webpage[3].

6 SATsolver

In order to make using this library even easier for the purpose at hand, we're providing a thin wrapper around `pycryptosat` call `SATSolver`, which is described below.

SATSolver provides two functions for invoking solving:

```
  testKb(clauses)
  testLiteral(literal,clauses)
```

`testKb` is used for testing to see if a particular knowledge base is satisfiable. It creates a pycryptosat Solver object, adds the clauses, and calls the `solve` method. The result returned will be `True` if the combined clauses are satisfiable, and `False` otherwise.

Often, the user will simply want to know at a given point if a literal can be proved true/false or not relative to a previously constructed knowledge base. For this, we use the the `testLiteral` function can be used. It returns one of the symbols `True`, `False`, and `None`, which represent whether a particular literal is true, false, or unprovably neither relative to the knowledge base.

Here we provide a test case in which we take our liar and truth-teller example and apply the `SATSolver` code.

```
  clauses = [[-1,-2],[2 ,1],[-2,-3],[3,2],[-3,-1],[-3, -2],[1,2,3]]
  print('Knowledge base is satisfiable:',testKb(clauses))
  print('Is Cal a truth-teller?',end=' ')
  result = testLiteral(3,clauses)
  if result==True:
      print('Yes.')
  elif result==False:
      print('No.')
  else:
      print('Unknown.')
```

This code may be used to efficiently solve the exercises of the next section.

[3]See URL `https://github.com/msoos/cryptominisat`.

7 Exercises

For each of the following problems (which you may or may not be assigned to do), there are four exercises:

(a) Express all relevant problem facts as a propositional logic knowledge base. Clearly explain the meaning of your propositional symbols.

(b) Convert the propositional logic knowledge base to CNF.

(c) Use resolution theorem proving to solve the problem.

(d) Solve the problem computationally with a SAT solver.

Exercise (a) should precede (b) which should in turn precede (c) and (d).

Problems:

1. **Horn Clauses:** (from Russell and Norvig, Exercise 7.9) 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?

2. **Liars and Truth-tellers 1:** (adapted from OSSMB[4] 82-12) 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.
   - 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?

3. **Liars and Truth-tellers 2:** (adapted from OSSMB 83-11) 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.
   - 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?

4. **Robbery and a Salt:** (from OMG[5] 22.1.1) The salt has been stolen! Well, it was found that the culprit was either the Caterpillar, Bill the Lizard or the Cheshire Cat. The three were tried and made the following statements in court: CATERPILLAR: Bill the Lizard ate the salt. BILL THE LIZARD: That is true! CHESHIRE CAT: I never ate the salt.

   As it happened, at least one of them lied and at least one told the truth. Who ate the salt?

[4]Ontario Secondary School Mathematics Bulletin
[5]Ontario Mathematical Gazette

5. **Multiple Choice:** (adapted from CRUX[6] 357) In a certain multiple-choice test, one of the questions was illegible, but the choice of answers was clearly printed.

   (a) All of the below.
   (b) None of the below.
   (c) All of the above.
   (d) One of the above.
   (e) None of the above.
   (f) None of the above.

   Assuming that more than one answer may be true, divide answers into three groups:

   - those that are necessarily true,
   - those that are necessarily false, and
   - those that could be true or false.

   How many different possible ways are there to consistently circle true subset(s) of answers? List the subset(s).

6. **A Sanctum for the Good En-doors-ment:** (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[7] are knights, so is A.

   Smullyan's problem: Which door should the philosopher choose?

   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.

[6] Crux Mathematicorum
[7] Here, "I" is H referring to himself.

7. **An Honest Name:** (from FUNCT[8] 1.2.4) Three golfers named Tom, Dick, and Harry are walking to the clubhouse. The first man in line says, "The guy in the middle is Harry." The man in the middle says, "I'm Dick." The last man says, "The guy in the middle is Tom." Tom, the best golfer of the three, always tells the truth. Dick sometimes tells the truth, while Harry, the worst golfer, never does. Figure out who is who.

8. **Liars and Truth-tellers 3:** (adapted from JRM[9] 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?

8 Reasoning about Clue

In this section, we describe the skeleton implementation of Python `clueReasoner` code which maintains a knowledge base of Clue game information, and performs deductive reasoning based on `SATSolver`. Portions of the implementation which supply clauses to `SATSolver` are deliberately left unimplemented as exercises for the reader. When these portions are implemented, the `clueReasoner` is fully functional and will allow the user to make expert deductions about the game.

