A Visual Language for Querying and Updating Graphs

H. Blau N. Immerman D. Jensen

{blau, immerman, jensen}@cs.umass.edu
Department of Computer Science, University of Massachusetts, Amherst, MA 01003-9264

Abstract

QGRAPH is a new visual language for querying and updating graph databases. In QGRAPH the user can draw a query consisting of some vertices and edges with specified relations between their attributes. The response will be the collection of all subgraphs of the database that have the desired pattern. QGRAPH is very useful for knowledge discovery. QGRAPH has a powerful and elegant counting feature that enables the user to easily specify how many of certain objects and links should exist in order for a subgraph to match a query. QGRAPH has a clean formal semantics which we describe in detail. We show that QGRAPH has expressive power corresponding to a well-defined subset of FO(COUNT), i.e., first-order logic with counting quantifiers.

1 Introduction

QGRAPH is a new visual language for querying and updating graph databases. A key feature of QGRAPH is that the user can draw a query consisting of vertices and edges with specified relations between their attributes. The response will be the collection of all subgraphs of the database that have the desired pattern.

QGRAPH is a full-fledged query and update language which can create new objects and links and can update the attributes of existing objects and links wherever a subgraph matching a given query pattern occurs. QGRAPH has a powerful and elegant counting feature (numeric annotation) that enables the user to easily specify how many of certain objects and links should exist in order for a subgraph to match a query. QGRAPH is designed to apply to graph databases with multiple attributes attached to objects and links. Each attribute consists of a set of values.

QGRAPH has a clean formal semantics which we describe in detail (Section 3). When queries are written using such a simple, intuitive tool it is crucial to give precise semantics, so that the interpretation of a query does not depend on possibly diverging intuitions. We show that QGRAPH has expressive power corresponding to a well-defined subset of FO(COUNT), i.e., first-order logic with counting quantifiers. QGRAPH bears some relation to another visual query language, GraphLog, which, in the presence of ordering, has query expressibility $\text{FO(TC)} = \text{NSPACE}[\log n]$ [CM]. There are also weaker relationships with languages designed to query XML, or semi-structured data [AQM, ABS].

We have designed QGRAPH to be useful for knowledge discovery and data mining in large graph databases. Our knowledge discovery algorithms [JN, NJ] construct probabilistic models of the dependencies among the attributes of objects and links in a local neighborhood. We also wish to support the ad hoc exploration of databases that is essential for effective knowledge discovery in practical applications. These uses are enabled because QGRAPH provides numerical annotations, returns complex objects in response to queries, and admits very efficient query evaluation.

In Section 5 we discuss some future work in which we plan to optimize QGRAPH query evaluation given statistical information about the database. We discuss plans to allow the QGRAPH interface to estimate the size of matches and time needed to produce them. We will also consider adding stronger features such as transitive closure in this context, i.e., with warnings to the user when the processing of a query might be prohibitive.

2 Language description

A QGRAPH query is a labeled graph in which the vertices correspond to objects and the edges to links. We use the terms *vertex* and *edge* when referring to the query, *object* and *link* when referring to the database. The query specifies the desired structure of vertices and edges. It may also place boolean conditions on the attribute values of matching objects and links, as well as global constraints relating one object or link to another. Each vertex and edge of a QGRAPH query has a unique label. The query must be a connected graph.

A query consists of *match* vertices and edges and optional *update* vertices and edges. The former determine which subgraphs in the graph database constitute a match for the query. The latter determine what modifications are made

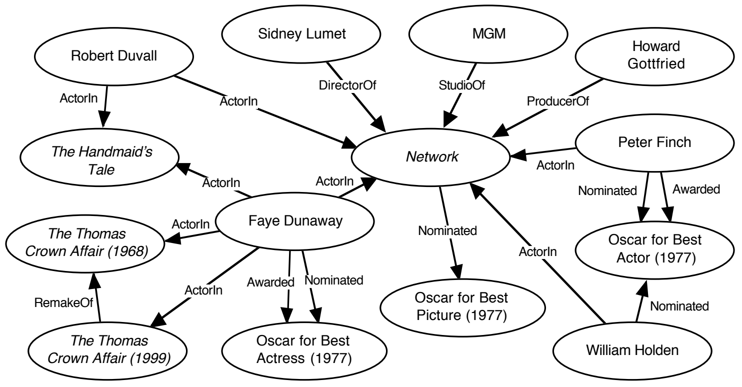

Figure 1: Graphical data fragment from a movie database

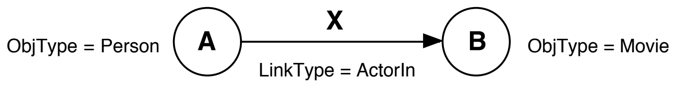

Figure 2: Find all `Person`, `ActorIn`, `Movie` subgraphs

to the matching subgraphs. A query with only match vertices and edges serves to identify and display a collection of subgraphs. To match the query, a subgraph must have the correct structure and satisfy all the boolean conditions and constraints. A query with both match and update vertices and edges can be used for attribute calculation and for structural modification of the database. The query processor first finds the matching subgraphs using the query's match elements, then makes changes to those subgraphs as indicated by the query's update elements.

