UCRL- TR-22544

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

A Semantic Graph Query Language

Ian Kaplan
Lawrence Livermore National Laboratory

October 17, 2006

A Semantic Graph Query Language[1]

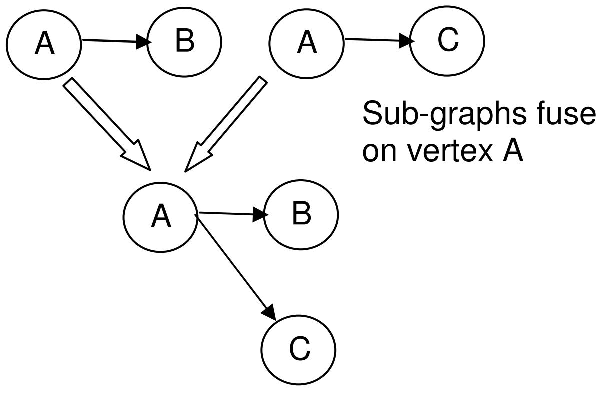
Ian Kaplan
Complex Networks Group, LLNL
October 17, 2006

Abstract

Semantic graphs can be used to organize large amounts of information from a number of sources into one unified structure. A semantic query language provides a foundation for extracting information from the semantic graph. The graph query language described here provides a simple, powerful method for querying semantic graphs.

Semantic Graphs

In a semantic graph information is represented by graph vertices that are joined to other graph vertices by edges. The possible structure of the graph is described by an ontology that defines the vertex types, the edge types and how edges may interconnect vertices to form a directed graph. The graph ontology also defines one or more attribute types for each vertex type. For example, a vertex of type *person* might have the attributes "name", "date of birth", "city of birth" and "country of birth". The key attributes of a vertex should be chosen to uniquely identify a vertex in the graph. Some semantic graphs also allow vertices to have non-key attributes that may take on multiple values and are not used to uniquely identify a vertex. For a vertex of type *person*, an example of a non-key attribute might be "address". The non-key attribute "address" could have multiple values, since many people have lived at more than one address. When vertices are published into a semantic graph, vertices with the same key attribute values will "fuse" (e.g., a vertex with a particular set of values will be represented in the graph once).

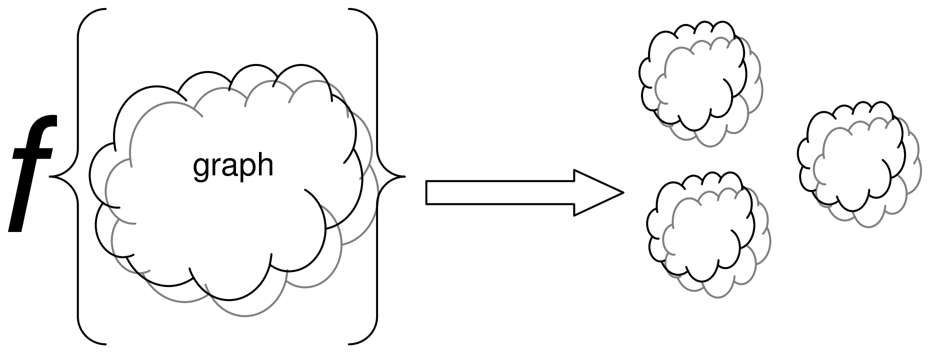


The edges in the graphs discussed in this paper are directed. There may be complementary edges, but bi-directionality in an edge is not required.

[1] LLNL technical report UCRL-TR-225447. This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48. Funding provided by LDRD 06-ERD-009.

Querying a Graph Database

A graph query does not add, remove or alter vertices or edges. The result of a query is logically a "view" of the graph, filtered according to the constraints of the query. The diagram below shows how a graph query function acts on a graph and produces a result that is itself a graph. The result graph may consist of the empty graph (e.g., no vertices or edges), a set of vertices without edges or a set of one or more connected graphs.



The graph query language described here does not alter the key attributes associated with a vertex or add or remove edges in the graph. Nor does the query language include support for loading data into the graph.

Query Language Design Goals

1. The query language should be supported on both scalar and large scale distributed systems. The query language should not rely on any particular underlying database technology.
2. The query language should provide the features needed to build sophisticated graph queries. These features should be as simple as possible to allow efficient query language implementation.
3. A query operation should be able to operate on either the entire graph (referred to as the "base graph", below) or on a sub-graph defined by another query.
4. Implementation of the language should include as little software infrastructure as possible. For example a graphical query interface should not be required to support the query language.
5. The query language should allow new functions to be added to the language by simply adding the function syntax to the parser and support software. There should be no impact from an addition on unrelated syntax. Examples of graph functions include path traversal and strongly connected sub-graphs.
6. The query language should support the features needed to act as building blocks for complex graph algorithms.

