Specific to General Learning via Least-General Generalizations

- We will describe the Golem learning algorithm, which is available for download and is much more practical than inverse resolution, but is based on specific-to-general search rather than general-to-specific.

- Golem attempts to solve the same learning problem as FOIL.

Given:

- $B$ = background knowledge of ground atoms
- $P$ = positive examples
- $N$ = negative examples

Find:

$H$ s.t. $H$ is a definite clausal theory

$H \land B \models P$

$\forall n \in N, H \land B \not\models n$

Golem however does allow for the learned clauses to include function symbols,
Furthermore Golem uses a similar outer loop as FOIL, where one rule is learned at a time and examples are removed at each step.

\[
\begin{align*}
R &= \emptyset \\
\text{While } P \neq \emptyset \text{ do} \\
& \quad r := \text{Learn Rule}(P, N) \\
& \quad P := P - \{e \in P : \forall r \in e\}
\end{align*}
\]

All of the work is done in Learn Rule, and which has the goal of returning a definite clause that covers as many positives as possible and no negatives.

Golem uses a specific-to-general approach to implementing Learn Rule (also called a bottom-up approach).
Golem's LearnRule is based on the concept of least-general generalization (LGG) which we will now cover.

Intuitively, given two clauses $C_1$ and $C_2$ (you can think of these as rules), we can talk about the notion of the LGG of $C_1$ and $C_2$, which is a clause $C$ that is more general than $C_1$ and $C_2$, but the least general such clause.

Let's denote our generality relation as $\preceq$.

More formally, $C = \text{LGG}(C_1, C_2)$ if $C \succeq C_1$ and $C \succeq C_2$ and there is no $C'$ s.t. $C' \preceq C$ and $C' \succeq C_1$ and $C' \succeq C_2$ (strictly less general).
According to this definition, an LGG of $C_1$ and $C_2$ covers all of the positive examples that $C_1$ and $C_2$ cover, but as few additional as possible.

One issue that arises in theory and practice is what definition of generality ($\leq$) we should use for the LGG process.

Intuitively it would make sense to define generality in terms of entailment:

$$C_1 \leq C_2 \iff C_2 \models C_1$$

Under this definition, $C_2$ will cover all the positives that $C_1$ covers. To see this suppose $C_1 \models e$ and $C_1 \leq C_2$ then we get

$$C_2 \models e$$

as desired.
However, using entailment as the notion of generality has 2 main problems:

1) The LGF is not unique under entailment.
2) Computing entailment between clauses (even Horn clauses) is undecidable in general.

For these reasons researchers have instead focused primarily on another notion of generality which can be checked and has a unique LGF.

For the following, we will think of clauses as being represented as sets of literals:

- For example, \( C = \{a \lor b \rightarrow c\} \) is represented as \( C = \{a, b, c\} \).
Def' let \( C_1 \) and \( C_2 \) be two clauses. \( C_2 \Theta \)-subsumes \( C_1 \) iff there is a substitution \( \Theta \) such that \( C_2 \Theta \subseteq C_1 \).

For example:

\[ C_2 : \text{parent}(y,x) \rightarrow \text{daughter}(y,x) \]
\[ C_1 : \text{parent}(y,x) \land \text{female}(x) \rightarrow \text{daughter}(y,x) \]
\[ C_3 : \text{parent}(\text{alan}, \text{sophie}) \land \text{female}(\text{sophie}) \rightarrow \text{daughter}(\text{sophie}) \]

Here we have that

\( C_2 \Theta \)-subsumes both \( C_1 \) and \( C_3 \) and
\( C_1 \Theta \)-subsumes \( C_3 \) (not vice versa).

\( \Theta \)-subsumption implies implication but

\( \Theta \)-subsumption is a sufficient but not necessary condition for entailment.
Proposition: If $C_2 \theta$-subsumes $C_1$, then $C_2 \models C_1$.

proof: Assume there is a $\theta$ s.t. $C_2 \theta \subseteq C_1$.

To show $C_2 \models C_1$, assume that $C_2$ is true and then try to prove $C_1$.

Assume $C_2$ is true.

