UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Artificial Intelligence

Knowledge Representation and Inference

LESSON 14

prof. Antonino Staiano

M.Sc. In "Machine Learning e Big Data" - University Parthenope of Naples

Knowledge Engineering in FOL

- Knowledge engineering is the process of constructing the KB
  - It consists of investigating a specific domain, identifying the relevant concepts (knowledge acquisition), and formally representing them
- This requires the interaction between
  - a domain expert (DE)
  - a knowledge engineer (KE), who is an expert in knowledge representation and inference, but usually not in the domain of interest
- A possible approach, suitable for special-purpose KBs (in predicate logic), is the following

Knowledge Engineering

1. Identify the task
   - what range of queries will the KB support?
   - what kind of facts will be available for each problem instance?
2. Knowledge acquisition
   - eliciting from the domain expert the general knowledge about the domain (e.g., the rules of chess)
3. Choice of a vocabulary
   - what concepts must be represented as objects, predicates, functions?
     - The result is the domain's ontology, which affects the complexity of the representation and the inferences that can be made
       - E.g., in the wumpus game, pits can be represented as objects or unary predicates on squares

Knowledge Engineering

4. Encoding the domain's general knowledge acquired in step 2 (this may require revising the vocabulary of step 3)
5. Encoding a specific problem instance (e.g., a specific chess game)
6. Posing queries to the inference procedure and getting answers
7. Debugging the KB, based on the results of step 6

UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Inference in Predicate Logic

Inference in Predicate Logic

- Inference algorithms are more complex than in propositional logic, due to quantifiers and functions
- Basic tools: two inference rules for sentences with quantifiers (Universal and Existential Instantiation), that derive sentences without quantifiers
- This reduces first-order inference to propositional inference, with complete but semi-decidable inference procedures:
  - algorithms exist that find a proof KB $\vdash \alpha$ in a finite number of steps for every entailed sentence KB $\models \alpha$
  - no algorithm is capable to find the proof KB $\nvdash \alpha$ in a finite number of steps for every non-entailed sentence KB $\nvDash \alpha$
- Therefore, since one does not know that a sentence is entailed until the proof is done, when a proof procedure is running one does not know whether it is about to find proof, or whether it will never find one

Inference in Predicate Logic

- Modus Ponens can be generalized to predicate logic, leading to the first-order versions of the Forward Chaining and Backward Chaining algorithms, which are complete and semi-decidable and limited to Horn clauses
- The Resolution rule can also be generalized to predicate logic, leading to the first-order version of the complete but semi-decidable resolution algorithm

Inference Rules for Quantifiers

- Let θ denote a substitution list $\{v_1/t_1, \ldots, v_n/t_n\}$, where:
  - $v_1,\ldots,v_n$ are variable names
  - $t_1, \ldots, t_n$ are terms (either constant symbols, variables, or functions recursively applied to terms)
- Let SUBST(θ, α) denote the sentence obtained by applying the substitution θ to the sentence α
  - Example:
    - SUBST({y/One}, ∀x, y Eq(S(x), S(y)) ⇒ Eq(x, y))

    produces

    ∀x Eq(S(x), S(One)) ⇒ Eq(x, One)

Inference Rules for Quantifiers

- Universal Instantiation (UI)

$$\frac{\forall \mathrm{v}\ \alpha}{\mathrm{SUBST}(\{\mathrm{v/t}\ \},\alpha)}$$

  where t can be any term without variables (ground term)

- Since a sentence ∀x α[x] states that α is true for every domain element in place of x, then one can derive that α is true for any given element t
- Example: from ∀x N(x) ⇒ N(S(x)) one can derive
  - N(Z) ⇒ N(S(Z)), for θ = {x/Z}
  - N(S(S(Z))) ⇒ N(S(S(S(Z)))), for θ = {x/S(S(Z))}
  - …

