# CS 151, Spring 00
# Logic 2

Margaret M. Fleck

February 7, 2000

Reading: Russell and Norvig, chapters 7, 9, and sections 10.1-10.5.

## 1 Computer reasoning algorithms

The above rules work well for humans, but they are too complex to implement on a computer. They are also too complex for formal proofs about what reasoning algorithms do. Therefore, logicians and designers of reasoning algorithms prefer to work with very small sets of reasoning rules.

Specifically, the "resolution algorithm" derives proofs using the following procedure:

- Convert all sentences to a special form (CNF).
- Use only one inference rule, a form of resolution.
- Apply proof by contradiction. That is, start by assuming all your background information and the negative of the statement you want to prove, and derive a contradiction.
- Use search techniques to explore the space of possible proofs.

The resulting proofs don't look anything like what a normal mathematician would write, and they are difficult for a human to read. However, it's relatively easy to build a computer algorithm that creates such proofs.

A few AI programs have tried to apply a more complex set of reasoning rules, creating more human-like proofs. However, such algorithms have never gotten past the experimental stage, whereas some resolution theorem provers have become reliable packaged products that have proved some new theorems.

## 2 Conjunctive Normal Form

Most theorem proving algorithms convert all sentences from FOPC to "conjunctive normal form" (CNF). In CNF, we have a database of sentences. The content of the entire database can be thought of as the AND (conjunction) of the content of all the individual sentences.

Each sentence in the CNF database is a disjunction (OR): $\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n$. Each of the $\alpha_i$ is a "literal," i.e. an atomic sentence or the negation of an atomic sentence. Remember that an atomic sentence is a predicate applied to some number of terms. And each term is a variable, constant, or a function applied to some number of terms.

CNF sentences have no explicit quantifiers. Variables that were existentially quantified have been replaced with Skolem constants or functions. All remaining variables are assumed to be universally quantified.

So, for example, the sentence $\forall x$, cat(x) $\Rightarrow$ furry(x) $\wedge$ independent(x) is converted into the sentence furry(x) $\vee$ $\neg$cat(x) and the sentence independent(x) $\vee$ $\neg$cat(x).

When converting to CNF by hand, on paper, the steps are as follows:

- Rename all bound variables, so that each quantifier defines a variable with a distinct name.
- Remove $\Rightarrow$ and $Leftrightarrow$ using the equivalence $pRightarrowq \equiv q \wedge \neg p$.
- Move $\neg$ inwards, onto the smallest possible expressions, using the five equivalences for negation (see above).
- Move all quantifiers to the left, preserving their order.
- Replace all existentially quantified variables with Skolem constants and functions, removing all the existential quantifiers. The Skolem function replacing some variable x takes as inputs all the universally quantified variables whose quantifiers are to the left of x's quantifier.
- Delete all universal quantifiers.
- Move $\vee$ inside $wedge$ using the distributive law.
- Drop parentheses.
- If the sentence is a conjunction, divide it into multiple sentences by splitting at the $wedge$ operators.

When the sentence is represented inside a computer program, using linked list structure to encode grouping, the procedure is slightly different:

- Rename all bound variables, so that each quantifier defines a variable with a distinct name.
- Remove $\Rightarrow$ and $Leftrightarrow$ using the equivalence $pRightarrowq \equiv q \wedge \neg p$.
- Move $\neg$ inwards, onto the smallest possible expressions, using the five equivalences for negation (see above).
- Replace all existentially quantified variables with Skolem constants and functions, removing all the existential quantifiers. The Skolem function replacing some variable x takes as inputs all the universally quantified variables whose quantifiers are outside x's quantifier in the list/grouping structure.
- Delete all universal quantifiers.
- Move $\vee$ inside $wedge$ using the distributive law.
- When a disjunction ($vee$, OR expression) occurs inside another disjunction, drop the parentheses to make a disjunction taking three or more inputs. Do the same for conjunctions ($wedge$, AND expressions).
- If the sentence is a conjunction, divide it into multiple sentences by splitting at the $wedge$ operators.

In both cases, the procedure assumes that input sentences contain only bound variables, no free variables.

Some of the steps can be done in different orders. In particular, Russell and Norvig's procedure does variable renaming later. It is best to rename variables as early as possible because this avoids technical subtleties in later steps.

# 3 Skolemization

In the 1920's, a mathematician named Thoralf Skolem introduced (or perhaps formalized) a trick for removing overt existential quantifiers from FOPC sentences. If a sentence asserts the existence of an object satisfying certain properties, we can make up a name for one specific individual satisfying them and assert that these properties hold for that individual. The name we use must be a new name, not already used for anything else in the proof.

