On Querying OBO Ontologies using a DAG Pattern Query Language

Amarnath Gupta
Simone Santini

Univ. of California San Diego

What is an OBO Ontology?

- OBO – Open Biomedical Ontologies is a consortium
  - Serves a standard for developing Gene-Ontology-like ontologies (despite subtle differences)
  - Maintains a repository of biomedical ontologies that have this structure
  - Many members of the repository are on related (or relatable) areas

Other Elements of an OBO Specification

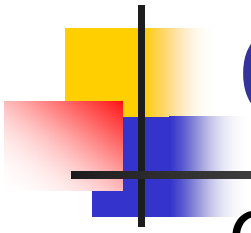
- An OBO Ontology may specify
  - A set of type names through a *typedef* declaration
  - A set of subset names through a *subsetdef* declaration
- Each term can also specify
  - **relationship:** a typed relationship between this term and another term. The value of this tag should be the relationship type id, and then the id of the target term.
  - **domain, range:** the children (parents) that can be assigned to relationships with this type. If the domain is set, term relationships with this type may only have children (parents) that are the same as, or subclasses of, the domain term
  - **is_transitive**, **is_symmetric**, **is_cyclic:** descriptors of relationships.

An example snippet from an OBO Ontology

[Term]
id: GO:0003674
name: molecular_function
def: "The action characteristic of a gene product." [GO:curators]
subset: goslim

[Term]
id: GO:0016209
name: antioxidant activity
is_a: GO:0003674
def: "Inhibition of the reactions brought about by dioxygen or peroxides. …"
[ISBN:0198506732]

[Term]
id: GO:0045174
name: glutathione dehydrogenase (ascorbate) activity
xref_analog: EC:1.8.5.1 ""
def: "Catalysis of the reaction…" [EC:1.8.5.1]
synonym: dehydroascorbate reductase []
is_a: GO:0009055 \ is_a: GO:0015038 \ is_a: GO:0016672



Our Current Abstraction

- Consider a database where
  - the data is a set of elements,
    - each element is structured like an unranked directed acyclic graph
      - The nodes of the DAG have properties represented as attribute-value pairs
      - The edges of the DAG
        - are binary
        - have no labels*
        - are unordered
- How should we store data, formulate queries and retrieve information from such a database?

Why this DAG Abstraction?

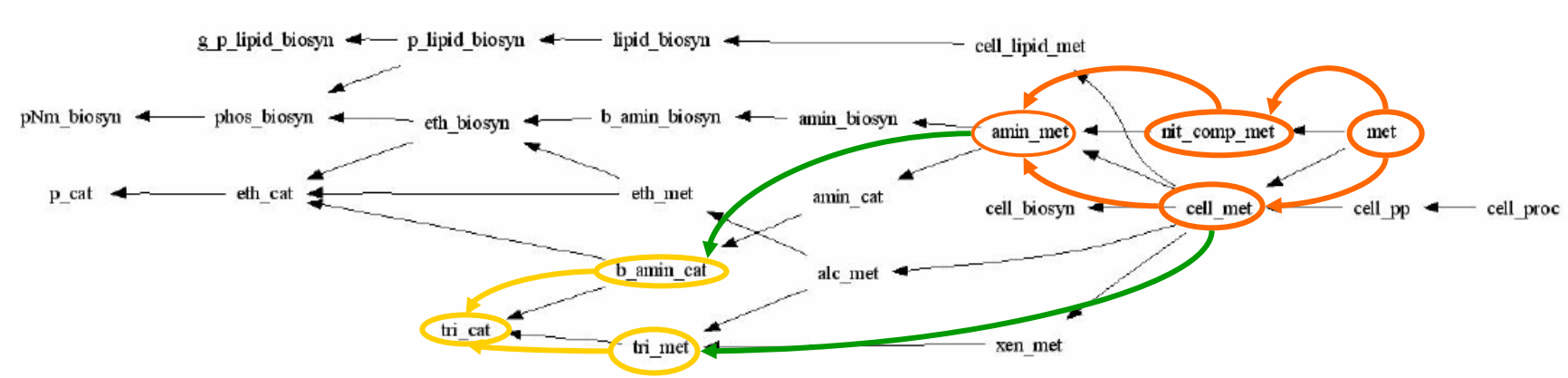
- A lot of data in the world are DAG-structured
  - Many ontologies
  - Classification systems with multiple inheritance
  - Phylogenetic networks that consider speciation, hybridization and lateral gene transfer [Moret 2004]
- Tree databases are currently a strong research focus
  - DAGs form the next level in structural complexity and hence the next frontier to be conquered
  - Some theory and techniques from tree database research can be extended to DAGs