We begin by describing the module level variables of `clueReasoner`:

```
caseFile = "cf"
players = ["sc", "mu", "wh", "gr", "pe", "pl"]
extendedPlayers = players + [caseFile]
suspects = ["mu", "pl", "gr", "pe", "sc", "wh"]
weapons = ["kn", "ca", "re", "ro", "pi", "wr"]
rooms = ["ha", "lo", "di", "ki", "ba", "co", "bi", "li", "st"]
cards = suspects + weapons + rooms
```

[8] Function Journal
[9] Journal of Recreational Mathematics

`caseFile` the symbol representing the case file

`players` a list of symbols representing the players in their turn order

`extendedPlayers` a list of symbols representing the players with the case file added to it

`suspects` a list of symbols representing the suspects

`weapons` a list of symbols representing the weapons

`rooms` a list of symbols representing the rooms

`cards` a list of symbols representing all suspects, weapons, and rooms

It is important to note the separate roles of the `players` and `suspects` lists. While players do take on the identities of suspects, the game can be played with 3 to 6 players, thus the `players` list will contain a subset of the `suspects` list. Also, the `player` list is given in *turn order.*

The symbols used to represent cards and players are, for the most part, the first two letters of their names, as shown in the following table:

Col. Mustard	`mu`	Knife	`kn`	Hall	`ha`
Prof. Plum	`pl`	Candlestick	`ca`	Lounge	`lo`
Mr. Green	`gr`	Revolver	`re`	Dining Room	`di`
Mrs. Peacock	`pe`	Rope	`ro`	Kitchen	`ki`
Miss Scarlet	`sc`	Lead Pipe	`pi`	Ballroom	`ba`
Mrs. White	`wh`	Wrench	`wr`	Conservatory	`co`
				Billiard Room	`bi`
Case File	`cf`			Library	`li`
				Study	`st`

The next functions use the indices (position numbers) of these symbols in their respective lists to compute unique literal integers for `SATSolver` clauses. We mentioned earlier that each Clue atomic sentence c_p symbolizes the statement "The card c is in place p.". There is an atomic sentence for each place and card pair. We assign an integer to each sentence for our SATSolver as follows. Suppose we have a place index i_p and a card index i_c. Then the integer corresponding to c_p is $i_p \times \texttt{numCards} + i_c + 1$.

For example, consider the atomic sentence pi_{wh}. The player Mrs. White ("wh") has index 2. The lead pipe card ("pi") has index 10. There are 21 cards. Therefore, the integer corresponding to the atomic sentence c_p is $2 \times 21 + 10 + 1 = 53$. Each atomic sentence has a unique number from 1 through $(\texttt{numPlayers} + 1) \times \texttt{numCards}$.

There are two version of the `getPairNum` functions. The first takes symbols, and the second takes integer indices.

⟨*Get the literal integer for the player-card pair*⟩≡
```
  def getPairNumFromNames(player,card):
      return getPairNumFromPositions(extendedPlayers.index(player),
                                     cards.index(card))

  def getPairNumFromPositions(player,card):
      return player*len(cards) + card + 1
```

Now we come to the portions of the `clueReasoner` that are the core of the project. Here you must consider the knowledge to be represented, and construct a list of suitable clauses for the SAT-solver.

In `initialClauses`, you add general knowledge about the game which is known by all players before the game starts.

The knowledge you will encode falls into four categories:

1. **Each card is in at least one place.** That is, each card is in a player's hand or the case file. We do not allow the possibility that a card was accidentally left in the box[10].

2. **If a card is in one place, it cannot be in another place.** Since there is only one of each card, the fact that it is in one place means in cannot be in another place. For each pair of places, it cannot be in both.

3. **At least one card of each category is in the case file.** We choose one card at random from each category to be placed in the case file. For example, at least one of the weapon cards is in the case file.

4. **No two cards in each category are in the case file.** Since only one card from each category is placed in the case file, we cannot, for example, have both the Pipe `pi` and Wrench `wr` in the case file. For each pair of cards within a category, at least one of the two must not be in the case file.

First, the programmer should seek to represent some of these facts on paper in propositional logic and convert them to CNF in order to gain a good understanding of what clauses need to be constructed. Then, the `initialClauses` function should be coded so that it returns a list of these clauses.