In formal logic, a new name introduced this way is called a "Skolem constant." It is a constant because it refers to one specific individual in the world. In other areas of mathematics, authors often call such names "new variables," using "variable" to mean a symbol whose exact value is not known. This is simply a difference in terminology: the concept is the same.

Formally, the rule is

$$\exists x, \alpha \vdash \mathrm{subst}(\{\mathrm{x/c}\}, \alpha)$$

where $c$ is a new constant. A concrete example of applying this rule might be:

$$\exists x, \mathrm{climbed(Sprague, x)} \wedge \mathrm{placed(banner, x)} \wedge \neg \mathrm{caught(x)} \vdash \mathrm{climbed(Sprague, culprit)} \wedge \mathrm{placed(banner, culprit)} \wedge \neg \mathrm{caught(culprit}$$

When the sentence has a mixture of existential and universal quantifiers, the value of an existentially quantified variable may depend on the values of universally quantified variables defined first. In such cases, we introduce a "Skolem function" rather than a Skolem constant. For example:

$$\forall x, \exists r, \mathrm{occupies(s, r)} \vdash \forall \mathrm{x}, \mathrm{occupies(s, room - of(x))}$$

Formally

$$\forall x_1, ldots, x_n, \exists y, \alpha \vdash \mathrm{subst}(\{\mathrm{y/f(x_1, \ldots, x_n)}\}, \alpha)$$

$f$ must be a new function name, not previously used in the proof. The inputs to $f$ must be the variables defined by all the universal quantifiers that are outside the existential quantifier that defines $y$.

# 4 The resolution rule

The most basic form of the resolution rule is $\neg\alpha \Rightarrow \beta, \alpha \Rightarrow \gamma \vdash \beta \vee \gamma$. To create proofs with this rule, we would need to combine it with other inference rules. However, if we extend the resolution rule somewhat, it can be used as the only inference rule for sentences in conjunctive normal form.

First, we make resolution take an arbitrary number of disjuncts, in an arbitrary order. That is

$$\begin{array}{ll} p_1 \vee \ldots \vee p_j \vee \ldots \vee p_n & \\ q_1 \vee \ldots \vee q_k \vee \ldots \vee q_m & \\ & \vdash \quad p_1 \vee \ldots \vee p_{j-1} \vee p_{j+1} \vee \ldots \vee p_n \vee \\ & \quad\;\; q_1 \vee \ldots \vee q_{k-1} \vee q_{k+1} \vee \ldots \vee q_m \end{array}$$

where $p_j = \neg q_k$. That is, if $p_j = \neg q_k$, you derive the output of resolution by appending the two lists of terms but deleting $p_j$ and $q_k$.

Finally, resolution works when $p_j$ is not the exact literal negation of $q_k$ but, rather, only the negation of something that matches $q_k$ if you replace some variables in the two expressions with the right values. The matching process is "unification," the algorithm you implemented in assignment 1.

Specifically, suppose that $p_j$ and the negation of $q_k$ can be unified, using the substitution list $\theta$. Then the output of unification is created by appending the two lists of inputs, but deleting $p_j$ and $q_k$ and applying the substitutions in $\theta$ to the remaining terms.

## 5 Proving things with resolution

So, a preliminary attempt at a resolution theorem proving algorithm might look as follows:

- Convert inputs (background information and goal) to CNF.
- Use a search algorithm (e.g. BFS) to generate all resolution proofs using the input information.
- Halt when the goal sentence is produced.

For example, suppose that we want to prove $\mathrm{in(beans, bottom - shelf)}$ using the following information about putting away food.

- $\forall x, y, \mathrm{goes - in(x, y)} \wedge \neg \mathrm{out(x)} \Rightarrow \mathrm{in(x, y)}$
- $\forall x, \mathrm{canned(x)} \wedge \mathrm{vegetable(x)} \Rightarrow \mathrm{goes - in(x, bottom - shelf)}$
- $\mathrm{canned(beans)}$
- $\mathrm{vegetable(beans)}$
- $\neg\mathrm{out(beans)}$

In CNF, these would look like:

- $\neg\mathrm{goes - in(x, y)} \vee \mathrm{out(x)} \vee \mathrm{in(x, y)}$
- $\neg\mathrm{canned(x)} \vee \neg\mathrm{vegetable(x)} \vee \mathrm{goes - in(x, bottom - shelf)}$

- canned(beans)
- vegetable(beans)
- $\neg$out(beans)

From $\neg\text{canned}(x) \vee \neg\text{vegetable}(x) \vee \text{goes}-\text{in}(x, \text{bottom}-\text{shelf})$ and canned(beans), the system can infer $\neg\text{vegetable}(\text{beans}) \vee \text{goes}-\text{in}(\text{beans}, \text{bottom}-\text{shelf})$.