We know that for any $\theta$ that $C_2 \theta$ is also true since variables in $C_2$ are universally quantified. In particular for the $\Theta$ that demonstrates $\theta$-subsumption $C_2 \theta$ is true. Now $C_1$ is just $C_2 \theta \cup \{C_1 - C_2 \theta\}$ or equivalently we can get $C_1$ by adding disjuncts to $C_2 \theta$, but $C_1$ must also be true because $A \vee B$ is true if $A$ is true for any $B$. This shows that $C_1$ is true.
The proposition shows \( \theta \)-subsumption is sufficient for entailment.

The following shows it is not necessary.

\[
C_2 : P(x) \rightarrow P(f(x)) \\
C_1 : P(x) \rightarrow P(f(f(x)))
\]

we have that

\[C_2 \models C_1 \quad \text{(why?)}\]

but \( C_2 \) does not \( \theta \)-subsume \( C_1 \).

But we have gained the

But let's go ahead and define our notion of generality in terms of \( \theta \)-subsumption.

\[C_1 \leq C_2 \text{ iff } C_2 \ \theta \text{-subsumes } C_1\]

Still, \( C_2 \) covers all positives that \( C_1 \) does.
With this notion we have gained a computable generality relation (requires finding a substitution).

Also we have gained the property that LGG are unique wrt ≤, (unique up to variable renaming and certain other normalizations).

So now the question is how to compute LGG(C₁, C₂) under θ-subsumption.

Intuitively the procedure is a type of "anti-unification" of C₁ and C₂. The procedure we will define works for any clause, definite or not, possibly involving function symbols.

We will define the LGG of terms followed by the LGG of literals, followed by the LGG of 2 clauses.
LG G Definition for Literals

Terms:
- \( LG G(t, t) = t \) (identical terms)
- \( LG G(f(t_1, ..., t_n), f(s_1, ..., s_n)) = f(LG G(t_1, s_1), ..., LG G(t_n, s_n)) \)

If \( t = f(t_1, ..., t_n) \) and \( t' = g(s_1, ..., s_m) \) for \( f \neq g \):
- \( LG G(t, t') = V_{tt'} \), where \( V_{tt'} \) is a new variable associated with \( t \) and \( t' \)
- \( LG G(s, t) = V_{st} \), if \( s \) or \( t \) is a variable

Atoms:
- \( LG G(p(t_1, ..., t_n), p(s_1, ..., s_n)) = p(LG G(t_1, s_1), ..., LG G(t_n, s_n)) \)
- \( LG G(p(t_1, ..., t_n), q(s_1, ..., s_m)) \) is undefined when \( p \neq q \)

Literals: (a literal is an atom or its negation)
- For positive literals \( L_1 \) and \( L_2 \)
  - We get the \( LG G \) between atoms
- For \( L_1 = \neg A_1 \) and \( L_2 = \neg A_2 \) where
  - \( A_1 \) and \( A_2 \) are atoms
  - \( LG G(L_1, L_2) = \neg LG G(A_1, A_2) \)
- If \( L_1 \) and \( L_2 \) are of opposite polarity then \( LG G(L_1, L_2) \) is undefined

Examples:
- \( WEAN(\neg a, b) \Rightarrow LG G(R(a, c), R(a, d)) = R(a, v_{ca}) \)
- \( LG G(R(c, f(c)), R(a, f(a))) = R(v_{ca}, f(v_{ca})) \)
- \( LG G(R(c, f(c)), R(a, f(a))) = R(v_{ca}, f(v_{ca})) \)
LGG for Clauses:

Suppose $C_1 = \{L_1, \ldots, L_n\}$
$C_2 = \{K_1, \ldots, K_m\}$
are 2 clauses with $n$ and $m$ literals each.

$LGG(C_1, C_2) = \{L' \in LGG(L_i, K_j) \mid L_i \in C_1, K_j \in C_2, \text{LGG}(L_i, K_j) \text{ is defined}\}$

That is, $LGG(C_1, C_2)$ is the disjunction of all pairwise LGGs between literals of $C_1$ and $C_2$ that are well defined.

