Today's Lecture

- A more efficient way to compute the results of a datalog program
- Datalog\textsuperscript{−}: Datalog with negation
  - Problems
  - Stratification
- A little bit of magic

Evaluation - Non-recursive Datalog

- Non-recursive datalog = datalog without recursion
- Assume also no negation
- Then the datalog program can be represented as a acyclic graph and evaluated in a bottom-up manner

\[
\begin{align*}
S(X,Y,Z) & : T(X,Y,Z) \\
P(X,Y) & : R(X, a, Y), S(Y, b, c) \\
P(X, Z) & : R(X, Z, Y)
\end{align*}
\]
Consider a datalog program (with no negation)
For each IDB $P_i$

$P_i = \{\}$

Repeat until no change in IDBs {
apply rules in the datalog program on edb and idbs
For each $i$, if new facts $F_i$ are inferred

$P_i = P_i \cup F_i$
}

Consider every possible assignment of values to variables. For every such assignment which makes all the subgoals true, the tuple corresponding to the head is true and is the current fact inferred.

WS$^3$ is a fixpoint for the WheelSubparts program
A fixpoint $v$ of a function $f$ is such that $f(v) = v$
The minimal model of a datalog program is the smallest fixpoint
Example

TrikeSubparts(Y) :: Assembly(trike, Y, _)
TrikeSubparts(Y) :: TrikeSubparts(X), Assembly(X,Y, _)

- TS₀ = {}
- Round 1:
  - Facts inferred = { TS(wheel), TS(frame) }
  - :: TS¹ = { TS(wheel), TS(frame) }
- Round 2:
  - Facts inferred = { TS(wheel), TS(frame), TS(seat), TS(pedal), TS(spoke), TS(tire) }
  - :: TS² = { TS(wheel), TS(frame), TS(seat), TS(pedal), TS(spoke), TS(spoke), TS(tire) }

Example

- Round 3:
  - Facts inferred = { TS(wheel), TS(frame), TS(seat), TS(pedal), TS(spoke), TS(tire), TS(rim), TS(tube) }
  - :: TS³ = { TS(wheel), TS(frame), TS(seat), TS(pedal), TS(spoke), TS(tire), TS(rim), TS(tube) }
- Round 4:
  - Facts inferred = { TS(wheel), TS(frame), TS(seat), TS(pedal), TS(spoke), TS(tire), TS(rim), TS(tube) }
  - No new predicates can be inferred!
  - STOP

<table>
<thead>
<tr>
<th>Part</th>
<th>Subpart</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trike</td>
<td>Wheel</td>
<td>3</td>
</tr>
<tr>
<td>Trike</td>
<td>Frame</td>
<td>1</td>
</tr>
<tr>
<td>Frame</td>
<td>Seat</td>
<td>1</td>
</tr>
<tr>
<td>Frame</td>
<td>Pedal</td>
<td>1</td>
</tr>
<tr>
<td>Wheel</td>
<td>Spoke</td>
<td>2</td>
</tr>
<tr>
<td>Wheel</td>
<td>Tire</td>
<td>1</td>
</tr>
<tr>
<td>Tire</td>
<td>rim</td>
<td>1</td>
</tr>
<tr>
<td>Tire</td>
<td>Tube</td>
<td>1</td>
</tr>
</tbody>
</table>
Observations

• Observation 1:
  – Naïve evaluation algorithm is inefficient
  – Facts are repeatedly inferred even though they are already known to be true from the previous rounds

• Observation 2:
  – At round $i$, $(i > 1)$, if a fact $f$ is inferred for the first time in round $i$, then it must use at least one new fact inferred in the previous round (round $i-1$)
  – Otherwise, the fact $f$ must have already been inferred in round $i-1$ or less

Example

• $TS^0 = \{\}$
• Round 1:
  – Facts inferred = { $TS(\text{wheel})$, $TS(\text{frame})$ }
  – $TS^1 = \{ TS(\text{wheel}), TS(\text{frame}) \}$
• Round 2:
  – Facts inferred = { $TS(\text{wheel})$, $TS(\text{frame})$, $TS(\text{seat})$, $TS(\text{pedal})$, $TS(\text{spoke})$, $TS(\text{tire})$ }