Inference Rules for Quantifiers

- Existential Instantiation (EI)

$$\frac{\exists v\ \alpha}{\text{SUBST}(\{v/t\},\alpha)}$$

  where t must be a new constant symbol that does not appear elsewhere in the KB

- A sentence $\exists v\ \alpha[v]$ states that there is some domain element satisfying a condition
- The above rule just gives a name to one such element, but that name must not belong to another element
- Example
  - $\exists x\ Crown(x) \wedge OnHead(x,John)$
    - $Crown(C1) \wedge OnHead(C1, John)$, provided that *C1* is not in the KB

Inference Rules for Quantifiers

- A more general form of Existential Instantiation must be applied when an existential quantifier appears in the scope of a universal quantifier:

$$\forall x,\ldots\ \exists y,\ldots\ \alpha[x,\ldots,y\ \ldots]$$

- For instance
  - from ∀x ∃y Loves(x,y) (Everybody loves somebody)
  - it is not correct to derive ∀x Loves(x,A) (Everybody loves A)
    - the latter sentence means that everybody loves the same person

Inference Rules for Quantifiers

- Instead of a constant symbol, a new function symbol, known as the Skolem function, must be introduced with as many arguments as universally quantified variables. Therefore, from:

$$\forall x,\ldots\ \exists y,\ldots\ \alpha[x,\ldots,y\ \ldots]$$

the correct application of EI derives:

$$\forall x,\ldots\ \alpha[x,\ldots,F_1(x),\ldots]$$

- For instance, from

$$\forall x\ \exists y\ \mathit{Loves}(x,y)$$

one can correctly derive

$$\forall x\ \mathit{Loves}(x,F(x))$$

where F maps any individual x to someone loved by x

Inference Algorithms and Quantifiers

- First-order inference algorithms usually apply Existential Instantiation as a pre-processing step
  - every existentially quantified sentence is replaced by a single sentence
- It can be proven that the resulting KB is inferentially equivalent to the original one, i.e., it is satisfiable when the original one is
  - Accordingly, the resulting KB contains only sentences without variables and sentences where all the variables are universally quantified
- Another useful pre-processing step is renaming all the variables in the KB to avoid name clashes between variables used in different sentences
  - For instance, the variables in $\forall x\ P(x)$ and $\forall x\ Q(x)$ are not related to each other, and renaming any of them (say, $\forall y\ Q(y)$) produces an equivalent sentence

Unification

- A widely used tool in first-order inference algorithms is unification, the process of finding a substitution (if any) that makes two sentences (where at least one contains variables) identical
- For instance, ∀x,y *Knows*(x,y) and ∀z *Knows*(*John*,z) can be unified by different substitutions
  - Assuming that Bill is one of the constant symbols, two possible unifiers are:
    - {x/John,y/Bill,z/Bill}
    - {x/John,y/z}
- Among all possible unifiers, the one of interest for first-order inference algorithms is the most general unifier, i.e., the one that places the fewest restrictions on the values of the variables
  - The only constraint is that every occurrence of a given variable can be replaced by the same term
- In the above example, the most general unifier is {x/John,y/z}, as it does not restrict the value of y and z

Unification Example

- Consider the sentence $\forall x$ *Knows(John,x)* (*John knows everyone*). Assume that the KB also contains the following sentences (note that different variables names are used in different sentences):
  1. *Knows(John,Jane)*
  2. $\forall y$ *Knows(y,Bill)*
  3. $\forall z$ *Knows(z,Mother(z))*
  4. *Knows(Elizabeth,Bill)*
- The most general unifier with *Knows(John,x)* is:
  1. {*x/Jane*}
  2. {*y/John,x/Bill*}
  3. {*z/John,x/Mother(John)*}
  4. no unifier exists, as the constant symbols *John* and *Elizabeth* in the first argument are different