Combining this with vegetable(beans), the system then infers $\text{goes}-\text{in}(\text{beans}, \text{bottom}-\text{shelf})$.

Combining this with $\neg\text{goes}-\text{in}(x, y) \vee \text{out}(x) \vee \text{in}(x, y)$, we then get $\text{out}(\text{beans}) \vee \text{in}(\text{beans}, \text{bottom}-\text{shelf})$.

Finally, combining this with $\neg$out(beans), we get $\text{in}(\text{beans}, \text{bottom}-\text{shelf})$.

Unfortunately, this above procedure cannot prove all goals that could be inferred from the input information. For example, suppose that our input information is the single sentence overcrowded(Platt). Direct proof using resolution can't derive $\text{overcrowded}(\text{Platt}) \vee \text{too}-\text{large}(\text{number}-\text{of}(\text{students}))$. The resolution rule gives us no way to introduce the second term in the goal, because it does not occur anywhere in the input sentences.

## 6 Resolution proof by contradiction

To be able to prove any goal using resolution, we have to use proof by contradiction. That is

- Negate goal.
- Convert background information and negated goal to CNF.
- Use a search method (e.g. BFS) to generate all resolution proofs starting from the background information and negated goal.
- Halt when the search procedure produces a null sentence, i.e. a contradiction.

Aside: a sentence with no literals is always false. This is because sentences in CNF are disjunctions and false is the identity element for OR. That is, you can OR false with any other value and always get the second value back. Some programming languages allow you to apply OR to no inputs, and sometimes you need to compute such a value as the last step in a recursive procedure. False is the right value to return because it is the identity value. Similarly, the identity for AND is true.

So, our revised resolution procedure might start with the negated goal $\neg\text{in}(\text{beans}, \text{bottom}-\text{shelf})$. Combining this with $\neg\text{goes}-\text{in}(x, y) \vee \text{out}(x) \vee \text{in}(x, y)$, we get $\neg\text{goes}-\text{in}(\text{beans}, \text{bottom}-\text{shelf}) \vee \text{out}(\text{beans})$.

Combining this result with $\neg\text{canned}(x) \vee \neg\text{vegetable}(x) \vee \text{goes}-\text{in}(x, \text{bottom}-\text{shelf})$, we get $\text{out}(\text{beans}) \vee \neg\text{canned}(x) \vee \neg\text{vegetable}(x)$.

Combining this with canned(beans), we get $\text{out}(\text{beans}) \vee \neg\text{vegetable}(x)$.

Combining this with vegetable(beans), we get out(beans).

Finally, we combine that with $\neg$out(beans), to get a sentence with no literals, our contradiction.

## 7 Facts about resolution

So, our final algorithm is a combination of resolution, proof by contradiction, and a search method (e.g. BFS) that can't get stuck in an infinite loop. Such an algorithm is

- Sound: if the algorithm returns success, the goal follows from the inputs.
- Complete: if the goal follows from the inputs, the algorithm will eventually return success.

Proofs of soundness can be found in a logic text. The basic idea is to show that the procedure works correctly in the simplest cases, and gradually build up a proof for more complicated cases. See a logic textbook.

Section 9.7 Russell and Norvig sketches a proof that resolution is complete. The basic idea is to reduce the problem of proving claims in FOPC to a problem of proving things when sentences have no variables.

Unfortunately, our resolution algorithm also has the following bad properties:

- Only semi-decidable: if the goal does not follow from the inputs, the algorithm is not guaranteed to halt. It may get into an infinite loop
- NP-hard: in the worst case, the size of the search space grows exponentially.

These two problems can't be fixed by tinkering with the algorithm.

Proofs that theorem proving for FOPC is only semi-decidable can be found in theory of computation textbooks. The essence of the problem is that there is no way to know in advance how long a proof will be. (Remember the knot equivalence example from search?)

To understand how a resolution theorem prover might get into an infinite loop, consider what happens if we combine the sentences $\neg P(f(x)) \vee P(x)$ and $\neg P(f(y))$. $P(x)$ and $\neg P(f(y))$ unify using the substitution $x = f(y)$. So the result of resolution is the sentence $\neg P(f(f(y)))$. If we then combine $\neg P(f(f(y)))$ and $\neg P(f(x)) \vee P(x)$ and $\neg P(f(y))$, we get $\neg P(f(f(f(y))))$. This can be repeated to create a sentence with as many applications of $f$ as desired.