  • $TS(\text{seat}) :: TS(\text{frame}), \text{Assembly}(\text{frame, seat, 1})$
  • $TS(\text{pedal}) :: TS(\text{frame}), \text{Assembly}(\text{frame, pedal, 1})$
  • $TS(\text{spoke}) :: TS(\text{wheel}), \text{Assembly}(\text{wheel, spoke, 2})$
  • $TS(\text{tire}) :: TS(\text{wheel}), \text{Assembly}(\text{wheel, tire, 1})$

<table>
<thead>
<tr>
<th>Part</th>
<th>Subpart</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trike</td>
<td>Wheel</td>
<td>3</td>
</tr>
<tr>
<td>Trike</td>
<td>Frame</td>
<td>1</td>
</tr>
<tr>
<td>Frame</td>
<td>Seat</td>
<td>1</td>
</tr>
<tr>
<td>Frame</td>
<td>Pedal</td>
<td>1</td>
</tr>
<tr>
<td>Wheel</td>
<td>Spoke</td>
<td>2</td>
</tr>
<tr>
<td>Wheel</td>
<td>Tire</td>
<td>1</td>
</tr>
<tr>
<td>Tire</td>
<td>Rim</td>
<td>1</td>
</tr>
<tr>
<td>Tire</td>
<td>Tube</td>
<td>1</td>
</tr>
</tbody>
</table>

New facts inferred at round 1
Semi-naïve Evaluation

• Idea: remember what are the new facts inferred at each round for each IDB predicate
  – That is, for each IDB predicate \( P \), we keep track of \( \Delta P \)
  - \( \Delta P \) is the set of new facts inferred in the last round
  – In each round, apply the rules such that at least one IDB in \( \Delta P \) is used in the inference

Semi-naïve Evaluation

• For each IDB \( P_i \),
  – \( P_i = \{\} \), \( \Delta P_i = \{\} \)
• Repeat until no change in IDBs {
   infer new \( \Delta P_i \) for each \( i \) by applying rules in the datalog program on edbs and current \( \Delta P_i \)s; except during the first round, the inference of new facts must use at least one predicate in the current \( \Delta P_i \)s
   Remove from \( \Delta P_i \) all facts that are already in \( P_i \)
   for each \( P_i \), \( P_i = P_i \cup \Delta P_i \)
}
Example

TrikeSubparts(Y) :: Assembly(trike, Y, _)
TrikeSubparts(Y) :: TrikeSubparts(X), Assembly(X,Y,_)
What is the answer to the following query?
Big(P) :- Assembly(P, S, Q), Q > 2, ¬Small(P)
Small(P) :- Assembly(P, S, Q), ¬Big(P)

Apply Big rule then Small rule
- Small is empty, so Big = {trike}
- Small = {frame, wheel, tire}

Apply Small rule then Big rule
- Big is empty, so Small = {trike, frame, wheel, tire}
- Big = {}

Note that neither one is less than the other. We have two fixpoints

<table>
<thead>
<tr>
<th>Part</th>
<th>Subpart</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trike</td>
<td>Wheel</td>
<td>3</td>
</tr>
<tr>
<td>Trike</td>
<td>Frame</td>
<td>1</td>
</tr>
<tr>
<td>Frame</td>
<td>Seat</td>
<td>1</td>
</tr>
<tr>
<td>Frame</td>
<td>Pedal</td>
<td>1</td>
</tr>
<tr>
<td>Wheel</td>
<td>Spoke</td>
<td>2</td>
</tr>
<tr>
<td>Wheel</td>
<td>Tire</td>
<td>1</td>
</tr>
<tr>
<td>Tire</td>
<td>rim</td>
<td>1</td>
</tr>
<tr>
<td>Tire</td>
<td>Tube</td>
<td>1</td>
</tr>
</tbody>
</table>

To know which is the answer evaluated by the program, the user has to be aware of the procedure of evaluation.