⟨*Add initial clauses*⟩≡
```
# TO BE IMPLEMENTED AS AN EXERCISE
def initialClauses():
    clauses = []

    # Each card is in at least one place (including case file).
    for c in cards:
        clauses.append([getPairNumFromNames(p,c) for p in extendedPlayers])

    # A card cannot be in two places.

    # At least one card of each category is in the case file.

    # No two cards in each category can both be in the case file.

    return clauses
```

[10]This sadly happens on occasion.

At the beginning of the game, the player is dealt a hand (set) of cards. This is private information the player has about what is *not* in the case file. Before the beginning of the game, we use the `hand` function to note that the given cards are in the possession of that player.

Here the programmer should add the simple clauses representing the known possession of these cards.

⟨*Return list of clauses indicating player information*⟩≡

```
# TO BE IMPLEMENTED AS AN EXERCISE
def hand(player,cards):
    return []
```

Earlier, we noted that the turn order of the `players` list is important. The `suggest` function is where this information is used. In the game of Clue, a player may suggest a suspect, a weapon, and a room, upon entering that room on the board. Since the board movement is irrelevant to the deduction, we will not detail the circumstances of a suggestion here.

When a suggestion of three cards is made, the player to the left of the suggester (clockwise) checks their private cards to see if any of the cards are part of the suggestion. If so, the player must *refute* the suggestion by privately showing one of these refuting cards to the suggester. If not, the player states that they cannot refute the suggestion, and attention then turns to the next player clockwise. The next player does the same, either refuting or not, and this continues clockwise until the first possible refutation, or until all other players have stated that they cannot refute the suggestion.

Suppose all six players are playing with the following turn order: Scarlet, Mustard, White, Green, Peacock, Plum. The meaning of the function call `suggest("sc", "wh", "pi", "ha", "pe", "ha")` is as follows: `suggester` Miss Scarlet suggested that Mrs. White committed the murder with the lead pipe in the hall (parameters `card1`, `card2`, and `card3`). This was refuted by `refuter` Mrs. Peacock, who showed Scarlet the hall card (`cardShown`). Consider that we learn much more than the fact that Mrs. Peacock has the hall card.

After Scarlet suggested $\{\text{wh}, \text{pi}, \text{ha}\}$, all players between Scarlet and the refuter Peacock must have been unable to refute the suggestion. Thus we *also* learn that these players, Mustard, White, and Green, do not have any of these cards. The knowledge we gain from a suggestion is both negative, which players do not have the suggested cards, and positive, which card a player has.

In many cases, a player does not get to see the card that is shown to refute the suggestion. In this case, the `cardShown` parameter is given the value `None`. Consider the function call `suggest("wh", "mu", "re", "ba", "pe", None)`. White suggests that Mustard committed the murder with the revolver in the ballroom. Peacock refutes the suggestion by showing a card privately to White, but the player (Scarlet) does not get to see this card. In this case we learn that Green does not have these cards. We also learn that Peacock has *at least one of these cards*. The information gained is similar to the previous case. The difference is that at least one of the three suggested cards must be in the refuter's hand.

In some cases, there may not be a refuter at all. This is indicated by having the `refuter` parameter with the value `None`. (The `cardShown` parameter should then also be `None`.) If no player can refute the suggestion, we simply gain information that all players except the suggester do not have the suggested cards. The suggester is never part of the refutation.

Implementing `suggest`, the programmer must bear all of these cases in mind. Detailed rules to Clue may be found at the Hasbro website[11].

⟨*Return clauses with suggestion information*⟩≡
```
# TO BE IMPLEMENTED AS AN EXERCISE
def suggest(suggester,card1,card2,card3,refuter,cardShown):
    return []
```

[11] See URL `https://tinyurl.com/y7d9ule5`.

The winner of clue is the first player to correctly make an *accusation* (not a suggestion) naming the three cards in the case file. Each player can make an accusation on any one turn, and checks the correctness of the declared accusation by privately looking in the case file. An accusation can only be made once, as the player either wins or loses depending on the correctness of the accusation. (If a player loses through incorrect accusation, the player continues to refute suggestions by other players, but may no longer take part in play otherwise.)