Example:

$C_1 = \neg \text{female}(\text{sophie}) \lor \neg \text{parent}(\text{alan}, \text{sophie}) \lor \text{daughter}(\text{sophie}, \text{alan})$

$C_2 = \neg \text{female}(\text{jane}) \lor \neg \text{parent}(\text{bob}, \text{jane}) \lor \text{daughter}(\text{jane}, \text{bob})$

$LGG(C_1, C_2) = \neg \text{female}(x) \lor \neg \text{parent}(V_{ax}, V_{ay}) \lor \text{daughter}(V_{ax}, V_{ay})$

$\quad = \text{female}(x) \land \neg \text{parent}(x, y) \rightarrow \text{daughter}(x, y)$

$C_1 = \neg \text{parent}(\text{alan}, \text{sophie}) \lor \neg \text{grandfather}(\text{father}(\text{alan}), \text{sophie})$

$C_2 = \neg \text{parent}(\text{xiaoli}, \text{sophie}) \lor \neg \text{grandfather}(\text{father}(\text{xiaoli}), \text{sophie})$

$LGG(C_1, C_2) = \neg \text{parent}(V_{ax}, \text{sophie}) \lor \neg \text{grandfather}(\text{father}(V_{ax}), \text{sophie})$

$\quad = \neg \text{parent}(x, \text{sophie}) \rightarrow \neg \text{grandfather}(\text{father}(x), \text{sophie})$
- The LGG operator just described does indeed compute a least-general clause that generalizes the inputs.
- Recall that \( C_1 \leq C_2 \) means \( C_2 \) \( \theta \)-subsumes \( C_1 \).

**Theorem:** Let \( C = LGG(C_1, C_2) \) be the clause computed by the above procedure. For any clause \( C' \), if \( C_1 \leq C' \) and \( C_2 \leq C' \) then \( C \leq C' \).

- Furthermore, it can be verified that \( LGG(C_1, C_2) \) does \( \theta \)-subsume both \( C_1 \) and \( C_2 \) as desired.
Note that the LGG operator is commutative and associative

\[
\text{LGG}(c_1, c_2) = \text{LGG}(c_2, c_1)
\]

and

\[
\text{LGG}(\text{LGG}(c_1, c_2), c_3) = \text{LGG}(c_1, \text{LGG}(c_2, c_3))
\]

Also note that for any set of clauses \( C = \{c_1, c_2, \ldots, c_n\} \) we can define the LGG for \( C \) to be the least general clause that is more general (or equivalent) to each \( c_i \).

We can compute the LGG of a set of examples via pairwise LGG operations:

\[
\text{LGG}(\{c_1, \ldots, c_n\}) = \text{LGG}(c_1, \{c_2, \ldots, c_n\})
\]

\[
\text{LGG}(\{c_1, \ldots, c_n\}) = \text{LGG}(\{c_1, \ldots, c_{n-1}\}, c_n)
\]

From above the order of pairwise LGGs does not matter.
- How can we use the LGG to solve our problem of learning a rule?

Given: B, P, N
Return: a rule (i.e., definite clause) \( r \) such that for many \( e \in P \)
\( B \land r \models e \) and for all \( e \in N \), \( B \land r \not\models e \).

---

- The high-level approach will first translate each positive example \( e \) into a very specific ground clause \( s(e) \) s.t. \( B \land s(e) \models e \).
However, each such \( s(e) \) will be so specific that it will not cover any other examples.

- Given the set of specific clauses 
\( C = \{ s(e) : e \in P \} \) we will then try to find a subset \( C' \) of 
\( C \) such that the rule
\( r = \text{LGG}(C') \) covers many positives and no negatives.
- Let's now consider each of these steps.

### Converting Positive Examples to Clauses

- Given a positive example \( e \) (e.g., \( Q(a,b) \)) we know that the clause \( \rightarrow e \) is a more general clause that covers \( e \). However this clause is quite general as it predicts \( \rightarrow e \) regardless of \( B \).

- We want to get a very specific clause that covers \( e \) relative to \( B \).

- Recall that \( B \) is a conjunction (or set) of ground facts. A very specific clause that covers \( e \) relative to \( B \) is:

\[
c = B \rightarrow e
\]

Indeed \( B \land c \not\models e \) but if we generalize \( c \), by removing a tit specialize \( c \) by adding facts not in \( B \) to the body then \( e \) will no longer be covered. So \( c \) is very specific.
Thus in a strong sense

\[ B \rightarrow e \] is the most specific clause relative to \( B \) that covers \( e \).