Desiderata for Querying DAGs

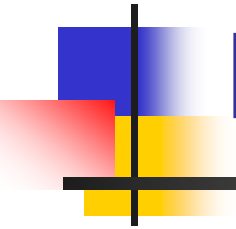
- Queries should
  - permit standard value-based queries on node content
    - Allow the special case where edges have their own content
  - support pattern queries
    - return ***subgraphs*** (witness graphs) that match the conditions in the query
  - support construction of result graphs by composing partial results of subqueries
  - support structure-aggregate queries that compute structural summaries of witness graphs

Combine both value-based queries and composable, structure-based queries

An Example


g_p_lipid_biosyn
p_lipid_biosyn
lipid_biosyn
cell_lipid_met
pNm_biosyn
phos_biosyn
eth_biosyn
b_amin_biosyn
amin_biosyn
amin_met
nit_comp_met
met
p_cat
eth_cat
eth_met
amin_cat
cell_biosyn
cell_met
cell_pp
cell_proc
b_amin_cat
alc_met
tri_cat
tri_met
xen_met

Toward a Query Language for DAG databases



Pattern Queries

What is a pattern query?

- Given a "pattern graph" H and a "data graph" G
  - α is a mapping from nodes of H to the nodes of G such that
    - for every node n_i of H, $\alpha(n_i)$ in G are the nodes that satisfy a predicate p(n_i)
  - μ is a mapping from edges of H to paths G such that
    - for every edge e_i (n_k, n_l) of H, there is a path from $\alpha(n_k)$ to $\alpha(n_l)$ in G such that the path satisfies some predicate p'(e_i)
  - p" is a predicate on the homeomorphic image of H on G
- A pattern query language specifies such predicates and mappings
- The result of a query is the set of subgraphs in G that satisfies both these mappings
  - Typically, the vocabulary for predicates p' is restricted

No constraint on node or edge disjointedness

L(Π), The Pattern Language

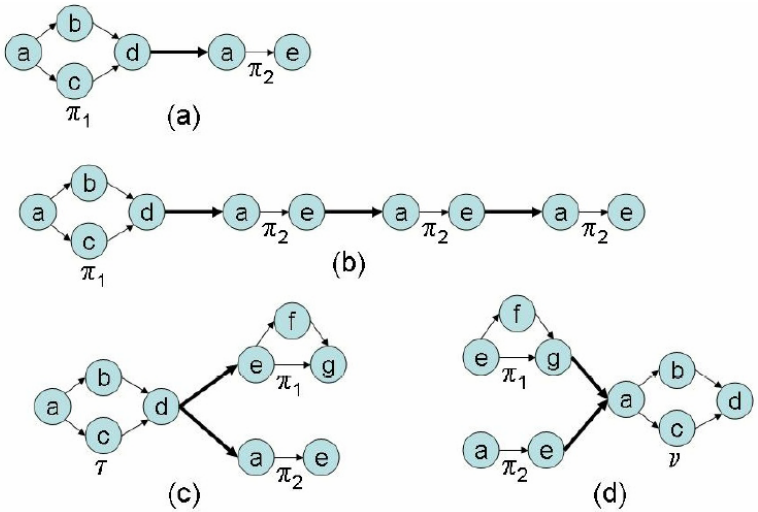
i) A predicate C in which the free variables are the names of the components of the node data type is a pattern; in particular t (the value "true");

ii) if π_1 and π_2 are patterns, then $\pi_1 - \pi_2$ is a pattern;

iii) if π and π' are patterns, and $n, m \in \mathbb{N} \cup \infty$, then $\pi'[-\pi](n,m)$ and $[\pi-](n,m)\pi'$ are patterns;

