First-order logic

• We saw how propositional logic can create intelligent behavior

• But propositional logic is a poor representation for complex environments

• First-order logic is a more expressive and powerful representation

Causal Rules

• Some hidden property causes percepts to be generated
  – These are predictions of perceptions you expect to have in the future given current conditions
    • A pit causes adjacent squares to be breezy
      \[ \forall x \text{ Pit}(x) \Rightarrow \exists y \text{ Adjacent}(x,y) \Rightarrow \text{Breezy}(y) \]
    • If all squares adjacent to a square a pitless, it will not be breezy
      \[ \forall x (\forall y \text{ Adjacent}(x,y) \Rightarrow \neg \text{Pit}(y)) \Rightarrow \neg \text{Breezy}(x) \]

Diagnostic Rules

• Rules leading from observed effects to hidden causes
  – After you’ve observed something, this rule offers an explanation
  – These rules explain what happened in the past
    • Breezy implies pits
      \[ \forall x \text{ Breezy}(x) \Rightarrow \exists y \text{ Adjacent}(x,y) \land \text{Pit}(y) \]
    • Not breezy implies no pits
      \[ \forall x \neg \text{Breezy}(x) \Rightarrow \neg \exists y \text{ Adjacent}(x,y) \land \text{Pit}(y) \]
    • Combining
      \[ \forall x \text{ Breezy}(x) \iff \exists y \text{ Adjacent}(x,y) \land \text{Pit}(y) \]
Conclusion
• If the axioms correctly and completely describe the way the world works and the way percepts are produced,
• then any complete logical inference procedure will infer the strongest possible description of the world state given the available percepts
• The agent designer can focus on getting the knowledge right without worrying about the processes of deduction

Discussion of models
• Let’s think about how we use models every day
  – Daily stock prices
  – Seasonal stock prices
  – Seasonal temperatures
  – Annual temperatures

Knowledge Engineering
– Understand a particular domain
  • How does stock trading work
– Learn what concepts are important in the domain (features)
  • Buyer confidence, strength of the dollar, company earnings, interest rate
– Create a formal representation of the objects and relations in the domain
  • For all stocks (price = low ^ confidence = high) => profitability = high

Identify the task
• What is the range of inputs and outputs
  – Will the stock trading system have to answer questions about the weather?
    • Perhaps if you’re buying wheat futures
    • Must the agent store daily temperatures or can it use another agent?

Assemble the relevant knowledge
– You know what information is relevant
– How can you accumulate the information?
  • Not formal description of knowledge at this point
  • Just principled understanding of where information resides

Formalize the knowledge
• Decide on vocabulary of predicates, functions, and constants
  – Beginning to map domain into a programmatic structure
  – You’re selecting an ontology
    • A particular theory of how the domain can be simplified and represented at its basic elements
    • Mistakes here cause big problems
Encode general knowledge
- Write down axioms for all vocabulary terms
  • Define the meaning of terms
- Errors will be discovered and knowledge assembly and formalization steps repeated

Map to this particular instance
- Encode a description of the specific problem instance
  - Should be an easy step
  - Write simple atomic sentences
    • Derived from sensors/percepts
    • Derived from external data

Use the knowledge base
• Pose queries and get answers
  - Use inference procedure
  - Derive new facts

Debug the knowledge base
• There will most likely be bugs
  - If inference engine works bugs will be in knowledge base
    • Missing axioms
    • Axioms that are too weak
    • Conflicting axioms

Enough talk, let’s get to the meat
• Chapter 9
  • Inference in First-Order Logic
    – We want to use inference to answer any answerable question stated in first-order logic

Propositional Inference
• We already know how to perform inference in propositional logic
  – Transform first-order logic to propositional logic
  – First-order logic makes powerful use of variables
    • Universal quantification (for all $x$)
    • Existential quantification (there exists an $x$)
Converting universal quantifiers