Thus, one choice would be to define \( \gamma(e) \) to be \( B \rightarrow e \).

However, \( B \rightarrow e \) will have many facts in the body that are completely unrelated to \( e \), meaning that the body is intuitively much bigger than it needs to be.

Example:

\[ e = \text{Rich}(John) \]

\[ B = \{ \text{HasNiceCar}(John), \text{HasGoodJob}(John), \text{HasNiceCar}(Jane), \text{HasGoodJob}(Jane) \} \]

\( B \rightarrow e \) includes facts about Jane in the body, which does not appear to be very useful. It seem more natural to limit the body to facts related to John.
In this case we would get

\[ \text{HasNiceCar(John)} \land \text{HasGoodJob(John)} \rightarrow \text{Rich(John)} \]

While the above is not strictly as specific as \( B \rightarrow e \), it is most specific when we restrict the body to facts that are linked to the head by some chain of predicates.

We will call such most specific clauses for \( e \) relative to \( B \) the saturation of \( e \) wrt \( B \), denoted \( s(e) \).

So the saturation is just \( B' \rightarrow e \) where \( B' \subseteq B \) contains all facts that are related somehow to \( e \).

Example:

\[ B = \{ R(a,b), R(b,c), R(c,d), P(b), R(s,t), R(t,u), P(t) \} \]

The saturation of \( e = Q(a,d) \) is

\[ R(a,b) \land R(b,c) \land R(c,d) \land P(b) \rightarrow Q(a,d) \]

related to \( e \) via chain of \( R \)'s.

\( R(s,t), R(t,u), P(t) \) not included since they do not link to \( e \).
Now that we have the notion of saturation we can turn all of the positive examples into clauses. We will think of these as the most specific clauses that entail the examples with the background knowledge B.

Given the saturations for e₁ and e₂ denoted \( s(e₁) \) and \( s(e₂) \) we know that \( LGG(s(e₁), s(e₂)) = C₁₂ \) covers both \( e₁ \) and \( e₂ \); i.e.

\[
B \land C₁₂ \models e₁ \land e₂
\]

and is the least general clause that does so.

We also know that if there is a negative example \( n \in N \) s.t. \( B \land C₁₂ \models n \) then no generalization of both \( s(e₁) \) and \( s(e₂) \) can avoid covering \( n \) (since \( LGG \) is least).
This suggests we search for the largest subset of saturated clauses that whose LGG covers no negatives. The resulting clause will cover many negatives and no positives.

There are two challenges with this approach:

1) Finding the best subset of saturated examples for which to compute GCG,
2) Dealing with LGGs that get very large.

We will first talk about (1) and in particular describe how Golem solves this task problem.

First, let's get some intuition about the problem.
Recall that the overall GOLEM algorithm is aimed at learning a set of rules $r_1, \ldots, r_n$ that together cover all exempt positives. This intuitively means that we can think of the positive examples as being grouped according to the rule that will be used to cover it.

For example, perhaps we have 4 pos examples:
\[
\{ \text{sticky}(m_1), \text{sticky}(m_2) \} = p
\]
\[
\{ \text{sticky}(m_3), \text{sticky}(m_4) \} = p
\]
and 1 neg example \[
\{ \text{sticky}(m_5) \} = n
\]
and $m_1$ and $m_2$ are sticky for a different reason than $m_3$ and $m_4$. That is, there are different rules for them.

Consider $B = \{ p(m_1), p(m_2), q(m_3), q(m_4) \}$

the saturations are:
\[
p(m_1) \rightarrow \text{sticky}(m_1) = s_1
\]
\[
p(m_2) \rightarrow \text{sticky}(m_2) = s_2
\]
\[
q(m_3) \rightarrow \text{sticky}(m_3) = s_3
\]
\[
q(m_4) \rightarrow \text{sticky}(m_4) = s_4
\]
We can verify that

\[
LGG(s_1, s_2) = \mathcal{P}(x) \rightarrow \text{sticky}(x)
\]

\[
LGG(s_3, s_4) = \mathcal{Q}(x) \rightarrow \text{sticky}(x)
\]

which looks like a good set of rules for these example that do not cover negative examples.