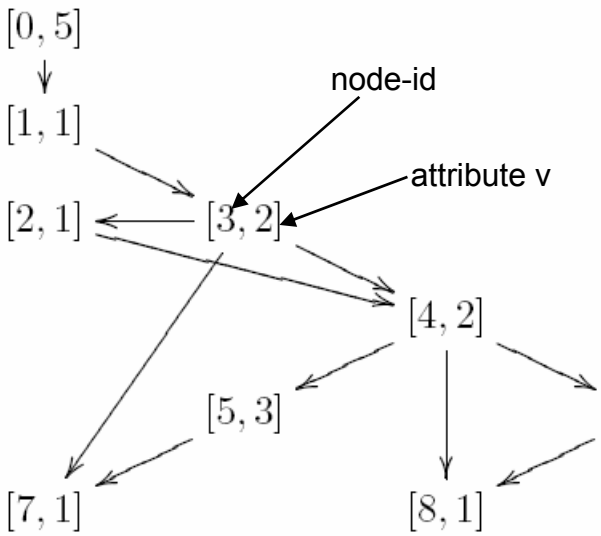
iv) if $\pi_1, \ldots, \pi_n$ are patterns, and τ, ν are patterns then $\tau - \{\pi_1-, \ldots, \pi_n-\}\nu$, $\tau\{-\pi_1, \ldots, -\pi_n\} - \nu$, $\{\pi_1-, \ldots, \pi_n-\}\nu$, and $\tau\{-\pi_1, \ldots, -\pi_n\}$ are patterns;

v) if π is a pattern and v a variable name, then $v : \pi$ is a pattern;



(a) $\pi_1 - \pi_2$, (b) $\pi_1[-\pi_2](2,4)$, (c) $\tau\{-\pi_1, -\pi_2\}$ and (d) $\{\pi_1-, \pi_2-\}\nu$

Patterns with Variables



The pattern **(v = 1)[−(v = 2)]*−(v = 1)** matches the graphs
[1, 1] → [3, 2] → [7, 1] , [1, 1] → [3, 2] → [2, 1] , [1, 1] → [3, 2] → [4, 2] → [8, 1] , and so on.

Adding variables **y : (v = 1)[−(v = 2)]* − x : (v = 1)**
the pattern will produce the set of pairs (y, x): {
([1, 1], [2, 1]), ([1, 1], [7, 1]), ([1, 1], [8, 1]), ([2, 1], [8, 1])}

Now consider the pattern query:
$\cup$[{x − y|g $\vdash$ y : (v = 1)[−(v = 2)]*−x : (v = 1) ← G_1}]
Result:
{[2, 1] → [1, 1], [7, 1] → [1, 1], [8, 1] → [1, 1], [8, 1] → [2, 1]}

Variables can be nodes or subgraphs

An Aside: Monoids

Definition 1 (Monoid) *A monoid of type T is a pair $(\oplus, \mathcal{Z}_\oplus)$, where $\oplus$ is an associative function of type $T \times T \rightarrow T$ and $\mathcal{Z}_\oplus$ is the left and right identity of $\oplus$.*

type $T_\oplus$	$\oplus$	$\mathcal{Z}_\oplus$	C/I
int	$+$	0	C
int	$\times$	1	C
int	max	0	CI
bool	$\vee$	false	CI
bool	$\wedge$	true	CI