Theory of computation textbooks are also the right place to look for proofs that FOPC theorem proving is NP-hard. The basic problem apparently involves proving goals such as $\alpha \vee \beta$. To prove this, one must explore two alternative lines of proof: prove $\alpha$ or prove $\beta$. Now imagine setting up a problem that had many such choice points: each one would expand the search space by a factor of two.

## 8 Incompleteness

Another weak point of FOPC theorem proving was shown by Gödel in his Incompleteness Theorem. Suppose that we are given a formal theory which contains ordinary arithmetic, is consistent, and in which the axioms and rules of inference are finite or recursively specified. The theorem then states that there is a sentence $S$ such that neither $S$ nor $\neg S$ can be proved within the system.

The proof is very general, so its conclusion can't be changed by changes to FOPC (e.g. second-order logic) or to the theorem-proving algorithm.

The proof uses a "diagonalization" argument, similar to proofs of the Halting Problem, Russell's paradox, or the uncountability of the real numbers. (You've probably seen one of these.)

First, all sentences in the theory are put in an order, numbered by the integers. This involves some clever hackery with general-purpose algorithms for converting sentences to numbers.

Then, we create a sentence $S$ which states that it, itself, cannot be proved. This requires that $S$ refer to itself, accomplished by using the index number for $S$, in a way that reverses polarity (involves negation). If you've seen the proof that the reals are uncountable, remember that you create a real not in the list by changing/reversing all the digits along the diagonal.

So we then get a contradiction of the following sort. Suppose we can prove $S$. Then $S$ is true. But the content of $S$ is that $S$ cannot be proved. So $S$ cannot be proved. This contradicts our assumption that $S$ can be proved.

The Saunders MacLane book (cited above) gives more details and references.

So how do we square the incompleteness theorem with the completeness of resolution? Resolution can prove anything that can be deduced from the input sentences. The bug is that a finite set of FOPC sentences cannot capture everything that is true about even a simple domain such as arithmetic. If we write a set of axioms that characterize arithmetic, we can always find a model which satisfies all the axioms but doesn't have the same structure as standard arithmetic.

For example, there is a set of numbers called the "hyperreals" which satisfy the standard axioms for the real numbers, but also include infinitely large and small numbers. The hyperreals can be used to prove facts in calculus (e.g. what is the derivative of some function) using algebra rather than $\epsilon$-$\delta$ proofs. We can add more axioms to distinguish the reals from the hyperreals, but Gödel's theorem implies that there will always be other alternative models.

# 9 Implementing resolution

To implement resolution, we need a way to structure the search process that generates all resolution proofs. One way to do this is to set up two collections of sentences: (a) a set $S$ of sentences that have already been processed and (b) a queue of sentences still to be considered. Breadth-first search can be used to ensure that we will definitely find a contradiction, if one exists.

The algorithm them looks like:

- Set $S$ to be the null set.
- Set the queue to contain the negated goal and all the input background information.
- Remove the first sentence $\alpha$ from the queue.
- If $\alpha$ is not in $S$, resolve $\alpha$ against all sentences of $S$, add the results of the resolutions (if any) to the end of the queue, and add $\alpha$ to $S$.
- Go back to the second step and repeat until a null sentence (contradition) is generated.

# 10 Implementation subtleties

When implementing resolution, be careful of two subtleties. First, before attempting to resolve two sentences, first make sure that variables have been renamed so that the two sentences have different variable names. This can be done when sentences are put into the database or right before you do the resolution.

Second, notice that some pairs of sentences can be resolved in more than one way. For example $P(A) \vee L(B)$ and $\neg P(x) \vee \neg L(x)$ can be resolved to produce either $P(A) \vee \neg P(B)$ or $L(B) \vee \neg L(A)$.

## 11 Useless sentences

When doing resolution, some sentences can be eliminated as useless or redundant. Looking for such sentences may or may not be worth it. It's a tradeoff between the decrease in the search space vs. the effort required to identify the sentences to be discarded.

Duplicate literals in a sentence are easy to detect and obviously useless. For example, edible(beans) $\vee$ edible(beans) can be reduced to edible(beans).

A sentence need not be saved if a stronger one is already in the database. For example, edible(beans) $\vee$ edible(okra) isn't worth saving if edible(okra) is already known.

A sentence is useless if it contains both the positive and negative versions of the same literal. Such sentences are always true, so they tell us nothing about the specific problem. Seen another way, resolution can never remove the matching positive and negative pair to extract the other literals. And if we resolve a sentence like $\alpha \vee \neg\alpha \vee \beta$ with some sentence $\alpha \vee \gamma$, we get the sentence $\alpha \vee \gamma \vee \beta$. But $\alpha \vee \gamma \vee \beta$ isn't worth keeping since we already know the stronger sentence $\alpha \vee \gamma$.