But if we combine examples from different underlying rules we will over generalize:

\[
LGG(s_1, s_3) = \rightarrow \text{sticky}(x)
\]

which covers the negative example.

This examples shows that overgeneralization will often occur when we combine examples for whose underlying predictive rules differ.

So the rule learner is essentially trying to discover which sets of example belong to the same underlying rule and then generalize to get that rule.
Now let's describe Golem's LearnRule:

For clause \( C \), example set \( E \), and background facts \( B \) we define:

\[
\text{covers}(c, B, E) = \{ e \in E \mid B \cup \{c\} \models e \}
\]

i.e., the set of examples entailed by \( B \) and \( C \).

Our input to LearnRule is \( B, P, N \) and we want to output a clause/rule \( C \) that maximizes

\[
| \text{covers}(c, B, P) | \text{ under the constraint that } \text{covers}(c, B, N) = \emptyset.
\]

Golem will first find a seed pair of examples to generalize (picking greedily according to coverage) and then greedily include one new pair at a time into the generalization as long as no negatives are covered.
Learn Rule \((B, P, N)\)

\[ I = \{ c = \text{LGG}(s(e), s(e')) | e \in B, e' \notin B, \text{covers}(c, B, N) = \emptyset \} \]

Repeat

\[ C^* = \arg \max_{c \in I} |\text{covers}(c, B, P)| \]

\[ P = P - \text{covers}(C^* \cap B, P) \]

\[ I = \{ c = \text{LGG}(s(e), s(e')) | e \in B, \text{covers}(c, B, N) = \emptyset \} \]

Until \(I \neq \emptyset\)

Return \(C^*\)

---

- The main complexity of the algorithm is the repeated LGG computations.

- How expensive are these computations?

  - The time to compute \(\text{LGG}(c_1, c_2)\) is \(O(1^{11c_2})\) and the size of the resulting clause is also bounded by \(1^{11c_2}\).

- Unfortunately this shows that the size of an LGG resulting from \(n\) pairwise LGGs is only bounded by \(1^{11c_2 \ldots c_n}\).
This shows that the LGG can grow exponentially large in the # of clauses, and thus so can the time.

This is bad news for a practical achieving a practical algorithm.

Let's look at a good case and bad case for an LGG computation and then consider practical fixes.

**Good/Ideal Case:**

Positives: \( E_1 = Q(f(a), b) \)
\( E_2 = Q(f(c), d) \)

\( B = \{ R(a, b), P(b), R(c, d), P(d) \} \)

\( S(E_1) = \{ R(a, b), P(b), Q(f(a), b) \} \)

\( S(E_2) = \{ R(c, d), P(d), Q(f(c), d) \} \)

\( LGF(S(E_1), S(E_2)) = \{ 7R(V_a, V_b), 7P(V_b), Q(f(V_a), V_b) \} \)

\( = (R(x, y) \land P(y)) \rightarrow Q(f(x), y) \)

This clause is what we might expect.

Suppose that we now attempt to generalize this clause with a new example that is inherently explained by a different rule.
\[ S(e_3) = \{ R(s, t), \square N(u), \Box N(u), Q(f(u), v) \} \]

\[ LGG(\Box C, S(e_3)) = \{ R(V_{xs}, V_{ye}), Q(f(V_{xs}), V_{yu}) \} \]

\[ = R(V_{xs}, V_{ye}) \rightarrow Q(f(V_{xs}), V_{yu}) \]

\[ = R(x, z) \rightarrow Q(f(x), y) \]

- \( S(e_3) \) has a different predictive pattern than \( S(e_1) \) and \( S(e_2) \) and as a result when we generalize we get the above clause with which is very likely to be overly general and cover negative example.

