Top-down Definite Clause Proof Procedure

Idea: search backward from a query to determine if it is a logical consequence of KB.
An answer clause is of the form:

$$yes \leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_m$$

The SLD Resolution of this answer clause on atom a_i with the clause:

$$a_i \leftarrow b_1 \wedge \ldots \wedge b_p$$

is the answer clause

$$yes \leftarrow a_1 \wedge \cdots \wedge a_{i-1} \wedge b_1 \wedge \cdots \wedge b_p \wedge a_{i+1} \wedge \cdots \wedge a_m.$$

Derivations

- An **answer** is an answer clause with $m = 0$. That is, it is the answer clause $yes \leftarrow$.
- A **derivation** of query "$?q_1 \land \ldots \land q_k$" from KB is a sequence of answer clauses $\gamma_0, \gamma_1, \ldots, \gamma_n$ such that
  - γ_0 is the answer clause $yes \leftarrow q_1 \land \ldots \land q_k$,
  - γ_i is obtained by resolving γ_{i-1} with a clause in KB, and
  - γ_n is an answer.

Top-down definite clause interpreter

To solve the query $?q_1 \wedge \ldots \wedge q_k$:

```
$ac$ := "$yes \leftarrow q_1 \wedge \ldots \wedge q_k$"
**repeat**
    **select** atom $a_i$ from the body of $ac$;
    **choose** clause $C$ from $KB$ with $a_i$ as head;
    replace $a_i$ in the body of $ac$ by the body of $C$
**until** $ac$ is an answer.
```

Nondeterministic Choice

Algorithms may be *non-deterministic:*
there are choices in the program that are left unspecified.

- Don't-care nondeterminism If one selection doesn't lead to a solution, there is no point trying other alternatives. **(select)**
  Example: resource allocation, where a number of requests occur for a limited number of resources, and a scheduling algorithm has to select who gets which resource at each time.
  Correctness is not affected by the selection, but efficiency and termination may be.
- Don't-know nondeterminism If one choice doesn't lead to a solution, other choices may. **(choose)**
  An *oracle*: it specifies, at each point, which choice will lead to a solution. Because our agent does not have such an oracle, it has to search through the space of alternate choices.

NP problems

- Don't-know non-determinism plays a large role in computational complexity theory.
- The class of *P* problems contains the problems solvable with time complexity polynomial in the size of the problem.
- The class of *NP* problems contains the problems that could be solved in polynomial time with an oracle that chooses the correct value at each time or, equivalently, if a solution is verifiable in polynomial time.
- It is widely conjectured that $P \neq NP$, which would mean that no such oracle can exist.
- One great result of complexity theory is that the hardest problems in the *NP* class of problems are all equally complex; if one can be solved in polynomial time, they all can. These problems are *NP*-complete. A problem is *NP*-hard if it is at least as hard as an *NP*-complete problem.

Search and non-determinism

- We consistently use the term **select** for *don't-care* non-determinism and **choose** for *don't-know* non-determinism.
- In a non-deterministic procedure, we assume that an oracle makes an appropriate choice at each time. Thus, a choose statement will result in a choice that will led to success, or will fail if there are no such choices.
- A non-deterministic procedure may have multiple answers, where there are multiple choices that succeed, and will fail if there are no applicable choices.
- The oracle is implemented by search.

Top-down definite clause interpreter

To solve the query $?q_1 \wedge \ldots \wedge q_k$:

```
ac := "yes ← $q_1 \wedge \ldots \wedge q_k$"
**repeat**
        **select** atom $a_i$ from the body of $ac$;
        **choose** clause $C$ from $KB$ with $a_i$ as head;
        replace $a_i$ in the body of $ac$ by the body of $C$
**until** $ac$ is an answer.
```

The best selection strategy is to select the atom that is most likely to fail.