Graph Functions

Overview

The query language proposed here is built around functions which can be nested to create complex graph query operations. The general form for a query function is:

```
[NamedGraph "="] GraphFunction ";"
```

The optional *NamedGraph* assignment allows a unique name to be associated with the graph that results from executing the query. This named graph can be referenced in another query. The *NamedGraph* is an arbitrary quoted string. Each *NamedGraph* string must be unique. For example:

```
"Ian 01/19/2005 15:03:234" = adjacency("MyBaseGraph", "country of interest",
                                        vertex { person },
                                        edge {any });
```

If a *NamedGraph* already has an associated graph, it is an error to attempt to assign another graph to that name, as long as the graph associated with the *NamedGraph* string exists in the graph database. If a named graph is not provided in the query, the query system will automatically generate a unique name for the graph that results from the query.

When the information for a semantic graph is loaded into the graph system, a name for the graph and the associated ontology are given a name. This graph name is used in queries that reference the entire semantic graph associated with the ontology. The sub-graph that results from a query on a graph will have the same ontology as the input graph. Queries cannot reference more than one ontology, although they can reference multiple instances of that ontology.

In the query language, white space, formatting and character case are not significant. Queries are terminated by a semicolon. The graph functions supported in the initial query language are:

```
GraphFunction ::= union     |
                  intersect |
                  diff      |
                  filter    |
                  adjacency |
                  pattern
```

The **union**, **intersect** and **diff** functions logically act like binary operators, with two graph operands (which are the results of other graph functions).

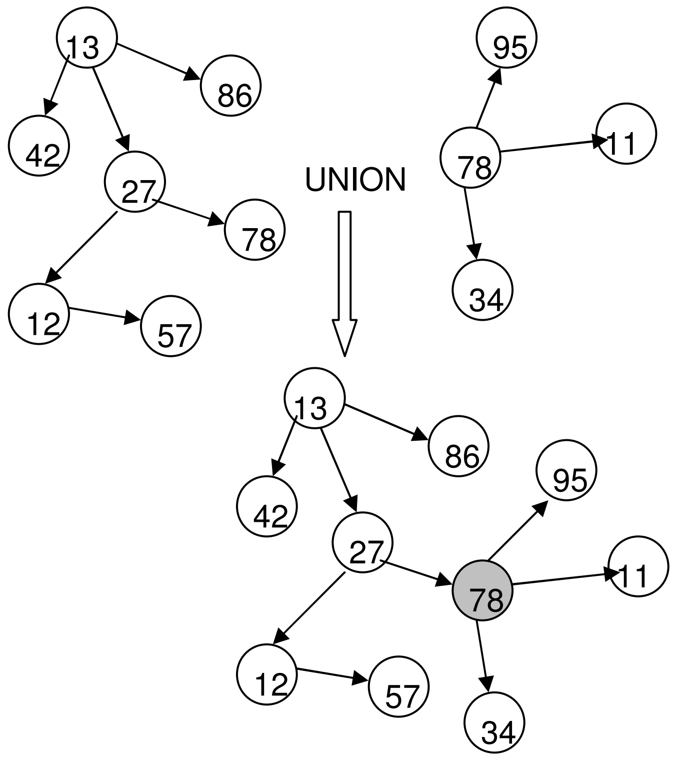
The **filter**, **adjacency** and **pattern** functions are given an input graph argument and calculate a sub-graph based on the other function arguments.

Binary Graph Functions

The binary graph functions **union**, **intersect** and **diff** each take two graph operands. These operands must be sub-graphs of the same semantic graph ontology. A runtime error will be reported if a query function references sub-graphs from different ontologies. For example, a binary graph function cannot have one operand that is an IMDB sub-graph and another operand that is a PubMed sub-graph.

Union function

The **union** function takes two graph arguments. The result of the **union** function will be a graph that combines the two graph arguments, where all of vertices are unique. This is shown in the diagram below, where vertex identity is indicated by an integer value. Note: integer identifiers for vertices are used to illustrate the operation of graph **union**. Vertices in the graph are identified by a hash ID formed from vertex attribute values.



Unless the two graph operands entirely overlap, the result of the **union** function will be larger than either of its input operands.

