Announcements

- Read Sections 10.2-10.3, and Chapter 11 in *AI: A Modern Approach* for Monday
- Homework #2 is due on Monday
  - Handin directories are already set up, in case you’re ready to turn it in
- Homework #3 will go out on Monday (but it won’t be due until after the midterm)

Logic Programming

- Computation as inference on logical KBs
  - Ordinary Programming
    1. Identify problem
    2. Assemble information
    3. Plan out your algorithms
    4. Program the solution
    5. Encode problem as data
    6. Run program on data
    7. Debug errors
  - Logic Programming
    1. Identify problem
    2. Assemble information
    3. Go have a beer!
    4. Encode information in the KB
    5. Encode problem as logical facts
    6. Ask queries
    7. Find/correct false facts

Logic Programming

- Logic programs are also called expert systems
  - Problem domain experts sit down and encode lots of information into the KB
  - System then reasons using that “expert” knowledge
  - Used for medical diagnosis, QA systems, etc.
  - KB must be in datalog format: FOL with no functions
- Fall loosely under the “think rationally” quadrant of AI research

Prolog

- Prolog is probably the most common logic programming language
- A program is:
  - A set of logic sentences in HNF (definite clauses)
    • Called the database (DB, basically the KB)
    • Ordered by programmer (top to bottom)
  - Executed by specifying a query to be proved
    • Backward-chaining with GMP
    • Uses DFS on the ordered facts and rules
    • Searches until a solution is found (or times out)
  - Can find multiple solutions

Prolog Execution

To Solve a Goal (i.e. answer a query)

- Try to unify:
  - First with each of the ground facts
  - When all facts fail, then with each of the consequents of the rules in the order in which they occur in DB
- Successful unification with a fact:
  - Solved, pop goal from stack
- Successful unification with a rule:
  - Solve the sub-goals in DFS manner (i.e. recursively attempt to solve each of the rule’s premises)
Prolog Execution

Backtracking During DFS
- While solving a rule:
  - If antecedent (premise) fails to be proved true
  - Then try to re-prove it using different facts or rules
- When a rule fails:
  - If an antecedent can’t be solved at all, the rule fails
  - Go on to the next rule in the program and try again
  (try to unify current goal with a different consequent)

Efficient implementation
- Unification use “open coding”
- Retrieval and matching of clauses by direct linking
- Sophisticated memory management
- Uses a closed world assumption
  - Negation as failure
  - E.g.
    \[
    \neg \neg \neg \neg \text{dead}(X) \Rightarrow \text{alive}(X)
    \]
    \[
    \neg \neg \text{alive}(\text{elvis}) \text{ succeeds if dead(elvis) fails}
    \]
Another Prolog Example

Let’s consider a simple KB that expresses facts about a certain family:

\[
\begin{align*}
\text{father}(\text{tom}, \text{dick}) & , \text{mother}(\text{tom}, \text{judy}) .
\text{father}(\text{dick}, \text{harry}) & , \text{mother}(\text{dick}, \text{mary}) .
\text{father}(\text{jane}, \text{harry}) & , \text{mother}(\text{jane}, \text{mary}) .
\end{align*}
\]

Now let’s also think about creating some FOL rules for defining family relations:

- Parent?
  \[\text{parent}(X, F) :- \text{mother}(X, F) .\]
  \[\text{parent}(X, F) :- \text{father}(X, F) .\]

- Grandmother?
  \[\text{grandy}(X, G) :- \text{parent}(X, Y) , \text{mother}(Y, G) .\]

How about this:

\[\text{parent}(X, F) :- \text{mother}(X, F) .\]
\[\text{parent}(X, F) :- \text{father}(X, F) .\]

Now let’s consider the relation sibling?

- Two people are siblings if they have the same mother and the same father (ignoring half-siblings, step-siblings, etc.)

How about this:

\[\text{sibling}(X, Y) :- \text{mother}(X, M) , \text{mother}(Y, M) , \text{father}(X, M) , \text{father}(Y, M) , X \neq Y .\]

Let’s run this and see what happens!

• Oop! Need to make sure \(X \neq Y\)!

\[\text{sibling2}(X, Y) :- \text{mother}(X, M) , \text{mother}(Y, M) , \text{father}(X, M) , \text{father}(Y, M) , X \neq Y .\]

More Prolog Syntax

Prolog has built-in operators (predicates) for mathematical functions and equalities:

\[
\begin{align*}
X &= 2 \times (Y + 1) .
D &< 20 .
1 \leq 2 X = Y .
X \neq Y .
\end{align*}
\]

The major data structure for Prolog is the list

- \([\ ]\) denotes an empty list
- \([H | T]\) denotes a list with a head (H) and tail (T)
  * The head is the first element of the list
  * The tail is the entire sublist after it
  * e.g. for the list \([a,b,c,d]\) . \(H=a\) and \(T=[b,c,d]\)

List Processing in Prolog

Suppose we want to define an “append” operator for lists... that is to take two lists \(L_1\) and \(L_2\), and merges their elements together into a new list \(L_3\)

- Usually this is done with a function
  \[\text{e.g.} \ L_3 \leftarrow \text{append}(L_1, L_2) .\]
  - But prolog programs are datalog: no functions allowed!
  - Create make-shift functions by defining predicates with the return value included as a parameter
  \[\text{e.g.} \ \text{append}(L_1, L_2, L_3) .\]

How about defining a simple predicate that takes the first two \(L_1\) and \(L_2\), and returns a new list \([L_1|L_2]\)?