Consider the example function call `accuse("sc, "pe", "pi", "bi", True)`. This represents the fact that the `accuser` Scarlet accused Mrs. Peacock of committing the murder with the pipe in the billiard room, and that this accusation is correct (parameter `isCorrect` is `True`). From a correct accusation, we gain knowledge that these three cards (parameters `card1`, `card2`, and `card3`) are in the case file.

Now consider what happens when the accusation is not correct. Suppose we have the function call `accuse("sc", "pe", "pi", "li", False)`. Then we learn that at least one of the three cards of the accusation is *not* in the case file. In implementing `accuse`, the programmer must handle both of these cases.

Note futher that whether the accusation is correct or not, one can deduce that the accuser does not have in hand any of the cards involved in the accusation. A player might `suggest` cards that are in hand in order to throw others off the trail, but a false `accuse` results in losing the game.

⟨*Return accusation information*⟩≡
```
# TO BE IMPLEMENTED AS AN EXERCISE
def accuse(accuser,card1,card2,card3,isCorrect):
    return []
```

The rest of the functions in our skeleton implementation are to aid the programmer in querying SATSolver and viewing the deductions that can be made at any given point. The `query` function takes a `player` (possibly `caseFile`) and `card` and tests that literal, returning `SATSolver`'s `True`, `False`, and `None` depending on the results of satisfiability testing.

⟨*Query whether a literal is true, false, or unknown*⟩≡
```
def query(player,card,clauses):
    return SATSolver.testLiteral(getPairNumFromNames(player,card),clauses)
```

These return codes may be converted to simple strings using `queryString`.

⟨*Convert a query return code to a string*⟩≡
```
def queryString(returnCode):
    if returnCode == True:
        return 'Y'
    elif returnCode == False:
        return 'N'
    else:
        return '-'
```

These two functions in turn are used in `printNotepad` to print out a table (or “detective notepad”) indicating the current state of the propositional knowledge about the locations of cards.

⟨*Print a table of current deductions*⟩≡

```
def printNotepad(clauses):
    for player in players:
        print '\t', player,
    print '\t', caseFile
    for card in cards:
        print card,'\t',
        for player in players:
            print queryString(query(player,card,clauses)),'\t',
        print queryString(query(caseFile,card,clauses))
```

Finally, we supply a `playClue` function with information from an actual Clue game which provides usage examples and a test of the implementation’s correctness. When `initialClauses`, `hand`, `suggest`, and `accuse` are all implemented properly, `printNotepad` should indicate the correct place of each card in this particular game both times it runs.

⟨*Test clueReasoner with a sample game*⟩≡

```
def playClue():
    clauses = initialClauses()
    clauses.extend(hand("sc",["wh", "li", "st"]))
    clauses.extend(suggest("sc", "sc", "ro", "lo", "mu", "sc"))
    clauses.extend(suggest("mu", "pe", "pi", "di", "pe", None))
    clauses.extend(suggest("wh", "mu", "re", "ba", "pe", None))
    clauses.extend(suggest("gr", "wh", "kn", "ba", "pl", None))
    clauses.extend(suggest("pe", "gr", "ca", "di", "wh", None))
    clauses.extend(suggest("pl", "wh", "wr", "st", "sc", "wh"))
    clauses.extend(suggest("sc", "pl", "ro", "co", "mu", "pl"))
    clauses.extend(suggest("mu", "pe", "ro", "ba", "wh", None))
    clauses.extend(suggest("wh", "mu", "ca", "st", "gr", None))
    clauses.extend(suggest("gr", "pe", "kn", "di", "pe", None))
    clauses.extend(suggest("pe", "mu", "pi", "di", "pl", None))
    clauses.extend(suggest("pl", "gr", "kn", "co", "wh", None))
    clauses.extend(suggest("sc", "pe", "kn", "lo", "mu", "lo"))
    clauses.extend(suggest("mu", "pe", "kn", "di", "wh", None))
    clauses.extend(suggest("wh", "pe", "wr", "ha", "gr", None))
    clauses.extend(suggest("gr", "wh", "pi", "co", "pl", None))
    clauses.extend(suggest("pe", "sc", "pi", "ha", "mu", None))
    clauses.extend(suggest("pl", "pe", "pi", "ba", None, None))
    clauses.extend(suggest("sc", "wh", "pi", "ha", "pe", "ha"))
    clauses.extend(suggest("wh", "pe", "pi", "ha", "pe", None))
    clauses.extend(suggest("pe", "pe", "pi", "ha", None, None))
    clauses.extend(suggest("sc", "gr", "pi", "st", "wh", "gr"))
    clauses.extend(suggest("mu", "pe", "pi", "ba", "pl", None))
    clauses.extend(suggest("wh", "pe", "pi", "st", "sc", "st"))
    clauses.extend(suggest("gr", "wh", "pi", "st", "sc", "wh"))
    clauses.extend(suggest("pe", "wh", "pi", "st", "sc", "wh"))
    clauses.extend(suggest("pl", "pe", "pi", "ki", "gr", None))
    print 'Before accusation: should show a single solution.'
    printNotepad(clauses)
    print
    clauses.extend(accuse("sc", "pe", "pi", "bi", True))
    print 'After accusation: if consistent, output should remain unchanged.'
    printNotepad(clauses)
```