The general form of the **union** function is shown below:

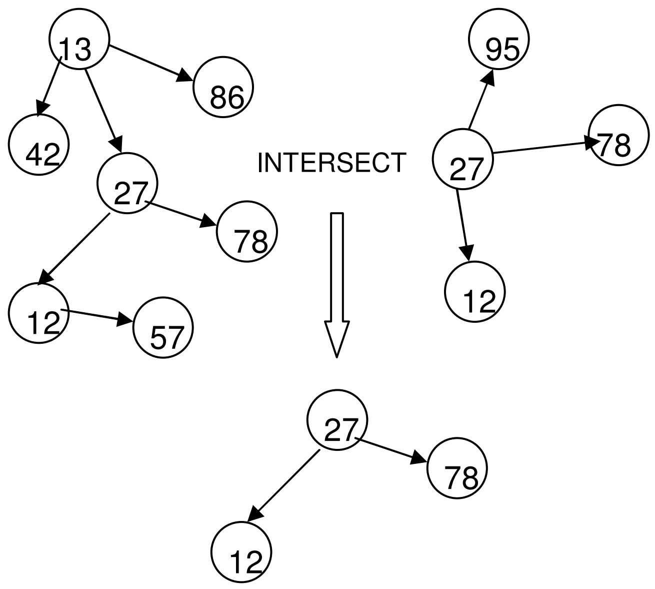
```
union (GraphFunction, GraphFunction );
```

The **union** below produces a graph (or set of graphs) that consists of the union of the graphs returned by the two adjacency functions.

```
union (adjacency( “MyBaseGraph”, “research_test”,
                vertex { person, country },
                edge { travels_to }),
      adjacency( “MyBaseGraph”, “research_test”,
                vertex { person, organization },
                edge { works_for });
      );
```

Intersect function

The **intersect** function returns the instances of vertices and edges that exist in common between the two graph arguments. This is shown in the diagram below. As in the previous diagram, the numbers are intended to indicate unique instances of vertices.



An instance of an edge is determined by its source and destination vertices. An edge will exist in the result only if instances of the source and destination vertex also exist in the result. For example, if a destination vertex is not in the intersection set, but the source vertex and edge are in the intersection set, the edge will be removed.

The general form of the **intersect** function is shown below:

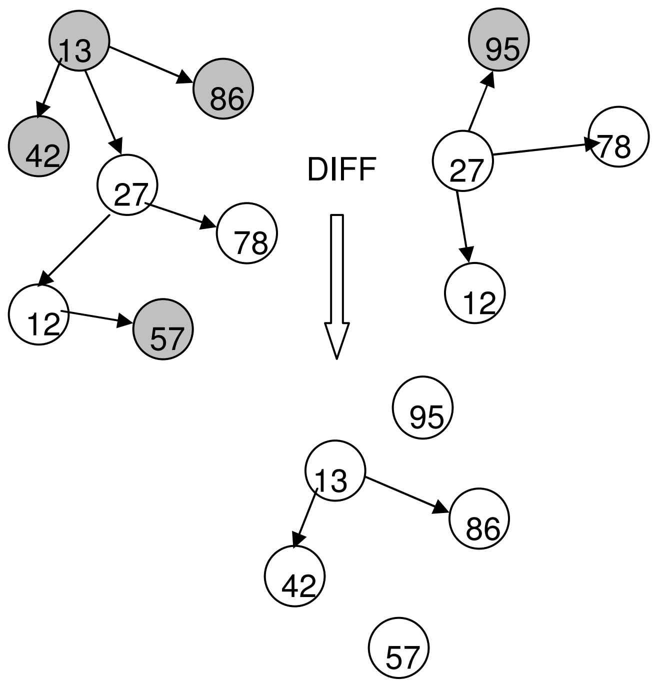
```
intersect (GraphFunction, GraphFunction );
```

The steps in calculating the **intersect** function are:

1. Create a graph that consists of the vertices and edges that are shared by the two graph operands.
2. Remove any edges that do not have a source and destination in the result graph.

Diff(erence) function

The **diff** function removes the vertices and edges that are <u>shared</u> by the two arguments. The operation of the **diff** function is shown in the diagram below:



The general form for the **diff** function is:

```
diff (GraphFunction, GraphFunction );
```

The steps in calculating the **diff** function are:

1. Create a graph that consists of the vertices and edges that are not shared by the two graph arguments.
2. Remove any edges that do not have a source and destination in the result graph.

Graph Filter Functions

Ontologies and Named Graphs

An ontology defines the type and structure of vertices (e.g., the vertex attributes) and the connections between the vertices (e.g., edge types and edge connections).

A data graph (referred to here simply as a "graph") is a set of vertices and edges constructed from a data set, where the graph structure is constrained by an ontology. Each graph that is loaded into the distributed system has an associated name and ontology. A base graph is a named graph that includes all of the vertices and edges loaded with a particular ontology structure. Queries create sub-graphs that are, logically, views of this base graph. The sub-graphs created by queries each have an associated name. This name is either explicitly assigned in the query or is implicitly assigned by the system to internally identify the query result. The ontology of a sub-graph is the same as the ontology of the input graph.

A query may only reference graphs associated with the same ontology. Queries across different ontologies will result in a run-time error. Queries across ontologies are disallowed because the result graph would have no defined ontological structure. A query may, however, reference two base graphs (e.g., graphs from different data loads) defined by the same ontology.