- Thus Golem would not consider \( \text{WHENSTERS} \) of adding in \( S(e_3) \) an error and adding \( S(e_3) \) into this generalization.
\[ e_1 = \{ \text{Q}(a, c) \} \quad \text{Q}(a, c) \]

\[ e_2 = \{ \text{Q}(c, d) \} \quad \text{Q}(d, e) \]

\[ B = \{ R(a, b), R(b, c), P(b), P(c), R(d, e), R(e, f), P(e), P(f) \} \]

\[ S(e_1) = \{ \text{R}(a, b), \text{R}(b, c), \text{P}(b), \text{P}(c), \text{Q}(a, c) \} \]

\[ S(e_2) = \{ \text{R}(d, e), \text{R}(e, f), \text{P}(e), \text{P}(f), \text{Q}(d, e) \} \]

\[ \text{LGE}(S(e_1), S(e_2)) = \{ \text{R}(V_{ad}, V_{be}), \text{R}(V_{ae}, V_{be}), \text{R}(V_{ad}, V_{cf}), \text{R}(V_{ae}, V_{cf}), \text{P}(V_{ae}), \text{P}(V_{bf}), \text{P}(V_{ae}), \text{P}(V_{cf}), \text{Q}(V_{ad}, V_{cf}), \text{Q}(V_{ad}, V_{cf}), \text{Q}(V_{ad}, V_{cf}) \} \]

We can see that this clause appears to contain a lot of garbage in addition to what appears to be good patterns.

For example, the head literals \( R(V_{ae}, V_{bf}) \) and \( R(V_{ad}, V_{ce}) \) are not even linked to the head variables in any way while \( R(V_{ad}, V_{be}) \) and \( R(V_{ae}, V_{cf}) \) are.
In general full LGGs will produce many such excess patterns and are leading to the blowup in size of these clauses.

Note that when there is more background knowledge so that the saturations are larger, the problem becomes even worse.

There are at least three ways to deal with the blowup, each one being a way to prune the results of LGGs.

1) Remove logically redundant literals.

Note that in the above LGG whenever \( R(V_{ad}, V_{be}) \) is true then so is \( \square R(V_{ae}, V_{of}) \). This means that we can remove \( R(V_{ae}, V_{of}) \) from the clause and retain logical equivalence.

If we remove logically equivalent literals from the above we get:

\[
R(V_{ad}, V_{be}) \land R(V_{be}, V_{ae}) \land P(V_{be}) \land P(V_{cf}) \rightarrow Q(V_{ad}, V_{cf})
\]

\[
R(x, z) \land R(z, y) \land P(z) \land P(y) \rightarrow Q(x, y)
\]
This clause looks correct as it captures the essential pattern in the two examples.

It is not always possible.

But, in general testing for logically redundant literals requires expensive theorem proving, which is undecidable in the worst case.

Further, in many case the clauses can still be large after such pruning due to the fact that many unimportant patterns were captured. Where importance is wrt separating classifying positive from negative examples.

So in practice we usually consider simple logical simplification rules that can be checked quickly.
2) When the set of negative examples is sufficient, a very effective pruning rule is to remove a literal if after its removal no negatives are covered.

Golem has a greedy strategy for iteratively removing such literals. This is a very effective approach but relies on a good set of negative examples.

3) Language Restriction

Another approach is to restrict the types of clauses allowed and to prune clauses to stay within the restriction.

For example, one could limit the length of a "chain" of variables that connect to a head via variable. So $R(x,y) \land P(y, z) \rightarrow Q(x, z)$ might be allowed but $R(x,y) \land P(y, z) \land Q(z, v) \rightarrow Q(x, w)$ might not.
Golem has a special restriction called i-determinacy, which we will not cover the details of.

- It is interesting to contrast FOIL and Golem, or more generally these general-to-specific and specific-to-general approaches.

- In order to find complex rules, FOIL relies on a heuristic to guide it in the correct direction.

  for example suppose a target rule is \( R(x,v) \land R(v,w) \land R(w,y) \rightarrow Q(x,y) \)

  Such rules can be difficult for FOIL to find since the heuristic may not see that individual body atoms are useful by themselves, such as \( R(v,y) \) and \( R(w,v) \).
- In other words, FOIL can sometimes fail due to short-sightedness. The situation can be improved by increasing the amount of search (using non-greedy search), but there is a practical limit to this.

- In contrast, Golem starts with very detailed rules that capture all patterns that the examples have in common, including ones that are not important and even logically redundant. So Golem can easily capture the pattern in the above rule, but the challenge is pruning down the rest of the patterns that are not important.

- As you might guess, people have considered combined approaches that, for example, will generate specific clauses via LGBs and generalize them via a FOIL-like process.