Sentences containing $\alpha$ and $\neg\alpha$ can be produced from innocent-looking sentences. For example, $P(x) \vee R(x) \vee B(s)$ and $\neg P(A) \vee \neg R(A) \vee M(t)$ resolve to produce $R(A) \vee \neg R(A) \vee B(s) \vee M(t)$.

Don't bother resolving a sentence against itself, because the result with contain some term $\alpha$ and also its negation.

Finally, there are versions of the above rules involving pairs of literals such as $P(A)$ and $P(x)$. For example, the sentence edible(okra) $\vee$ edible(x) can be simplified to edible(okra). (Whichever side of the sentence is true, the okra must be edible. However, since we know that the okra is edible, the sentence tells us nothing about any other objects.)

## 12 Other search strategies

Naive versions of resolution search tend to run very slowly. From a set of input sentences, resolution can produce a *very* large number of output sentences. So the search queue for breadth-first search tends to expand very fast.

The resolution algorithm can easily be modified to use a variety of search strategies, which improve running time and/or memory usage. Some of these are still guaranteed to find a contradiction, if one exists, but some are not.

**Depth-first search and iterative deepening:** Using depth-first rather than breadth-first search speeds up the algorithm substantially. However, it is no longer complete because it can get stuck in infinite loops. Iterative deepening can be used to largely gain the advantages of both breadth-first and depth-first search.

**Unit preference:** Resolution often uses a version of best-first search called "unit preference." In this strategy, sentences consisting of only one literal are kept in the queue ahead of longer sentences. These "unit clauses" are special, because resolving them with any sentence always produces a shorter sentence. First trying resolutions with unit clauses is a good strategy because the algorithm's ultimate goal is to produce the shortest-possible sentence: one containing no literals.

The longer sentences in the queue may or may not be kept ordered by length, depending on who's telling the story. It is also possible to rate the complexity of sentences using factors beyond number of literals. For example, also considering the number of symbols in the sentence could discourage the algorithm from getting into loops such as producing the $P(f(f(f(f(f(...(y)))))))$ sentences discussed above.

In general, using unit preference makes the search algorithm incomplete, because it can get stuck in loops. However, it has a very good reputation in theorem proving: it makes search converge much faster and rarely gets into loops on real problems.

**Set of support:** A much better way to reduce search is to do only resolutions in which one of the sentences is the negated goal or something derived from the negated goal. This strategy is called "set of support." It can be implemented in our queue algorithm by putting all background information into the set $S$ (set of already processed sentences) and making the initial queue contain only the negated goal.

**Linear resolution:** Alternatively, we can use a strategy called "linear resolution" or "ancestry filtering." This strategy only resolves two sentences if (a) one is in the original inputs or negated goal or (b) one was generated from the other.

Both set of support and linear resolution focus the search for proofs but do not prevent the search algorithm from being complete.

# 13 Organizing the database

Resolution can also be speeded up by organizing the database of sentences. In our algorithm, we need to repeatedly ask whether a sentence $\alpha$ is already in the set $S$. When $S$ gets large, it is inefficient to do this by searching linearly through $S$.

Remember that $S$ contains all the background information known to our theorem prover, as well as facts derived during the search for a proof. For a simple problem in set theory, $S$ might start small but grow larger as the search progresses. Real-world problems frequently use large databases of "commonsense" information about the world. Ambitious versions of such databases contain vast quantities of information (most of it irrelevant to any specific problem).

To do this, we can store $S$ as a hash table. The hash keys are predicates used in the sentences. Each sentence is store under all predicates used in it. For example, the sentence $\text{high}(\text{rating}(\text{Bill})) \vee \neg\text{attacking}(\text{Kenneth}, \text{Bill})$ would be stored under high and also under attacking.

One refinement of this strategy subdivides the indexing for each predicate. Sentences using the predicate positively are stored separately from those using it negatively. Single-literal sentences (unit clauses) may be stored separately from longer sentences.

If many sentences share the same predicate, one can sub-index by the values used in the predicate. This works best if one first indexes by arguments for which few sentences share the same value (e.g. SSN is better than first name). One can index each sentence in multiple ways (using different arguments) and have each individual query retrieve using the indexing method that is most appropriate. For example, if some inputs to the predicate are variables and some are constants, one should first attempt to look up the sentence using the arguments which have constant values.

Indexing a large, complicated database eventually requires one to use the full power of database management algorithms. This is beyond the scope of an AI course: see a textbook or course on databases.