9 Improving Performance

In this section, we first discuss simple means for improving reasoning performance by changing the knowledge base. We then describe three different means of adding to the knowledge base: deductive learning, inductive learning, and knowledge acquisition.

9.1 Deleting Clauses

Until now, we have made no effort (programmatically or on paper) to delete unnecessary clauses from the knowledge base. Reasoning engines often have a preprocessing stage that streamlines the knowledge base for greater reasoning efficiency. Here, we describe three simple types of unnecessary clauses: equivalent clauses, subsumed clauses, and tautology clauses[12].

Recall our knowledge base from the liar/truth-teller example problem:

(1)	$\{\neg A, \neg B\}$	Knowledge base
(2)	$\{B, A\}$	
(3)	$\{\neg B, \neg C\}$	
(4)	$\{C, B\}$	
(5)	$\{\neg C, \neg A\}$	
(6)	$\{\neg C, \neg B\}$	
(7)	$\{A, B, C\}$	

Did you notice that clauses (3) and (6) are logically equivalent? Each clause represents a disjunction ("or") of the literals, and by the commutative law for disjunction (e.g. $A \vee B \equiv B \vee A$), the ordering of literals within a clause is irrelevant. If one knows "It will rain or snow," this is the same as knowing "It will snow or rain." Thus, clause (6) adds nothing to the knowledge of clause (3), and can be safely deleted from the knowledge base. Indeed, any time that resolution would produce a clause which is a permutation of literals in a previous clause, we should not add it to the knowledge base. These are equivalent clauses.

Did you also notice that clause (2) entails clause (7)? Every model of $B \vee A$ is also a model of $B \vee A \vee \ldots$. Adding disjuncts to a clause has the effect of increasing the number of models for the clause. If one knows "It will rain or snow," this is stronger, more specific knowledge than "It will rain, snow, or shine." If one knows the former, one knows the latter. However, if one knows the latter, one does not know the former. We say that clause (7) is *subsumed* by clause (2). Clause c_1 subsumes clause c_2 if the literals of c_1 are a *subset* of the literals of c_2. A subsumed clause can be deleted from the knowledge base without loss of knowledge. Further, when resolution would produce a clause which is subsumed by a previous clause, we should not add it to the knowledge base.

Finally, we note that tautologies should be deleted from the knowledge base. As we noted before, the clause $\{A, \neg A\}$, or any clause with both positive and negative literals for the same atomic sentence, is always true for all truth assignments, and adds nothing to our knowledge. Knowing "It will rain or it won't rain," is useless. We can safely delete all tautology clauses from the knowledge bases.

9.2 Adding Clauses

People often urge each other not to "reinvent the wheel", to perform work that has already been done. This is the principle behind *dynamic programming algorithms.* (See *Solving the Dice Game Pig: an introduction to dynamic programming and value iteration*[13].) In computation, there are often

[12]The interested reader can also investigate the use of "pure clause elimination" (a.k.a. the "pure symbol heuristic" [**?**, §7.6]).

[13]See URL `http://cs.gettysburg.edu/%7Etneller/nsf/pig/index.html`.

important tradeoffs between computational time and memory. Dynamic programming stores results of computations to speed future computations dependent on such results.