Example: successful derivation

$a \leftarrow b \wedge c.$	$a \leftarrow e \wedge f.$	$b \leftarrow f \wedge k.$
$c \leftarrow e.$	$d \leftarrow k.$	$e.$
$f \leftarrow j \wedge e.$	$f \leftarrow c.$	$j \leftarrow c.$

Query: $?a$

$\gamma_0 : \quad yes \leftarrow a$

$\gamma_1 : \quad yes \leftarrow e \wedge f$

$\gamma_2 : \quad yes \leftarrow f$

$\gamma_3 : \quad yes \leftarrow c$

$\gamma_4 : \quad yes \leftarrow e$

$\gamma_5 : \quad yes \leftarrow$

Same example: failing derivation

$a \leftarrow b \wedge c.$	$a \leftarrow e \wedge f.$	$b \leftarrow f \wedge k.$
$c \leftarrow e.$	$d \leftarrow k.$	$e.$
$f \leftarrow j \wedge e.$	$f \leftarrow c.$	$j \leftarrow c.$

Query: $?a$

$\gamma_0 : \quad yes \leftarrow a$

$\gamma_1 : \quad yes \leftarrow b \wedge c$

$\gamma_2 : \quad yes \leftarrow f \wedge k \wedge c$

$\gamma_3 : \quad yes \leftarrow c \wedge k \wedge c$

$\gamma_4 : \quad yes \leftarrow e \wedge k \wedge c$

$\gamma_5 : \quad yes \leftarrow k \wedge c$

Top-down vs bottom-up comparison

- When the top-down procedure has derived the answer, the rules used in the derivation can be used in a bottom-up proof procedure to infer the query.
- Similarly, a bottom-up proof of an atom can be used to construct a corresponding top-down derivation.
- This equivalence can be used to show the soundness and completeness of the top-down proof procedure.
- As defined, the top-down proof procedure may spend extra time re-proving the same atom multiple times, whereas the bottom-up procedure proves each atom only once. However, the bottom-up procedure proves every atom, but the top-down procedure proves only atoms that are relevant to the query.

Search

- The non-deterministic top-down algorithm together with a selection strategy induces a search graph, which is a tree.
- Each node in the search graph represents an answer clause.
- The neighbors of a node "$yes \leftarrow a_1 \wedge \ldots \wedge a_k$", where a_i is the selected atom, represent all of the possible answer clauses obtained by resolving on a_i.
- There is a neighbor for each definite clause whose head is a_i.
- The goal nodes of the search are of the form $yes \leftarrow$.

Search Graph for SLD Resolution

$$\begin{tikzcd}
 & & & yes \leftarrow a \wedge d \arrow[dll] \arrow[d] \arrow[drr] & & \\
 & yes \leftarrow b \wedge c \wedge d \arrow[dl] \arrow[d] & & yes \leftarrow g \wedge d \arrow[dl] \arrow[d] & & yes \leftarrow h \wedge d \arrow[d] \\
yes \leftarrow j \wedge c \wedge d & yes \leftarrow k \wedge c \wedge d \arrow[d] & yes \leftarrow m \wedge d & yes \leftarrow f \wedge d \arrow[dl] \arrow[dr] & & yes \leftarrow m \wedge d \\
 & yes \leftarrow m \wedge c \wedge d & yes \leftarrow m \wedge d & & yes \leftarrow p \wedge d \arrow[d] & \\
 & & & & yes \leftarrow d \arrow[dl] \arrow[dr] & \\
 & & & yes \leftarrow m & & yes \leftarrow p \arrow[d] \\
 & & & & & yes \leftarrow
\end{tikzcd}$$

$a \leftarrow b \wedge c.$	$a \leftarrow g.$
$a \leftarrow h.$	$b \leftarrow j.$
$b \leftarrow k.$	$d \leftarrow m.$
$d \leftarrow p.$	$f \leftarrow m.$
$f \leftarrow p.$	$g \leftarrow m.$
$g \leftarrow f.$	$k \leftarrow m.$
$h \leftarrow m.$	$p.$

$?a \wedge d$

Search

- For most problems, the search graph is not given statically, because this would entail anticipating every possible query. More realistically, the search graph is dynamically constructed as needed. All that is required is a way to generate the neighbors of a node.
- Selecting an atom in the answer clause defines a set of neighbors. A neighbor exists for each rule with the selected atom as its head.
- Any of the search methods introduced before can be used to search the search space.
- Because we are only interested in whether the query is a logical consequence, we just require a path to a goal node; an optimal path is not necessary.
- There is a *different* search space for each query. A different selection of which atom to resolve at each step will result in a different search space.