- e.g. \(\text{append}(L_1, L_2, [L_1|L_2]) .\)
  - Nope! Let’s try again…

List Processing in Prolog

What we need to do is take one list and recursively add one element at a time from the other list, until we’ve added them all.

Let’s assume that we start with \(L_2\) and want to add the elements from \(L_1\) one at a time to the front

- Makes things easier: with \(H[T]\), \(H\) is the front element

What is our base case?

\* \(\text{append}(\{\}, L_2, L_2) .\)

- Now how do we deal with the recursive aspect?
  \* \(\text{append}(\{H[T]\}, L_2, [L_1|L_3]) :- \text{append}(L_1, L_2, L_3) .\)

Now we can ask the queries:

\* \(\text{append}([1,2,3], [a,b,c]) , [1,2,3,a,b,c]) .\)
  * Result: \(\text{yes}\)

\* \(\text{append}(1,2,3, [a,b,c]) , X ) .\)
  * Result: \(X = [1,2,3, a, b, c] .\)

Recall that, since prolog uses BC, we can try to find any single solution, or find all solutions

- After each result, type “;” to view another
Partitioning Lists

- Another useful application might be how to recursively sort prolog lists
- Most sorting algorithms utilize some partitioning method, where the list \( L \) is split into two sublists \( L_1 \) and \( L_2 \) based on a particular element \( E \)
  - e.g. splitting list \([1,5,3,9,7,4,1]\) on element \([5]\) would yield the lists \([1,3,4,1]\) and \([5,9,7]\)
- This would be a useful method to define first \( \text{partition}(E, L, L_1, L_2) \).

Partitioning Lists

- If \( H < E \), then we want to add \( H \) to the first list \( L_1 \):
  \[
  \text{partition}(E, [H|T], [H|T_1], L_2) : - \ 
  H < E, \text{partition}(E, T, T_1, L_2).
  \]
- However, if \( H \geq E \) then we'll add it to the second list \( L_2 \):
  \[
  \text{partition}(E, [H|T], L_1, [H|T_2]) : - \ 
  H \geq E, \text{partition}(E, T, L_1, T_2).
  \]
- These predicates, together with the base case, will partition all the list items less than \( E \) in the first list, and all greater or equal in the second list

Sorting in Prolog

- Now that we know how to partition one list into two, and also how to append two lists together, we have all the tools we need to sort a list!
- Let's consider insertion sort:
  - Walk through each position of the list
  - For each position, insert the list item \( i \) that belongs in that position, relative to other items in the list
  - Recursively, we can achieve the same effect by walking through each \( i \), partitioning a pre-sorted list on \( i \), and then appending the partitions on either side

Parsing with Prolog

- A lot of early natural language processing (NLP) research was historically done using logic systems, because HNF rules are analogous to grammar productions
  - e.g. A simple English grammar: \( S \rightarrow NP \ VP (NP) \)
  - \( S \) means “sentence,” \( NP \) means “noun phrase,” and \( VP \) means “verb phrase”
  - In prolog:
    \[
    s(Input) :- np(Input, MidL), vp(MidD, [1]).
    \]
    \[
    s(Input) :- np(Input, MidL), np(MidL), np(MidD), np(MidD, [1]).
    \]
- Once we add definitions for \( np \) and \( vp \) (and ultimately \( noun, verb, prep, det \), etc.), we will have a full-blown deterministic English parser!
Ordering Prolog Rules

- The rules in a Prolog program are searched depth-first, exploring the potential rules from top down.
- Imagine that we are designing a knowledge-based reflex agent that has multiple rules which it can unify:
  - We want to make more specific rules toward the top, and more general rules toward the bottom.
  - For recursive “functions,” that means making sure the base case comes before the recursive case(s).

Other Logic Systems

- Production Systems
  - Proposed by E.L. Post in 1943.
  - Equivalent in computational power to a Turing machine.
  - Rules are unordered, unlike Prolog.
  - Have been developed for a wide variety of problems, ranging from algebra word problems, mathematical and logical proofs, physics problems, and games.
  - Newell and Simon (1960s) used production systems to define model human cognition:
    - Production rules represent problem-solving skills stored in a person’s long-term memory.
  - Many other groups have tried to develop similar models of human cognition using production systems.
  - Harder to do inference than BC in Prolog.

Other Logic Systems

- Semantic Networks
  - Used widely in computational linguistics.
  - Developed to aid in machine translation and natural language understanding (e.g., WordNet is a famous SN).
  - Represent knowledge in a hierarchy of semantic classes to be able to deduce and disambiguate meaning:
    - e.g., “Jane looked for her keys. She needed milk.”
    - The query: Why was Jane looking for her keys?

Summary

- Logic programs are agent programs that use facts and rules in a KB to answer questions about a particular domain.
- Such programs arrive at conclusions (or decide on actions) in a logical way.
- Prolog is one of the most common logic programming languages:
  - Uses first-order definite clauses to encode the KB.
  - Searches for proofs recursively with BC and GMP.
  - Can answer yes/no queries, or find bindings.