A. Primitive Monoids

type $T_\oplus$	$\oplus$	$\mathcal{Z}_\oplus$	$\mathcal{U}_\oplus(a)$	C/I
set(α)	$\cup$	$\{\}$	$\{a\}$	CI
bag(α)	$\uplus$	$\{\!\{\}\!\}$	$\{\!\{a\}\!\}$	C
list(α)	$+\!\!+$	$[\,]$	$[a]$	

B. Collection Monoids

Embedding Π in Monoid Comprehension

- Monoid comprehension

monoid

generators

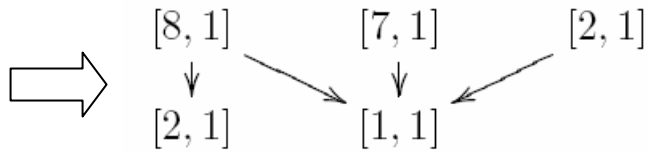
$\omega\{e|q_1,\ldots,q_n\}$ where

  - An expression of the form
    - q_i may have one of the following forms
      - $q_i \equiv x_i \leftarrow A$, where A is a constant or another monoid comprehension
      - $q_i \equiv g \vdash \pi(y_1,\ldots,y_m)$, where
        - y's are the free variables of pattern π
        - g is the collection of variables and constants collected from prior environments of computation (q's)
      - $q_i \equiv P(y_1,\ldots,y_m)$, where
        - P is a predicate
        - y's are the free variables of prior environments

Graph Monoids

- In addition to standard monoids, ω could be graph monoids
  - merge (g_1, g_2) – union the nodes and edges of the two graphs, fusing nodes that are equivalent
  - gmin(g_1, g_2) – the largest common graph contained in g_1, g_2
  - gmax(g_1, g_2) – the smallest graph g for which g_1, $g_2 \subset g$
    - gmax [{x − y|g ⊢ y : (v = 1)[−(v = 2)]*−x : (v = 1)}]

{[2, 1] → [1, 1], [7, 1] → [1, 1], [8, 1] → [1, 1], [8, 1] → [2, 1]}



Example Queries

- 1. Which biosynthesis processes under lipid biosynthesis are also classified as amine biosynthesis? (Q1)

$$\cup[\{x|g \vdash \{substr(name, \text{“}lipid\_biosyn\text{”}), substr(name, \text{“}amin\_biosyn\text{”})\}\#x : substr(name, \text{“}biosyn\text{”}) \leftarrow GO\}]$$

- 2. How does phosphatidylethanolamine biosynthesis (phos biosyn in Fig. 1) derive from cellular metabolism (cell met)? (Q2)

$$\cup[\{x|g \vdash x : ((name = \text{“}cell\_met\text{”})\#(name = \text{“}xen\_met\text{”})) \leftarrow GO\}]$$

- 3. Is there a case where a xenobiotic process (e.g., xen met) is a subprocess of at least two forms of cellular metabolism? (Q3)

$$\cup[\{z|g \vdash (name = \text{“}cell\_met\text{”})\#\{x, y\}\#z, (name = \text{“}xen\_met\text{”})\#z \leftarrow GO\}]$$

- 4. construct a reduced data graph by deleting all metabolism nodes except met, and connecting the non-deleted parent(s) of a deleted node n to its non-deleted children. (Q4)

$$merge(merge\{x{-}y|g \vdash x : ([t{-}]{*}B){-}A{-}y : (C[{-}t]{*})\}, merge\{z|g \vdash z : ([\neg A{-}]{*}\neg A)\})$$

An Algebra for DAGs

- 4 classes of algebraic operators
  - Pattern matching
    - select, path, match, …
  - Monoid manipulation
    - merge, g_union, g_intersect, …
  - Functional
    - apply, chain, …
  - Construction
    - insert_node, insert_edge, tuple_constructor …
- Additional functions like aggregates
    - diameter, size, lca…