With the exception of the binary graph functions **union**, **intersect** and **diff**, all functions in the query language operate on an *InputGraph* (or *UniverseGraph*) argument, and produce a result which is a sub-graph of this *InputGraph* (or *UniverseGraph*). The *InputGraph* argument is either:

1. A unique string that is associated with a base graph or with a previously calculated sub-graph.
2. A query function. For example, a **filter** function may be used as the *UniverseGraph* argument for a **pattern** function.

Filter

The **filter** function filters its *InputGraph* argument based on the *VertexFilter* and *EdgeFilter* parameters.

The **filter** function is outlined below:

```
filter (InputGraph, VertexFilter, EdgeFilter );
```

The *VertexFilter* and *EdgeFilter* expression define constraints for the graph elements that will be included in the result sub-graph. The *VertexFilter* and *EdgeFilter* expressions are used in other functions, in addition to the **filter** function.

VertexFiler and *EdgeFilter*

```
VertexFilter ::= [not] vertex VertexSpec
EdgeFilter ::= [not] edge EdgeSpec
VertexSpec ::= "{" any | EntityDefList "}"
EdgeSpec ::= "{" any | none | EntityDefList "}"
EntityDefList :: = EntityDef ("," EntityDef )*
EntityDef ::= Ident [WhereExpression]
```

The *EntityDefList* defines a list of vertices or edges that are combined by a logical OR. For example

```
   vertex {person, location, plot}
```

means a "person" OR a "location" OR a "plot" vertex.

Both the *VertexFilter* and *EdgeFilter* arguments can be defined by the reserved word **any** (meaning any edge or any vertex).

Each vertex or edge type may have an associated where expression that references the key attributes. For example:

```
vertex { person
         where name = "Tirza M. Kaplan" and
               birth_city = "San Francisco" and
               birth_date = "1962-04-25",
         organization
         where name = "Southern California Institute of Architecture"}
```

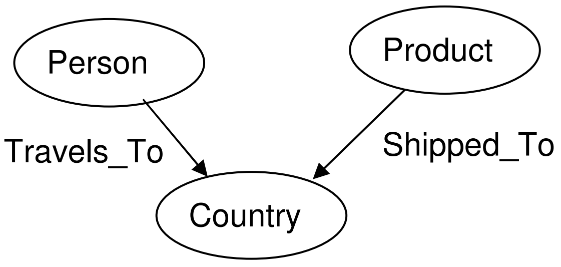
The where clause may also include expressions on the vertex degree:

```
vertex { person where degree_in < 16 }
```

If the *EdgeFilter* is defined by the reserved word **none**, no edges will be returned. A **filter** query with an *EdgeFilter* defied by the reserved word **none** will only return vertices. For example, the query below will return all of the vertices in the *InputGraph* of type person.

```
filter ("MyBaseGraph",
        vertex { person }, edge { none });
```

The **any** keyword may be used in a *VertexFilter* expression as well. The example that follows assumes an ontology that includes the graph below:



A portion of an ontology

The **filter** query below will return any vertex that is connected to a `travels_to` or a `shipped_to` edge. In the case of this ontology, the result graph will consist of a graph of country, person, and product vertices, connected by "travels_to" and "shipped_to" edges.

```
filter ("MyBaseGraph",
        vertex { any },
        edge { travels_to, shipped_to });
```

Note that the sub-graph returned by the filter function will only contain edges that are connected to vertices that are defined in the *VertexFilter*. In the query below, if there are no edges that connect shipment vertices to other shipment vertices, the query will return a graph that consists of shipment vertices, without edges.

```
filter ("MyBaseGraph",
        vertex { shipment }, edge { any });
```

If the ontology defines an edge that connects “person” vertices and “shipment” vertices, then the **filter** function below would return all “person” and “shipment” vertices, plus any edges that connected them (note that the filter function is not directional, the edge could be from a “person” vertex to a “shipment” vertex or from a “shipment” vertex to a “person” vertex.

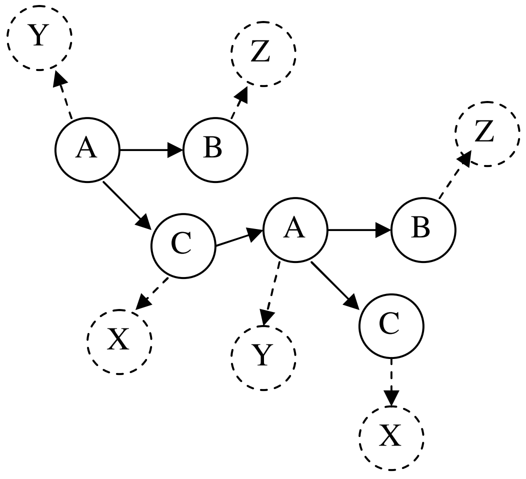
```
filter (“MyBaseGraph”,
        vertex {person, shipment} edge { any });
```

Adjacency

The **adjacency** function is outlined below:

```
                                        adjacency filter arguments
adjacency (UniverseGraph, GrowGraph, VertexFilter, EdgeFilter);
```

The **adjacency** function grows the *GrowGraph* argument by adding vertices that are one edge away from the vertices in the *GrowGraph*. In the example below, the *GraphGraph* is composed of three types of vertices: A, B and C. The **adjacency** function adds vertices of types X, Y and Z (connected by any edge) to the *GrowGraph*.