In the context of automated reasoning, one can store new knowledge produced through reasoning in order to potentially speed future reasoning that depends on such knowledge. Let us consider our previous liar/truth-teller example with unnecessary clauses eliminated. Suppose we perform resolution to prove that Amy is a liar:

(1)	$\{\neg A, \neg B\}$		Knowledge base
(2)	$\{B, A\}$		
(3)	$\{\neg B, \neg C\}$		
(4)	$\{C, B\}$		
(5)	$\{\neg C, \neg A\}$		
(6)	$\{A\}$		Assumed negation
(7)	$\{\neg B\}$	(1),(6)	Derived clauses
(8)	$\{C\}$	(4),(7)	
(11)	$\{\neg A\}$	(5),(8)	
(9)	$\{\}$	(6),(11)	*Contradiction!*

From this proof by contradiction we now know $\{\neg A\}$, and *can add it to our knowledge base.* Why might we want to do this? First, this would allow us to eliminate all subsumed clauses containing $\neg A$, resulting in a more compact knowledge base with equivalent knowledge and fewer possible resolutions. This in turn is appealing, as a proof is essentially a search through a maze of possible resolutions to a contradiction, and such compaction can simplify search.

Consider what happens when we add $\{\neg A\}$, delete subsumed clauses eliminate subsumed clauses $\{\neg A, \neg B\}$ and $\{\neg C, \neg A\}$, and prove that Bob is a truth-teller:

(1)	$\{B, A\}$		Knowledge base
(2)	$\{\neg B, \neg C\}$		
(3)	$\{C, B\}$		
(4)	$\{\neg A\}$		
(5)	$\{\neg B\}$		Assumed negation
(6)	$\{A\}$	(1),(5)	Derived clauses
(7)	$\{\}$	(4),(6)	*Contradiction!*

In the previous proof that Amy was a liar, there were eight possible clause pairings for the first resolution. Of these, two derive tautologies ((1,2) and (3,4)), and two derive desirable *unit clauses*[14] ((1,6) and (5,6)). In this proof that Bob is a truth-teller, there are now only five possible pairings for the first resolution. Of these, only one derives a tautology ((2,3)), and three derive desirable unit clauses ((1,4), (1,5), and (3,5)). There are various algorithmic strategies in searching for a resolution proof. However, even a randomized strategy will find a contradiction more easily with this compacted knowledge base.

With the new knowledge that Bob is a truth-teller, we can again remove subsumed clauses $\{B, A\}$ and $\{C, B\}$, and prove that Cal is a liar:

(1)	$\{\neg B, \neg C\}$		Knowledge base
(2)	$\{\neg A\}$		
(3)	$\{B\}$		
(4)	$\{C\}$		Assumed negation
(5)	$\{\neg B\}$	(1),(4)	Derived clauses
(6)	$\{\}$	(3),(5)	*Contradiction!*

[14]A unit clause is a clause of one literal.

This time, there were only two possible pairings for the first resolution ((1,3) and (1,4)), and both derive unit clauses. New knowledge that Cal is a liar subsumes clause (1), so our knowledge base is now:

(1)	$\{\neg A\}$	Knowledge base
(2)	$\{B\}$	
(3)	$\{\neg C\}$	

It is interesting to note that this knowledge base, the answer to the example exercise, subsumes all knowledge of the exercise. The exercise solution is the simplest expression of the exercise knowledge.

Should one always add the knowledge obtained through proof by contradiction? Will this always benefit us? No. One can prove tautologies (e.g. $A \vee \neg A$) or subsumed facts (e.g. $\neg A \vee B \vee \neg C$), and the addition of these to our knowledge base has no benefit.

Furthermore, consider that the number of possible (non-tautology) clauses of length l for a atomic sentences is $2^l \binom{a}{l}$:

a	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
1	2	0	0	0	0
2	4	4	0	0	0
3	6	12	8	0	0
4	8	24	32	16	0
5	10	40	80	80	32
10	20	180	960	3360	8064
20	40	760	9120	77520	496128
40	80	3120	79040	1462240	21056256
80	160	12640	657260	25305280	769280512

There are two important things to observe in this table. Not surprisingly, the first is that there are too many possible clauses to store for long clauses with large numbers of atomic sentences. Although subsumption may make the storage of a large fraction of these unnecessary, still one can easily store enough long-clause knowledge to be *detrimental* to learning. It is counterproductive if one adds many proven long clauses only to increase the complexity of the "maze" of possible paths to a resolution proof, slowing future proof searches.