Chen et al: VLDB 2005

A Core Algebra

path: the call path(g, n_1, n_2, h, k) return the set of paths between the nodes n_1 and n_2 in the graph g such that the length of the path is between h and k; the typing of this function is

$$\frac{g : \Gamma(\alpha) \;\; n_1, n_2 : \alpha \;\; h, k : \text{int}}{\text{path}(g, n_1, n_2, h, k) : \{[\alpha]\}} \tag{18}$$

merge: the call merge(g_1, g_2) merges the two graphs g_1 and g_2 by identifying the nodes with equal value; the operator requires that the two graphs have at least one node that can be identified: it returns null for disconnected graphs; its typing is

$$\frac{g_1, g_2 : \Gamma(\alpha)}{\text{merge}(g_1, g_2) : \Gamma(\alpha)} \tag{19}$$

σ: the call $\sigma(g, P)$ returns the set of all nodes of the graph g that satisfy the predicate P; its typing is

$$\frac{g : \Gamma(\alpha) \;\; P : \alpha \rightarrow \mathbf{2}}{\sigma(g, P) : \{\alpha\}} \tag{20}$$

apply: The operator apply$[\omega](A, f)$ applies the function f to all the elements of the structure A, and collects the results in a structure of type ω. It typing is:

$$\frac{A : \nu(\alpha) \;\; f : \alpha \rightarrow \beta \cup \{\perp\}}{\text{apply}[\omega](A, f) : \omega(\beta)} \tag{21}$$

chain: given a set of paths S, a graph g that contains them, and two integers h, k, chain(g, S, h, k) builds all the chains that can be built out of paths in S taking each path between h and k times. Its typing is:

$$\frac{S : \{[\alpha]\} \;\; g : \Gamma(\alpha) \;\; h, k : \text{int}}{\text{chain}[\omega](g, S, h, k) : \{[\alpha]\}} \tag{24}$$

From Pattern to Algebraic Plan

Preliminaries

- What is a plan?
  - An assignment of bound query variables to a structure that holds the pattern instance and the corresponding variables (called the environment)
  - a function call *plan(π,g,U)*
    - Where *g* is the input graph and *U* is the environment
- A simple example
  - Evaluating a single condition C
  - plan(z:C, g, e) =

```
u1 = (g, C);
e = apply[set](u1,
        fun x => (z := x)
    )
```

Assign to z the value x

The Translation Algorithm - I

- Consider the following pattern
  - y : (C1[–t]*C2[–t](5, 7) – x : (C3[–C4 – C5]*–C6) – C7)
- Step 1 – Normalize the expression
  - Break out the internal variables
    - y=C1[–t]*C2[–t](5, 7) – x – C7
    - x = C3[–C4 – C5]*–C6
  - Replace [-t]* and [t-]* by path symbols #, – or (a,b)
    - y=C1#C2(5, 7) – x – C7
    - x = C3[–C4 – C5]*–C6
  - Expand the * element
    - y=C1#C2(5, 7) – x – C7
    - x = C3–v*–C6
    - v = (C4 – C5)

The Translation Algorithm - II

- Step 2 – eliminate the repeated pattern[-π](n,m) by recursively calling plan
  - For a path pattern the fragment would be:
    - plan(x1 : (C4 – C5), g, u1);
    - u2 = apply[set](u1
      fun x2 => u1(x2) (Transform the set of environments into a set of graphs)
      );

      p_{45} = chain(g, u2, n, m);
  - Now the partially executed state looks like:
    - y=C1#C2(5, 7) – x – C7
    - x = C3 – p_{45} –C6