- Universal Instantiation
- Example: \( \forall x \ King(x) \land Greedy(x) \Rightarrow Evil(x) \) after substitution \( \{x/John\}, \{x/Richard\}, \{x/Father(John)\} \) becomes
  \[
  King(John) \land Greedy(John) \Rightarrow Evil(John),
  King(Richard) \land Greedy(Richard) \Rightarrow Evil(Richard),
  King(Father(John)) \land Greedy(Father(John)) \Rightarrow Evil(Father(John))
  \]
  We’ve replaced the variable with all possible ground terms (terms without variables)

Converting existential quantifiers

- Existential Instantiation
- Example: \( \exists x \ Crown(x) \land OnHead(x, John) \)
  - There is some thing that is a crown and is on John’s head...
  - Let’s call it \( C_1 \)
  becomes: \( Crown(C_1) \land OnHead(C_1, John) \)
  - You can replace the variable with a constant symbol that does not appear elsewhere in the knowledge base
  - The constant symbol is a Skolem constant

Existential Instantiation

- Only perform substitution once
  - There exists an x s.t. Kill (x, Victim)
    - Someone killed the victim
    - Maybe more than once person killed the victim
    - Existential quantifier says at least one person was killer
  - Replacement is
    - Kill (Murderer, Victim)

Complete reduction

- Convert existentially quantified sentences
  - Creates one instantiation
- Convert universally quantified sentences
  - Creates all possible instantiations
- Use propositional logic to resolve
  - Every first-order knowledge base and query can be propositionalized in such a way that entailment is preserved

Trouble ahead!

- Universal quantification with functions:
  \[
  King(John) \land Greedy(John) \Rightarrow Evil(John),
  King(Richard) \land Greedy(Richard) \Rightarrow Evil(Richard),
  King(Father(John)) \land Greedy(Father(John)) \Rightarrow Evil(Father(John))
  \]
  - What about \( (Father(Father(Father(John)))) \)?
    - Isn’t it possible to have infinite number of substitutions?
    - How well will the propositional algorithms work with infinite number of sentences?

A theorem of completeness

- If
  - a sentence is entailed by the original, first-order knowledge base
- Then
  - there is a proof involving just a finite subset of the propositional knowledge base
  - We want to find that finite subset
    - First try proving the sentence with constant symbols
    - Then add all terms of depth 1: Father (Richard)
    - Then add all terms of depth 2: Father (Father (Richard))
    - ...
Still more trouble

- Completeness says that if statement is true in first-order logic, it will also be true in propositions
  - But what happens if you've been waiting for your proposition-based algorithm to return an answer and it has been a while?
    • Is the statement not true?
    • Is the statement just requiring lots of substitutions?
- You don’t know!

The Halting Problem

- Alan Turing and Alonzo Church proved
  - You can write an algorithm that says yes to every entailed sentence but
  - You no algorithm exists that says no to every nonentailed sentence
- So if your entailment-checking algorithm hasn’t returned “yes” yet, you cannot know if that’s because the sentence is not entailed.
- Entailment for first-order logic is semi-decidable

Adapting Modus Ponens

- Did you notice how inefficient previous method was?
  - Instantiate universal quantifiers by performing lots of substitutions until (hopefully quickly) a proof was found
  - Why bother substituting Richard for x when you and I know it won’t lead to a proof?
  - Clearly, John is the right substitution for x.

Modus Ponens for propositional logic

\[
\alpha \Rightarrow \beta, \quad \alpha \\
\frac{}{\beta}
\]

Generalized Modus Ponens

- For atomic sentences \( p, p', \theta \) where there is a substitution \( \theta \) such that \( \text{Subst}(q, p') = \text{Subst}(q, p) \):
  \[
  p_1, p_1', \ldots, p_n', (p_1 \land p_2 \land \ldots \land p_n \Rightarrow q) \\
  \text{SUBST}(\theta, q)
  \]

\[
\forall x \text{ King}(x) \land \text{ Greedy}(x) \Rightarrow \text{ Evil}(x) \\
\text{King}(John) \land \text{ Greedy}(John) \Rightarrow \text{ Evil}(John)
\]