First-order Inference Example

- Consider a domain made up of two individuals denoted with the constant symbols *John* and *Richard*, and the following KB:
  1) $\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$
  2) $\forall y\ Greedy(y)$
  3) $King\ (John)$
  4) $Brother\ (Richard\ ,\ John)$
- Intuitively, this entails *Evil(John)*, i.e., *KB |= Evil(John)*
- The corresponding inference KB $\vdash$ Evil(John) can be obtained by using the above inference rules, as shown in the following

First-order Inference Example

- Applying Universal Instantiation to (1) produces:
  5) *King(John) ∧ Greedy(John) ⇒ Evil(John), with {x/John}*
  6) *King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard), with {x/Richard}*
- Applying Universal Instantiation to (2) produces:
  7) *Greedy(John), with {y/John}*
  8) *Greedy(Richard), with {y/Richard}*
- Applying And Introduction to (3) and (7) produces:
  9) *King(John)∧Greedy(John)*
- Applying Modus Ponens to (5) and (9) produces:
  10) *Evil(John)*

Generalized Modus Ponens

- All but the last inference steps in the above example can be seen as pre-processing steps whose aim is to "prepare" the application of Modus Ponens
- Moreover, some of these steps (Universal Instantiation using the symbol Richard) are clearly useless to derive the consequent of implication (1)
  - i.e., Evil(John)
- Indeed, the above steps can be combined into a single first-order inference rule, Generalized Modus Ponens (GMP):
  - given atomic sentences (non-negated predicates) p_i , p'_i , i = 1,...,n, and q, and a substitution θ such that Subst(θ,p_i) = Subst(θ,p_i') for all i:

$$\frac{(p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q), \quad p'_1, p'_2 \ldots, p'_n}{\textsc{Subst}(\theta, q)}$$

Generalized Modus Ponens

- In the previous example, GMP allows *Evil(John)* to be derived in a single step
  - and avoids unnecessary applications of inference rules like Universal Instantiation to sentences (1) and (2) with *{x/Richard}* or *{y/Richard}*
- GMP can be applied to sentences (1), (2), and (3), with θ = *{x/John, y/John}*
  1) $\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$
  2) $\forall y\ Greedy(y)$
  3) $King\ (John)$
  - this immediately derives *Evil(John)*
- The key advantage of GMP inference rule over propositionalization is that it makes only those substitutions that are required to allow inferences to proceed

Horn Clauses in Predicate Logic

- GMP allows the Forward Chaining (FC) and Backward Chaining (BC) inference algorithms to predicate logic
  - This in turn requires the concept of Horn clause
- A Horn clause in predicate logic is an implication $\alpha \Rightarrow \beta$ in which:
  - α is a conjunction of non-negated predicates
  - β is a single non-negated predicate
  - all variables (if any) are universally quantified, and the quantifier appears at the beginning of the sentence
- Example: $\forall x\ (P(x) \wedge Q(x)) \Rightarrow R(x)$
- Also single predicates (possibly negated) are Horn clauses:
  - $P(t_1,\ldots,t_n)$ equivalent to $(\text{True} \Rightarrow P(t_1,\ldots,t_n))$
  - $\neg P(t_1, \ldots, t_n)$ equivalent to $(P(t_1, \ldots, t_n) \Rightarrow \text{False})$

Forward Chaining in Predicate Logic

- Forward Chaining (FC) consists of repeatedly applying GMP in all possible ways, adding to the initial KB all newly derived atomic sentences until no new sentence can be derived
- FC is normally triggered by the addition of new sentences into the KB, to derive all their consequences
- For instance, it can be used in the Wumpus game when new percepts are added to the KB, after each agent's move

Forward Chaining in Predicate Logic

- A simple (but inefficient) implementation of FC

```
function Forward-chaining (KB)

local variable: new
repeat
    new ← ∅ (the empty set)
    for each sentence s = (p1 ∧...∧ pn ⇒q)in KB do
        for each θ such that Subst(θ, p1 ∧ . . . ∧ pn) =
            Subst(θ,p1' ∧...∧ pn')for some p1',...,pn' ∈ KB do
            q' ← Subst(θ, q)
            if q' ∉ KB and q' ∉ new then add q' to new
    add new to KB
until new is empty
return KB
```