The Translation Algorithm - III

- Step 3 – replace C's with node sets they evaluate to
  - U1 = σ(g,C1)
  - …
- Step 4 – replace path symbols by set of paths
    - p_{12} = apply[set](U1, fun x => apply[set](U2, fun y => path(x, y, 0, infty))
    - p_{23} = apply[set](U2, fun x => apply[set](U3, fun y => path(x, y, 5, 7))
    - p_{34} = apply[set](U3, fun x => apply[set](U4, fun y => path(x, y, 1, 1))
    - …
  - Now the state looks like
    - $y=p_{12} \sim p_{23} \sim x \sim p_{67}$
    - $x = p_{34} \sim p_{45} \sim p_{56}$

The Translation Algorithm - IV

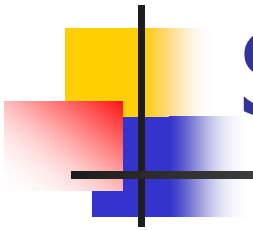
- Step 5 – replace path-valued variables by merging constituent paths
  - p_{36} = apply[set](p_{34}, fun x34 =>
    apply[set](p_{45}, fun x45 =>
    apply[set](p_{56}, fun x56 => merge(x34, merge(x45, x56)))
)
)
  - Enter p_{36} in the variable table for x
  - Our example
    - Perform $p_{12} \sim p_{23} \sim p_{36} \sim p_{67}$ and then derive p_{17}
- Step 6 – construct the environment

U = apply[set](p17, fun x17 =>
apply[set](p36, fun x36 => (x ▷ x36) ⊕ (y ▷ x17)
);

Tupling operator

Rewriting for Optimization

- Substitute the pattern
  - {select-block} {graph-retrieval-block} by
  - {select-block}{match-operation}{graph-retrieval-block}
- match – given graph g and pattern $\pi(\mathbf{y})$ where **y** is the set of free variables of π, and N, a candidate node-set for y, it returns a relation of bindings



Some Broad Comparisons

- How does this relate to XML query languages?
  - XML doesn't exactly apply because concepts like child ordering and document ordering are not relevant in our system
  - If our DAGs were trees, it can be proven that the expressive power of DQL (minus the construction part) will be equivalent to conditional XPath (Marx 2004)
- How about other semistructured languages like Lorel, UnQL and Strudel?
  - Most semistructured languages that support pattern queries are not based on monoid comprehension (exception: Fegaras and Maier)
  - DQL expressions more complex patterns
  - Lorel, UnQL does not support constructions
  - Strudel is the closest

Are biologists buying this?

- Our use cases are always driven by domain scientists' analysis needs
- Current use cases
  - Neuroscience: The Ontology Task Force for BIRN
    - Developing searchable lexicons and ontologies that are to be used for data integration called BIRNLex and MIND
      - Using ontologies like RO, FuGO, PATO,… and non-ontologies like UMLS in the process
  - Systems Biology
    - Extending SBML models with ontological references
    - Yeast classification database for MIPS
    - GO, of course
  - Biodiversity
    - Habitat classification

Conclusions and Future Work

- Conclusions
  - A simplified abstraction over ontology graphs
  - Useful for practical biological (and other) information exploration
  - Used in a system called Biological Networks [Baitaluk et al: BMC Bioinformatics 2006, Baitaluk et al: NAR 2006]
  - Being implemented in a system called OntoQuest [Chen et al: VLDB 2006]
- Future Work
  - Complete the calculus and algebra and the query processor
  - "Inferencing" aspects of ontologies
  - Extending the language to admit edge weights
  - Supporting "link analysis" type queries where path ranking and path strength are used
  - Extending to more general graphs

Acknowledgments

- BIRN OTF
  - Maryann Martone, UCSD
  - Christine Fenema Notestein, UCSD
  - William Bug, Drexel U.
  - Jessica Turner, UCI
  - Carol Bean, NIH
  - Daniel Rubin, Stanford
- Computer Science
  - Li Chen, UCSD
  - M. Erdem Kurul, Microsoft
- Systems Biology
  - Animesh Ray, KGI
  - Michael Baitaluk, UCSD
- Biodiversity
  - Karen Stocks, UCSD
  - NatureServe Team