The other important observation is that unit clauses are rare and precious. If one can prove them, add them, for there will be at most $2a$ such clauses. These have the greatest power to subsume other clauses, and have the greatest potential to benefit future resolution theorem proving.

As a general rule of thumb, prefer to add shorter clauses to your knowledge base that are relevant to your expected query distribution.

9.3 Deduction, Induction, and Acquisition

In his book *Machine Learning* [?], Tom Mitchell writes "A computer is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." In the context of our Clue project, what are T, P, and E? There are many possible choices, but we will examine one choice, and consider different ways that a reasoning program could "learn".

In the previous section, we saw that adding proven facts to the knowledge base can benefit future proof computation. This suggests that our task T is the task of reasoning, answering a query to the knowledge base. One natural performance measure P might be the expected average computational time over a given distribution of queries (e.g. single literal queries). What then is the experience E? In this case, E is the experience of deducing new knowledge and adding it to the knowledge base. This type of machine learning is called *deductive learning.*

There is a very different type of learning called *inductive learning*, or *learning by example*. Uncertain general knowledge of the world is induced from specific knowledge. Let us momentarily suppose that the rules of the game are not supplied to the reasoning system. Even the simple fact that each card is unique and is in exactly one location is not known. At the end of the game, one usually has seen or can deduce the locations of many cards, yet the locations of some may remain forever unknown. However, inductive learning might be used to take a substantial amount of observed or deduced card position data, and generalize the fact that a card is in exactly one location.

As a trivial explanatory example, suppose one has two upside down cups which may or may not have a ball underneath. For each trial, the cups are set up unobserved by another. Then one flips two coins, one for each cup. For each head flipped, one lifts and looks under the corresponding cup. For one-quarter of trials, nothing is observed. For one-half of trials, one looks underneath one cup. For the remaining one-quarter of trials, one looks underneath both cups.

Now suppose that when both cups are lifted, you always see exactly one ball under one cup or the other. Further suppose that when a single cup is lifted, you observe a single ball some of the time. Over time, such observations would give you greater and greater confidence that, regardless of coin tossed, there is exactly one ball underneath the cups each time.

Let atomic sentences B_1 and B_2 denote that there is a ball underneath cup 1 and cup 2 respectively. After a number of observations, one might *induce* the (uncertain) knowledge $B_1 \Leftrightarrow \neg B_2$. With more observations, this induced knowledge becomes more certain, but it can never be completely certain. For example, the person setting up the cups may be following a rule that exactly one ball will be placed under a random cup until the ball supply runs out (using a new ball each time), except when the Boston Red Sox win the World Series in the fourth game during a total lunar eclipse, in which case a second ball is added to celebrate.

Specific knowledge may be deduced from general knowledge. General knowledge may be induced from specific knowledge. Thus it is often said that induction is the *inverse* of deduction, just as machine vision is the inverse of computer graphics rendering. Algorithms for inductive learning of logical knowledge are beyond the scope of this project. However, the interested reader can pursue this topic further through by investigating the field of *inductive logic programming (ILP)*. We recommend a survey article by Page and Srinivasan [**?**], chapter 19 of [**?**], and chapter 10 of [**?**].

Finally, we note that there are times when knowledge is much more easily *supplied* than deduced or induced. Consider again the Clue knowledge that a card is in exactly one location. We supply this knowledge to our system as part of the initial knowledge base. This is considerably easier than inducing the same knowledge through repeated observation. However, many have observed that the difficult part of logical reasoning is not the reasoning itself but the acquisition of the knowledge. *Knowledge acquisition* is often a significant engineering challenge of knowledge-based systems, and this challenge is often referred to as the *knowledge acquisition bottleneck*.

The preceding project often illustrates how difficult it can be for one who is an “expert” on Clue rules to completely and correctly encode all fundamental knowledge of the game. Imagine building the same system without a guide as to what knowledge should be added to the knowledge base.

Returning to Mitchell’s definition, we see that the three “experiences” of deduction, induction, and knowledge acquisition can add knowledge to the knowledge base, thus potentially improving (or even making possible) the performance of the task of answering queries to the knowledge base.

10 Advanced Projects

In addition to this small project, there are several directions a student may take to go more in depth. These advanced projects are independent of one another, although the first may be combined with either of the latter two.