Example

- *The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American*
  - We aim to derive Criminal(West)
- We need to represent these facts as FOL Horn clauses
  - . . . it is a crime for an American to sell weapons to hostile nations"
    - American(x)∧Weapon(y)∧Sells(x,y, z)∧Hostile(z) ⇒ Criminal(x)
  - "Nono . . . has some missiles."
    - The sentence ∃x Owns(Nono,x) ∧Missile(x) is transformed into two definite clauses by Existential Instantiation, introducing a new constant M
    - Owns(Nono,M)
    - Missile(M)
  - "All of its missiles were sold to it by Colonel West"
    - Missile(x)∧Owns(Nono,x) ⇒ Sells(West,x,Nono)

Example

- We will also need to know that missiles are weapons:
  - Missile(x) ⇒Weapon(x)
- and we must know that an enemy of America counts as “hostile”
  - Enemy(x,America) ⇒ Hostile(x)
- “West, who is American . . .”
  - American(West)
- “The country Nono, an enemy of America . . .”
  - Enemy(Nono, America)

Forward Chaining Example

- Summarizing (the universal quantifiers are not shown to keep the notation uncluttered)

$$
\begin{aligned}
&(American(x) \wedge Hostile(y) \wedge \\
&\quad Weapon(z) \wedge Sells(x, y, z)) \Rightarrow Criminal(x) && (1)\\
&(Missile(x) \wedge Owns(Nono, x)) \Rightarrow Sells(West, Nono, x) && (2)\\
&Enemy(x, America) \Rightarrow Hostile(x) && (3)\\
&Missile(x) \Rightarrow Weapon(x) && (4)\\
&American(West) && (5)\\
&Enemy(Nono, America) && (6)\\
&Owns(Nono, M) && (7)\\
&Missile(M) && (8)
\end{aligned}
$$

Forward Chaining Example

- The FC algorithm carries out two repeat-until loops on the above KB
- No new sentences can be derived after the second loop
- First iteration
  - GMP to (2), (7) and (8), with $\{x/M\}$:
    (9) $Sells(West, Nono, M)$
  - GMP to (3) and (6), with $\{x/Nono, y/America\}$:
    (10) $Hostile(Nono)$
  - GMP to (4) and (8), with $\{x/M\}$:
    (11) $Weapon(M)$
- Second iteration
  - GMP to (1), (5), (10), (11) and (9), with $\{x/West, y/Nono, z/M\}$:
    (12) $Criminal(West)$