The queries below might be used to calculate the adjacency in the diagram above. The **filter** query selects a graph composed of vertices of type A, B or C. This graph is associated with the graph name “growGraph”. This graph is then used as the *GrowGraph* argument to the **adjacency** function. The result is the original *GrowGraph*, plus the new vertices and edges.

```
“growGraph” = filter (“AlphabetGraph”,
                 vertex {A, B, C} edge { any });

adjacency(“AlphabetGraph”, “growGraph”,
     vertex{X, Y, Z}, edge{ any } );
```

The query language allows functions to be used as arguments to other functions. Instead of using a named graph for the *GrowGraph* argument in the **adjacency** function, the **filter** function could be used as the argument. This is shown in the example below:

```
adjacency(“AlphabetGraph”,
         filter (“AlphabetGraph”,
              vertex {A, B, C} edge { any }),
     vertex{X, Y, Z}, edge{ any } );
```

The graph that results from adding vertices and edges to the *GrowGraph* must be contained within the *UniverseGraph*. If the *GrowGraph* and its adjacent vertices and edges lie outside of the UniverseGraph, the **adjacency** function will return the empty graph.

The **adjacency** function can be thought of as being logically composed to two operations:

1. Add the adjacent vertices and edges defined by the vertex and edge filters to the *GrowGraph* argument.
2. Intersect the expanded *GrowGraph* with the *UniverseGraph* to produce the final result. Only those vertices and edges that fall within the *UniverseGraph* will be included in the result.

The *VertexFilter* and *EdgeFilter* arguments may have the value **any**, which indicates that any vertex or edge can be included in the adjacency shell that is added to the *GrowGraph*.

Syntactically the *EdgeFiler* is also allowed to have the value **none**, but this will result in a semantic error, since an adjacency without any edges makes no sense.

The example below will find all of the vertices and edges that are adjacent to the vertices of type `person` in the graph.

```
“PersonAdj” = adjacency (“MyBaseGraph”,
                           filter (“MyBaseGraph”,
                              vertex {person}, edge { none }),
                          vertex {any}, edge {any});
```

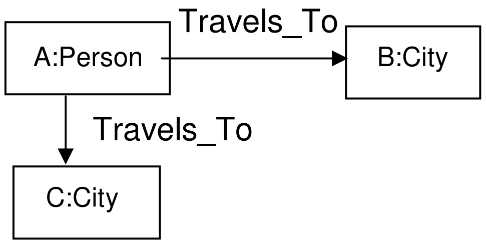
The **adjacency** function adds to the *GrowGraph*. It does not filter it. The vertices and edges in the *GrowGraph* do not have to be included in the vertex and edge filter arguments to be included in the function result.

The *VertexFilter* and *EdgeFilter* arguments in the **adjacency** function are used to control which edges and vertices are added to the *GrowGraph* argument. For example, the query below will return two vertices of type `person`, with edges of type `lives_in` connecting to vertices of type `city` (i.e., the cities that Borroughs and Warhol lived in).

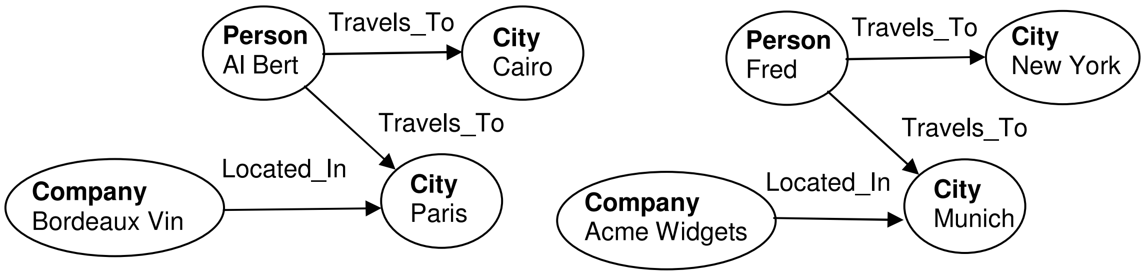
```
adjacency ("MyBaseGraph",
           filter ("MyBaseGraph",
                     vertex { person where name = "William S. Burroughs",
                              person where name = "Andy Warhol" }
                     edge {none}),
            vertex { city }
            edge { lives_in } );
```