10.1 Counting Cards

One set of facts that we have omitted from the knowledge base concern the number of cards in a player's hand. In a game of Clue, players know the number of cards held by other players. In a six player game, each player holds three cards. In a three player game, each player holds six cards. In four and five player games, players hold differing numbers of cards, depending on position in the turn order.

Such constraints can be expressed verbally as "Player p has n cards." In our representation of facts, this equivalent to "Exactly n of c_p is true for all cards c." This, in turn, is equivalent to saying "At least n of c_p is true, and at least $(21 - n)$ of $\neg c_p$ is true." Sentences which place such "at least", "at most", and "exactly" constraints on a number of literals being true are often called **pseudo-boolean constraints**.

The goal of this advanced project is to augment our `clueReasoner` to make use of these cardinality constraints in some reasonably efficient manner. It is permissible to trade off completeness for efficiency, so long as one can demonstrate cases where the reasoning engine makes use of this knowledge.

There are a number of approaches one can take. Perhaps the simplest is to allow such reasoning outside of the `SATSolver`. In other words, one can look at the inferences of the `SATSolver`, and check if our cardinality constraints can yield further inferences. If so, we add clauses that represent such inferences to the `SATSolver`, and repeat the process.

Another approach requires knowledge of digital logic, specifically, how to design an adder circuit. If one views a set of literals as single bit numbers (0 = false, 1 = true), then cardinality constraints express that we wish the sum of these single-bit numbers to be in a certain range. In the case of an "exactly" form of constraint, we know the desired sum. While encoding a circuit that tests for a certain sum propositionally is a compact way to express such information, the impact of such a representation on the efficiency of reasoning is an open research question.

Perhaps the most promising approach is to work directly with the pseudo-boolean constraints with an algorithm such as the WalkSAT generalization WSAT(PB) [**?**]. A survey of such algorithms can be found in chapter 6 of [**?**]. These algorithms can work with constraints of the form "At least n of this set of literals is true." *All* propositional facts of Clue can be put concisely into this form. Our facts in CNF form can be thought of as a specific case of this where $n = 1$ for each clause. One can thus generalize DPLL and stochastic local search algorithms by adding an additional number for each generalized clause that indicates how many literals must be true for the clause to be satisfied.

This is an area rich with possibilities for creativity and research challenges.

10.2 DPLL Implementation

The preceding project largely focused on the task of knowledge representation. We relied on an implemented `SATSolver` to perform propositional reasoning for us. It would be satisfying to complete both the knowledge representation and the reasoning components.

One way to do this is to extend the `SATSolver` implementation, overriding (replacing) the `makeQuery` method with a rudimentary implementation of DPLL as described in [**?**, **?**, **?**].

There are many possible optimizations. While none are strictly necessary, one may learn much about such implementations by choose one optimization, reading the literature on it, implementing it, and comparing performance with and without the optimization. For example, consider the "watched literal" optimization used to speed unit resolution [**?**].

10.3 Stochastic Local Search Implementation

Another possible solver to implement is based on *stochastic local search* [**?**]. Even inefficient implementations of the WalkSAT algorithm happen to perform very well for Clue reasoning. Descriptions of WalkSAT can be found in [**?**, **?**, **?**, **?**].

What is important to note is that we cannot perform proof by contradiction with stochastic local search. If we fail to find a satisfying truth assignment for a knowledge base and negated hypothesis, it may be the case that the search was simply unsuccessful. The fact that a search is stochastic does not mean that the search is also complete. This incompleteness of many stochastic local searches means that our proof procedure is unsound if we treat an unsuccessful incomplete stochastic local search as we do a contradiction.

Nonetheless, the performance of such an approach is quite impressive. For this problem, an unoptimized Java implementation of WalkSAT takes less time to run with high reliability than zChaff plus the overhead of the Java interaction with zChaff.

If desired, a simple implementation of WalkSAT can then be improved by examining high-performance implementations such as given in UBCSAT[15].

Further, one can examine further literature on WalkSAT improvements such as Novelty, Novelty+, and Adaptive Novelty+, the random category winner of the http://satlive.org/SATCompetition/2004/SAT 2004 competition for the SAT and ALL (SAT + UNSAT) benchmarks.

A high quality implementation of a stochastic local search SAT solver is a worthy and satisfying[16] advanced project.

[15]See URL `http://www.satlib.org/ubcsat/`.

[16]no pun intended