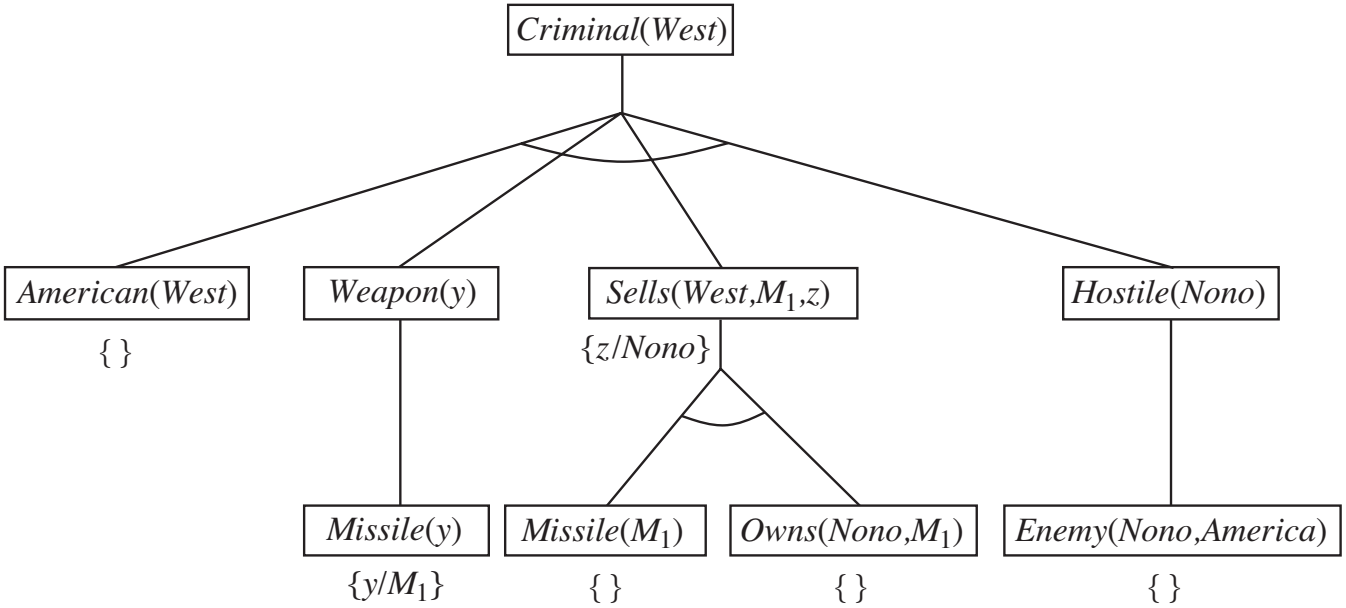
$$(American(x) \land Hostile(y) \land Weapon(z) \land Sells(x, y, z)) \Rightarrow Criminal(x) \quad (1)$$
$$(Missile(x) \land Owns(Nono, x)) \Rightarrow Sells(West, Nono, x) \quad (2)$$
$$Enemy(x, America) \Rightarrow Hostile(x) \quad (3)$$
$$Missile(x) \Rightarrow Weapon(x) \quad (4)$$
$$American(West) \quad (5)$$
$$Enemy(Nono, America) \quad (6)$$
$$Owns(Nono, M) \quad (7)$$
$$Missile(M) \quad (8)$$

Backward Chaining in Predicate Logic

- The first-order Backward Chaining (BC) algorithm starts from a sentence (query) to be proven and recursively applies GMP backward
- Note that every substitution that is made to unify an atomic sentence with the consequent of an implication must be propagated back to every antecedent
- If the consequent of an implication unifies with more than one atomic sentence, at least one unification must allow the consequent to be proven

Backward Chaining Example

- A proof by BC can be represented as an And-Or graph
- The following graph (which should be read depth-first, left to right) shows the proof of the query *Criminal(West)* using the previous sentences (1)–(8) as the KB



The Resolution Algorithm

- Completeness theorem for predicate logic (Gödel, 1930)
  - *For every first-order sentence α entailed by a given KB* (KB $\models \alpha$) *there exists some inference algorithm that derives* α (KB $\vdash \alpha$) *in a finite number of steps*
  - The opposite does not hold
    - *Predicate logic is semi-decidable*
- A complete inference algorithm for predicate logic: Resolution (1965) based on
  - Converting sentences into Conjunctive Normal Form
  - Using only Resolution inference rule
  - Proof by contradiction
    - to prove *KB* $\models \alpha$, prove that *KB* $\wedge \neg \alpha$ is unsatisfiable (contradictory)
  - Refutation-completeness
    - if *KB* $\wedge \neg \alpha$ is unsatisfiable, then resolution derives a contradiction in a finite number of steps

Applications of FC, BC and Resolution

- FC
  - Encoding condition-action rules to recommend actions, based on a data-driven approach
    - Production systems (production: condition-action rules)
    - Expert systems
- BC
  - Logic programming languages (e.g. Prolog), used for
    - Rapid prototyping
    - Symbol processing applications (compilers, NL parsers, …)
- Resolution
  - Main application -> theorem provers, used for
    - Assisting mathematicians
    - Proof checking
    - Verification and synthesis of hardware and software