Pattern Queries

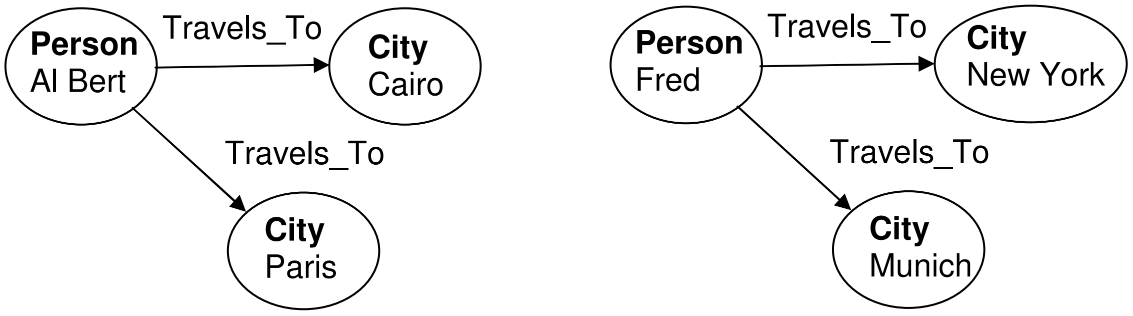
A semantic graph can be searched for patterns that may reveal important information. An example of a pattern is shown below (patterns are shown with square boxes, the graph instance data is shown with ovals). In this pattern only the types for the vertices and edges are specified. However, attribute constraints could also be included (each vertex is defined by a *VertexSpec*).



A base graph might include the set of graphs shown below:



When the pattern above is matched against these graphs, there would be the following results:



Note that the "Company" vertices were not returned, since they were not included in the pattern definition.

Pattern Query Syntax

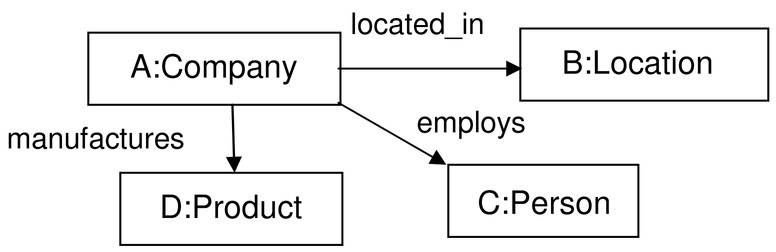
The pattern query is outlined below:

```
PatterFunction: pattern “(“ UniverseGraph “,” PatternEdgeSpec “)” “;”

PatternEdgeSpec: PatternLinkDef (“,” PatternLinkDef )*

PatternLinkDef: PatternVertex EdgeFilter PatternVertex

PatternVertex: Ident “:” (any | EntityDef)

EdgeFilter ::= [not] edge EdgeSpec

VertexSpec ::= “{“ any | EntityDefList “}”

EdgeSpec ::= “{“ any | none | EntityDefList “}”

EntityDefList :: = EntityDef (“,” EntityDef )*

EntityDef ::= Ident [WhereExpression]
```

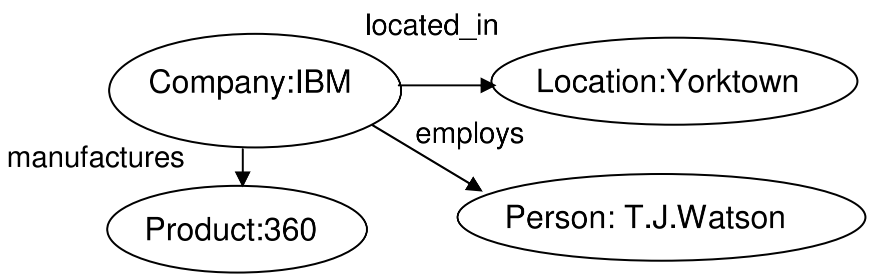
An example of a pattern query is shown below:

```
pattern( “MyBaseGraph”,
         A:company edge{ located_in } B:location,
         A:company edge{ employs } C:person,
         A:company edge{ manufactures } D:product );
```

This pattern is show in the diagram below:



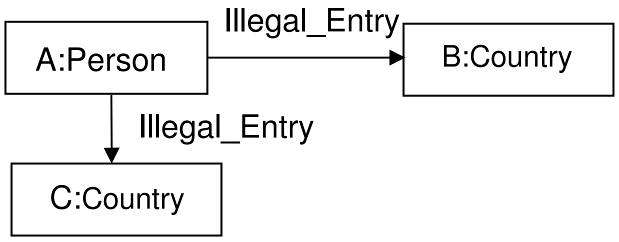
An example of a graph pattern that would match this query is shown below:



Note that the link definitions in the pattern are joined by an implicit *and* operation. The pattern above is composed of:

```
A:company edge{ located_in } B:location
```
and
```
A:company edge{ employs } C:person.
```

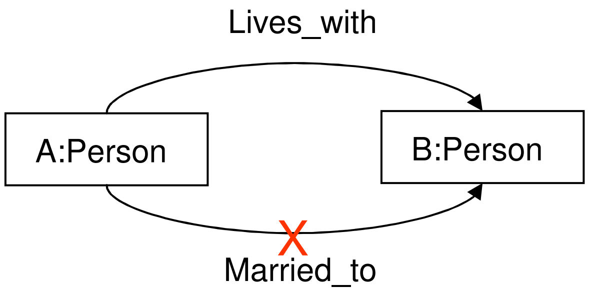
The identifiers that are associated with each vertex type in the pattern specification define an instance of that type in the graph. A graph might have many unique instances of the type `person`. In a pattern `A:person` and `B:person` indicate two different person vertices. In the pattern below a single Person vertex is joined to two Country vertices by "Illegal_Entry" edges. In this example, the Country vertices are different. A pattern like this might be used to find people who have illegally entered two different countries.



The pattern query would be:

```
 pattern (“MyBaseGraph”,
          A:Person, edge{ Illegal_Entry } B:Country,
          A:Person, edge{ Illegal_Entry } C:Country );
```

In some cases a pattern query needs to be able to specify that a link does not exist. For example, if we want to find all of the people who live together but are not married, a query like the one diagrammed below could be used.

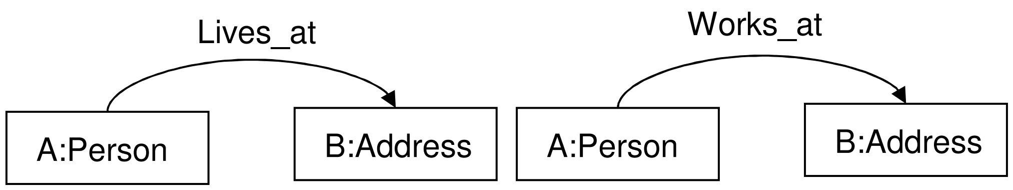


The *X* over the edge in the diagram indicates that the edge that should not be included. The pattern query is shown below:

```
 pattern (“MyBaseGraph”,
           A:Person edge{Lives_with} B:Person,
           A:Person not edge{Married_to} B:Person );
```

Note that the identifiers A and B specify the same person in each of the two links.

There could also be cases where the objective of a query would be to return a pattern that has a link of type Lives_at or a link of type Works_at, as shown in the diagram below:



The pattern query is:

```
pattern (“MyBaseGraph”,
         A:Person, edge{Lives_at, Works_at} B:Address );
```

The edge expression in this pattern is similar to the edge filter used in the **filter** and **adjacency** functions. The edge expression combines edge definitions with an implicit *or* operation.

Because of the edges that are joined by an *or* operation, the pattern function above specifies three patters:

1. A:Person *Lives_at* B:Address
2. A:Person *Works_at* B:Address
3. A:Person *Lives_at* B:Address, A:Person *Works_at* B:Address

Annotating the Graph

Queries specified in the graph query language can be submitted from a language like Java. A result set from one query can be operated on by another query. This allows complex graph algorithms to be implemented, where control takes place in Java. Some of these algorithms become easier (or practical) when the graph can be annotated with information. For example, an algorithm that calculates paths might use successive adjacency operations and mark each adjacency shell as it’s calculated. The new adjacency shell can be calculated as those vertices that are connected to the current shell but are unmarked. This forces the adjacency shell to go only one way when there are bidirectional links.

Annotation will be added to a future version of the query language, once the base language has been implemented.

Support for Query Execution

This section provides a brief overview of query language. The focus is on the primitive data objects and operations needed to execute the query language in a parallel environment.

A SIMD View of Query Language Support

In most cases, the operations that support the query language described above are Single Instruction, Multiple Data (SIMD) operations, where the definition of “instruction” is broadly defined to include high level operations. A distributed graph engine would implement a common set of operations. In most cases the query execution control would send commands to the processors to execute the same operation on each processor.

Objects used to represent intermediate results

Although the final set of graphs returned by a query may not be large, the intermediate steps taken in processing the query may process huge amounts of graph data across the distributed system. Intermediate results are left on the distributed processors. Only when the query has been processed is the final result returned to the client.

As discussed above, the input of a query is a named graph. This graph is filtered by the query, resulting in another named graph. The base graph is just another instance of a named graph. This means that the result of a query sub-expression should be stored in an instance of the same objects used to store the larger base graph. This allows a query to be executed against either the base graph or a sub-graph.