The problem: With the use of NOT, the addition of tuples in one relation can “reject” the inference of tuples in other relations. This does not happen when there is no negation.
Big(P) :- Assembly(P, S, Q), Q > 2, NOT Small(P)  
Small(P) :- Assembly(P, S, Q), NOT Big(P)  

Big depends on Small and Assembly  
Small depends on Big and Assembly  
(Big depends on itself, Small depends on itself)  

In general, if P depends on Q and Q depends on R, then P depends on R  

Big and Small are mutually recursive  

Big depends negatively on Small  
Small depends negatively on Big  

Stratification  

• To deal with the problem caused by negation, one widely adopted solution is to consider only datalog programs that are stratifiable  
• Classify tables into strata or layers:  
  – Stratum 0: Predicates that do not depend on other predicates  
  – Stratum 1: Predicates that depend on predicates in stratum 0 or 1, and depend negatively only on tables in stratum 0  
  – ...  
  – Stratum n: Predicates that do not belong to a lower level strata, depend only on predicates in stratum n or lower strata, and depend negatively only on predicates in lower strata  
• A datalog program is stratifiable if it can be classified into strata according to the above algorithm  
• Stratification is a syntactic restriction to datalog programs
Example

Big(P) :- Assembly(P, S, Q), Q > 2, NOT Small(P)
Small(P) :- Assembly(P, S, Q), NOT Big(P)

- Big and Small are not stratifiable
  - Stratum 0: Assembly (nodes with no outgoing edges)
  - Stratum 1: Big, Small???

Example

Big'(P) :- Assembly(P, S, Q), Q > 2
Small'(P) :- Assembly(P, S, Q), NOT Big'(P)

- Stratum 0: Assembly
- Stratum 1: Big'
- Stratum 2: Small'
- Therefore, the datalog program consisting of Big' and Small' is stratifiable
- Note that this program is not recursive
Evaluating Stratified Datalog programs

- Evaluate stratum-by-stratum, starting from stratum 0
- At each stratum, compute the least fixpoint using naïve or semi-naïve algorithm

- Stratification gives a natural evaluation order and goes well with intuition; any occurrence of NOT involves a predicate in the lower stratum, which by then is completely defined!
- The result is a least fixpoint (although there can be more than one)

Big'(P) :- Assembly(P, S, Q), Q > 2
Small'(P) :- Assembly(P, S, Q), NOT Big'(P)

- Stratum 0: Assembly
- Stratum 1: Big'
  - Big' = { trike }
- Stratum 2: Small'
  - Small' = { frame, wheel, tire }

<table>
<thead>
<tr>
<th>Part</th>
<th>Subpart</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trike</td>
<td>Wheel</td>
<td>3</td>
</tr>
<tr>
<td>Trike</td>
<td>Frame</td>
<td>1</td>
</tr>
<tr>
<td>Frame</td>
<td>Seat</td>
<td>1</td>
</tr>
<tr>
<td>Frame</td>
<td>Pedal</td>
<td>1</td>
</tr>
<tr>
<td>Wheel</td>
<td>Spoke</td>
<td>2</td>
</tr>
<tr>
<td>Wheel</td>
<td>Tire</td>
<td>1</td>
</tr>
<tr>
<td>Tire</td>
<td>rim</td>
<td>1</td>
</tr>
<tr>
<td>Tire</td>
<td>Tube</td>
<td>1</td>
</tr>
</tbody>
</table>

Note that Small'={ trike, frame, wheel, tire }, Big'={} is another least fixpoint for this program although it is not the result of stratified evaluation
Another Example

Components(P,S) :- Assembly(P,S,Q)
Components(P,S) :- Assembly(P, P’, Q), Components(P’, S)
Big’(P) :- Components(P, S)
Small’(P) :- Components(P,S), NOT Big’(P)

• A recursive datalog program with negation
• Stratum 0: Assembly
• Stratum 1: Components, Big’
• Stratum 2: Small’

• Evaluation:
  – Components = {…}, Big’ = {…}
  – Small’ = {…}

Using magic to speed up evaluation

• In relational algebra, selections are often pushed to the innermost expression possible in order to discard irrelevant tuples early
  – Avoids subsequent redundant evaluation steps
  – Especially useful if condition is highly selective, i.e., only a handful of tuples belong to the result (compared to the input relation)
• Same idea can be applied to non-recursive datalog to speed up evaluation. Can we also apply the same technique to recursive datalog programs?
Pushing selections to avoid irrelevant inferences

SameLevel(S1,S2) :- Assembly(P,S1,_),
    Assembly(P,S2,_)  
SameLevel(S1,S2) :- Assembly(P1,S1,_),
    SameLevel(P1,P2),
    Assembly(P2,S2,_)  