Generalized Modus Ponens

- This is a lifted version
  - It raises Modus Ponens to first-order logic
  - We want to find lifted versions of forward chaining, backward chaining, and resolution algorithms
- Lifted versions make only those substitutions that are required to allow particular inferences to proceed
Unification

- Generalized Modus Ponens requires finding good substitutions
  - Logical expressions must look identical
  - Other lifted inference rules require this as well
- Unification is the process of finding substitutions

Unification

- Unify takes two sentences and returns a unifier if one exists
  \[ \text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) \]
- Examples to answer the query, \text{Knows} (John, x):
  - Whom does John know?

\[
\begin{align*}
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(\text{John}, \text{Jane})) &= \{x/\text{Jane}\} \\
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(\text{Bill}, \text{Bill})) &= \{x/\text{Bill}, y/\text{John}\} \\
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Mother}(y))) &= \{y/\text{John}, x/\text{Mother}(\text{John})\} \\
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(z, \text{Elizabeth})) &= \text{fail}.
\end{align*}
\]

Unification

- Consider the last sentence:
  \[ \text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) = \text{fail} \]
  - This fails because \( x \) cannot take on two values
  - But “Everyone knows Elizabeth” and it should not fail
  - Must standardize apart one of the two sentences to eliminate reuse of variable

\[ \text{UNIFY}(\text{Knows}(\text{John}, z), \text{Knows}(z_1, \text{Elizabeth})) = \{x/\text{Elizabeth}, z_1/\text{John}\} \]

Unification

- Multiple unifiers are possible:
  \[ \text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, z)) \]
  - \( \{y/\text{John}, x/z\} \quad \text{or} \quad \{y/\text{John}, x/\text{John}\} \)
  Which is better, \text{Knows} (John, z) or \text{Knows} (John, John)?
  - Second could be obtained from first with extra subs
  - First unifier is more general than second because it places fewer restrictions on the values of the variables
  - There is a single most general unifier for every unifiable pair of expressions

Storage and Retrieval

- Remember Ask() and Tell() from propositions?
  - Replace with Store(s) and Fetch(q)
    - Store puts a sentence, \( s \), into the KB
    - Fetch returns all unifiers such that query \( q \) unifies with some sentence in the KB
    - An example is when we ask \text{Knows} (John, x) and returns

\[
\begin{align*}
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(\text{John}, \text{Jane})) &= \{x/\text{Jane}\} \\
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(\text{Bill}, \text{Bill})) &= \{x/\text{Bill}, y/\text{John}\} \\
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Mother}(y))) &= \{y/\text{John}, x/\text{Mother}(\text{John})\} \\
\text{UNIFY}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) &= \text{fail}.
\end{align*}
\]

Fetch()

- Simple
  - Store all facts in KB as one long list
  - For each fetch(q), call Unify (q, s) for every sentence s in list
    - Inefficient because we’re performing so many unifies…
- Complex
  - Only attempt unifications that have some chance of succeeding
Predicate Indexing

- Index the facts in the KB
  - Example: unify Knows (John, x) with Brother (Richard, John)
  - Predicate indexing puts all the Knows facts in one bucket and all the Brother facts in another
    - Might not be a win if there are lots of clauses for a particular predicate symbol
    - Consider how many people Know one another
    - Instead index by predicate and first argument
    - Clauses may be stored in multiple buckets

Subsumption lattice

- How to construct indices for all possible queries that unify with it
  - Example: Employs (AIMA.org, Richard)
    - Employs(AIMA.org, Richard)  Does AIMA.org employ Richard?
    - Employs(x, Richard)  Who employs Richard?
    - Employs(AIMA.org, y)  Whom does AIMA.org employ?
    - Employs(x, y)  Who employs whom?

Subsumption lattice

- Each node reflects making one substitution
- The “highest” common descendent of any two nodes is the result of applying the most general unifier
- Predicate with n arguments will create a lattice with O(2^n) nodes
- Benefits of indexing may be outweighed by cost of storing and maintaining indices