Edge Table

The connections between the vertices on a local processor and the off processor connections are represented by an edge table which exists for named graph. A row in the edge table is shown below:

edge_id	**edge_type**	**src_id**	**src_type**	**src_loc**	**dest_id**	**dest_type**	**dest_loc**

Here the src_id and dest_id are the cryptographic hashes for the source and destination vertices. The src_loc and dest_loc are the locations for these vertices (where one location is "local"). When there is an out-going or in-coming edge from one distributed processor to another, there will be a dest_loc or src_loc that is not the local system.

Vertex Table

Every edge table for a named graph is paired with a vertex table. The vertex table consists of a vertex identifier and a vertex type. A row in this table is shown below:

vertex_id	**vertex_type**

Note that the attributes of a vertex are not stored in the vertex table, but in a table that is associated with each vertex type.

Distributed Graph Operations

Vertex Select

The simplest graph operation is a vertex select. A vertex select is very similar to an SQL select operation: "select * from *table* where *column_expression*". A vertex select will usually be a sub-expression in a larger query. Each processor in the distributed system will execute the vertex select on its local portion of the graph. The result will be placed in a vertex object, which will then be used in higher level query sub-expressions.

Adjacency

The adjacency query outlined above can be implemented using three operations:

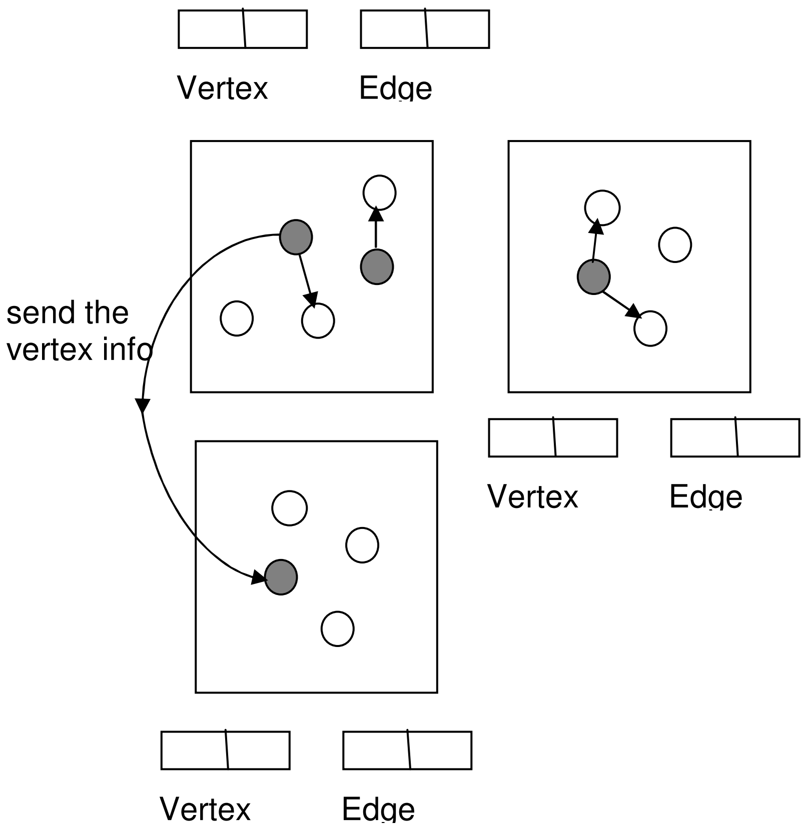
1. Calculate the adjacencies for each vertex in the *GrowGraph*.

   The result of this calculation will be a vertex table that contains the center vertex and the adjacency vertices on each processor. The edge table will consist of the edges that join the center vertex and the adjacent vertices, on that processor, plus the edges to external processors.

   Before the adjacencies can be calculated, information about the *GrowGraph* vertices must be sent to the distributed processors. There are two ways this could be done:

   1. The information about the *GrowGraph* vertices could be sent to every distributed processor. This would require collecting the information about the *GrowGraph* and then broadcasting it to each processor. This has the drawback of bottlenecking the calculation on the "gather" operation.
   2. A processor with one or more *GrowGraph* vertices can discover from the edge table which processors host the adjacent vertices. The information about the *GrowGraph* vertices can be sent selectively to these processors. The distributed nature of this operation makes this an attractive approach and this is the method proposed here.

   For an edge that crosses between processors, send the *GrowGraph* vertex to the remote processor. The remote processor will build a local vertex and edge table. Note that we only need to send the vertex ID and type, since the remote processor has an edge table that records which local vertices have incoming edges for that center.



3. Filter the vertices and edges on the basis of the vertex and edge filters. This operation can be done in SIMD fashion on each of the processors, filtering the vertex and edge tables.

Acknowledgements

Like the platypus, this query language has gone through many evolutionary steps. The language structure has been influenced by comments from the Complex Networks group, lead by Scott Kohn.