• To find all parts at the same level of spoke (spoke is the first field), it is irrelevant to compute all parts that are at the same level with rim or tube, for example:

    trike
      /   
     /     
wheel  frame
      |     |
  spoke tire
      |     |
rim tube

Pushing selections to avoid irrelevant inferences

SameLevel(spoke,S2) :- Assembly(P,spoke,_),
    Assembly(P,S2,_)  
SameLevel(spoke,S2) :- Assembly(P1,spoke,_),
    SameLevel(P1,P2), Assembly(P2,S2,_)  

• Can we simply replace S1 with spoke?
• The above program computes
  – SameLevel = { (spoke, tire) }
  – Wrong because seat and pedal are also at the same level
  – The problem: This program no longer infers that wheel and frame are at the same level, which is necessary for the other inferences that seat and pedal are also at the same level with spoke.

Winter 2003
Pushing selections to avoid irrelevant inferences

• Idea: Compute all ancestors of spoke. Then compute all parts that are at the same level with each of these ancestors

\[
\text{MagicSameLevel}(P) \leftarrow \\
\text{MagicSameLevel}(S), \text{Assembly}(P,S,\_)
\]

\[
\text{MagicSameLevel}(\text{spoke}) \leftarrow 
\]

• MagicSameLevel = \{spoke, wheel, trike\}

\[
\text{SameLevel}(S1,S2) \leftarrow \\
\text{MagicSameLevel}(S1), \\
\text{Assembly}(P,S1,\_), \text{Assembly}(P,S2,\_)
\]

Intuitively, this two predicates restricts the search to only those that are at the same level with spoke, wheel, or trike

Pushing selections to avoid irrelevant inferences

• The entire program:

\[
\text{MagicSameLevel}(P) \leftarrow \\
\text{MagicSameLevel}(S), \text{Assembly}(P,S,\_)
\]

\[
\text{MagicSameLevel}(\text{spoke}) \leftarrow 
\]

\[
\text{SameLevel}(S1,S2) \leftarrow \\
\text{MagicSameLevel}(S1), \\
\text{Assembly}(P,S1,\_), \text{Assembly}(P,S2,\_)
\]
Magic Sets algorithm

• Idea is to rewrite the datalog program so that the rewritten program is more restricted during its evaluation; the rewritten program is still equivalent to the original one

• Roughly 3 steps:
  – Input: the query pattern. E.g., SameLevel(spoke, S1)
  – Output: the rewritten datalog program

  

  \[
  \begin{align*}
  \text{SameLevel}(S1,S2) & \iff \text{Assembly}(P,S1,\_), \text{Assembly}(P,S2,\_) \\
  \text{SameLevel}(S1,S2) & \iff \text{Assembly}(P1,S1,\_), \text{SameLevel}(P1,P2), \text{Assembly}(P2,S2,\_) \\
  \end{align*}
  \]

  • Generate Adorned Program
    – Program is rewritten to make the relevant patterns of the datalog program explicit

    \[
    \begin{align*}
    \text{SameLevel}(S1,S2) & \iff \text{Assembly}(P,S1,\_), \text{Assembly}(P,S2,\_) \\
    \text{SameLevel}(S1,S2) & \iff \text{Assembly}(P1,S1,\_), \text{SameLevel}(P1,P2), \text{Assembly}(P2,S2,\_) \\
    \end{align*}
    \]

  • Add Magic Filters
    \[
    \begin{align*}
    \text{SameLevel}(S1,S2) & \iff \text{Magic}(S1), \text{Assembly}(P,S1,\_), \text{Assembly}(P,S2,\_) \\
    \text{SameLevel}(S1,S2) & \iff \text{Magic}(S1), \text{Assembly}(P1,S1,\_), \text{SameLevel}(P1,P2), \text{Assembly}(P2,S2,\_) \\
    \end{align*}
    \]
• Define Magic Filters
  Magic(P) :- Magic (S), Assemby(P,S,_)  
  Magic(spoke) :-.
• See book for a more complete description

Summary
• Semi-naïve evaluation: A more efficient way to compute the results of a datalog program
• Stratification as a syntactic restriction to deal with datalog with negation
• A little bit of magic – to be more selective in your inferences so as to speed up efficiency
• Datalog is often used, in one form or the other, as an “internal” language in several systems
• Lots of research in this area. See bibliography for a complete reference