More Logic

CS / Philo 372
week 5

Quantification

- $\exists x P(x)$
  - Existential Quantification
  - Read: there exists an x such that P(x) is true

- $\forall x P(x)$
  - Universal Quantification
  - Read: for all x P(x) is true

Quantification

- Suppose this set of facts
  - english(george).
    english(henry).
    english(william).
    english(richard).
    english(john).
    french(henri).
- What do each of these sentences mean?
  - forAll x english(x) => king(x).
  - thereExists x english(x) => evil(x).
  - thereExists x thereExists y english(x) & french(y) => fight(x,y)
  - thereExists x forAll y english(x) & french(y) => fight(x,y).

Quantification Equivalence

- forall x not(P(x)) === not(exists x P(x))
- not(forall x P(x)) === exists x not(P(x))
- forall x P(x) === not(exists x not(P(x)))
- exists x P(x) === not(forall x not(P(x)))

Unification

- The process of finding facts that can consistently satisfy the constraints specified in a sentence
  - Recall from last week
  - edge2(X,Y,Z) :- edge(X,Z) , edge(Z, Y).
  - When asked edge2(a,Mid,f) should get
    - Mid=d
    - Mid=c
  - When asked edge2(Bgn,d,Nd) you get
    - Bgn=a, Nd=f
    - Bgn=a, Nd=g
    - Bgn=a,Nd=e

Forward Chaining Production Systems

- Basic concept
  - Start at the facts and use rules to derive new facts
  - Keep on deriving new facts until one of the new facts is the thing that you want to prove
    - hassecrets(X) & citizenof(X,Z) & paidby(X,Y) & enemyof(Y,Z) => traitor(X,Z).
      spy(x) => hassecrets(x)
      cia(x) => hassecrets(X) & citizenof(X, usa)
      ....
    - cia(bill).
      paidby(bill, cuba).
      ....

Forward Chaining

- Major problem of FC is time
  - Usual first trick is to have few facts
    - For instance R1 / XCON had more than 10,000 rules in its library but would often start with ~10 facts
    - The set of facts (both starting and derived) is called the "working memory"
- Conjunct Ordering Problem
  - Find the ordering of conjuncts in the premise of a rule such that the total cost of determining if the rule is satisfies is minimal.
    - NP-complete

Forward chaining ...

- Conflict sets
  - At any one time several rules will satisfy all of their preconditions, how do you choose which one to execute (this is usually called “firing”)
    - The first one in the rule base
    - The most specific one
    - The one most recently satisfied by changes to WM.
    - ...
- Incremental FC
  - Do not recalc which rules match every time a new fact is deduced. (The conflict set does not change much because of one new fact)
    - Rete algorithm (a time-space trader)

SOAR

- A “general” mechanism for learning and acting
  - Newell (CMU), Laird (PARC), Rosenbloom(Stanford)
  - Throughout 1980's
  - Based on production systems

- Assume exists a “general performance system”
- Then a general learner must be general in
  - Task – works on all tasks
  - Knowledge – based on any info (examples, hints, ...)
  - Aspect – works on all aspects of system

Soar – the system

- Tasks have 4 required components
  - Goal – checkmate
  - Current problem space – chess
  - State – the chess board
  - Operator – a legal move from among many
    - Also “augmentations”
- Many tasks can be worked on concurrently
- Each task can have many subtasks

Soar tasks

- Tasks start with only a goal statement
  - problem can be broken down into filling the rest of the task fields
    - Namely: problem space, initial state and operator
- LT memory is a production system
  - Rules fire in parallel during “elaboration phase” which is used to to select operator
  - Rules fire until “quiessence”
    - Different for prior discussion of production systems?

Soar – performance

- Hope at end of elaboration phase is a uniquely identified operator to apply
- However, there may be an “impass”
  - No operator to apply (dead end)
  - Several operators appear to be equivalent – maze
  - No operator is better than any other – dark maze
  - Operators might be applied but all are rejected
- If have an impass, create a new (sub) task
  - Note that the new task might have a different problem spaces than the parent task

Learning in Soar

- 3 problems need to be addressed when thinkin about learning within a performance system
  1) When is learning needed
  2) What needs to be learned
  3) When is the info to be learned available
- In Soar these have natural answers – subtask solution
- But Soar subtasks are rather specific
  - They need to be generalized
    - “Identifier variabilization”
    - Implicit generalization – the subtask contains only a fraction of the info in the larger task

Backward Chaining
Logic Programming

- Start with a conclusion and work backward until you find a set of facts that are in the database

- Negation as failure
- Infinite loops
  - Depth-first backward chaining is “Incomplete”
    - Is breadth first also incomplete?
    - Is this a problem for forward chaining also?

More Prolog -- Building Lists

- Problem – determine if a list is a palindrome
  - Yes: [], [a], [a,a], [a,b,a], ...
  - No: [a,b], ...
- Idea: a list is a palindrome if the list and its reverse are identical
  - palindrome(X) :- samelist(X,Y), reverse(X,Y).
  - Samelist
    - Base case: samelist([],[]).
    - Other rules?
  - Reverse
    - Base case: reverse([], []).
    - reverse([H|T],Res) :- reverse(T,Res2), append(Res2,[H], Res).
      - NOTE: the reversed list is being built on the returns

Prolog – more list building

- Recognize a^nb^n
  - Yes: [], [a,a,a,b,b,b], ...
  - No: [a], [b,b], [a,a,b,b,b], ...
- Basecase:
  - anbn([]).
- Rule:
  - anbn([a|Tail]) :- anbn(Tail, [a]).
- Another basecase:
  - Anbn([], []).
- More Rules:
  - anbn([a|Tail], Aa) :- anbn(Tail, R), append(Aa,[a]. R)
  - anbn([b|Btail], [a|Atail]) :- anbn(Btail, Atail).

Prolog – doing math

- Same problem:
- anbn2([]).
- anbn2([a|T]) :- anbn2(T, 1).
- anbn2([], 0).
- anbn2([a|T], N) :- plus(N,1,Sm), anbn2(T,Sm).
- anbn2([b|T], N) :- plus(N, -1, Sm), anbn2(T,Sm).

Prolog – more lists

- Take a multilevel list and flatten it
  - [[[[a]]]] -> [a] or [a[b[c,d],e]] -> [a,b,c,d,e]
- Base case
  - fltten([], []).
- Calls
  - fltten([A|T], X) :- atom(A), fltten(T,Y), append([A],Y,X).   % note the use of “atom”
  - fltten([A|T], X) :- append([A],Y,X), fltten(T, Y).
- What happens is reverse append and fltten in last rule?