Figure 1 is an example of the graph databases for which we designed QGRAPH. Our database consists of objects, binary links, and attributes that record features of the object or link. An object or link can have zero or more attributes. All attributes are set-valued. For example, a person can have multiple names. The figure shows a fragment from a database about movies. The labels on the objects indicate their name, and the labels on links indicate their type. Not shown in the figure are other attributes of objects, such as the year a movie was released or the location of a studio. Similarly, links could have attributes, such as the salary an actor received for starring in a given movie.

2.1 Conditions

The query in Figure 2 finds all subgraphs with an `ActorIn` link between a `Person` and a `Movie`. The type restrictions are expressed by *conditions* on the two vertices and one edge of the query. In this example only one attribute is tested in each condition; in general a condition can be any boolean combination of restrictions on attribute values.

A, **B**, and **X** are unique labels assigned to each vertex and edge in the query. We use letters at the beginning of the alphabet for vertices, and those from the end of the alphabet for edges. The labels have no intrinsic meaning and do not indicate anything about the type of object or link that would match the labeled element. Where desired, type restrictions are enforced with conditions on vertices and edges.

For the sample database of Figure 1, this query produces 8 matches (Figure 3). Unlike the SELECT statement in SQL, a QGRAPH query does not specify which attributes of matching objects and links should be included in the result. Evaluating a QGRAPH query returns a collection of all the matching subgraphs from the database. The user can examine any subgraph in the resulting collection, and any object or link in that subgraph, with the user interface. All the object and link attributes, not just those mentioned in the query conditions, are available for inspection.

2.2 Numeric annotations

To group the actors together for each movie, we add a *numeric annotation* to the `Person` vertex (Figure 4). Executing this query against the database produces 4 matches (Figure 5), one for each movie, compared with 8 matches for the

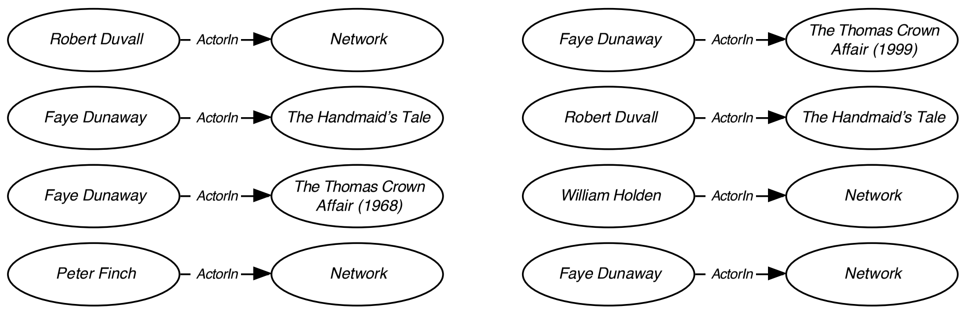

Figure 3: Matches for query in Figure 2

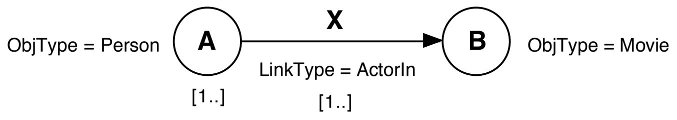

Figure 4: For each movie, find all its actors

query without the numeric annotation (Figure 3). A numeric annotation can be specified on a vertex or an edge of a QGRAPH query. (We will see in Section 2.6 that a subquery can also have a numeric annotation.) A numeric annotation takes one of three forms. An *unbounded range* $[i..]$ on a vertex (or edge) means at least i instances of the annotated object (or link) must be present in any matching database fragment. A *bounded range* $[i..j]$ means at least i and no more than j instances are required for a match. An *exact* annotation $[i]$ means exactly i instances are required. i can be any integer ≥ 0; j can be any integer $> i$. If the lower end of the possible range is 0, the annotated structure is optional in any matching database fragment. (The annotation $[..j]$ is not allowed because it would be ambiguous between $[0..j]$ and $[1..j]$.) The annotation [0] on a vertex (or edge) indicates negation: to match the query, a database fragment must *not* contain the corresponding object (or link). To be well-formed, a query must remain a connected graph when any optional or negated structures (annotations [0], [0..], or $[0..n]$) are removed. To avoid ambiguities of interpretation, only one of any two adjacent vertices can be annotated.

A numeric annotation serves two purposes in a query. It groups together into one match repeated isomorphic substructures that would otherwise create multiple matches for the query (compare Figures 3 and 5). It places limits on how many such structures can occur in matching portions of the database. To group the substructures without limiting their number, we use the annotation [1..] (as in Figure 4). There is no mechanism in QGRAPH to limit the number of matching substructures without grouping them together. If we changed the annotation on the vertex **A** in

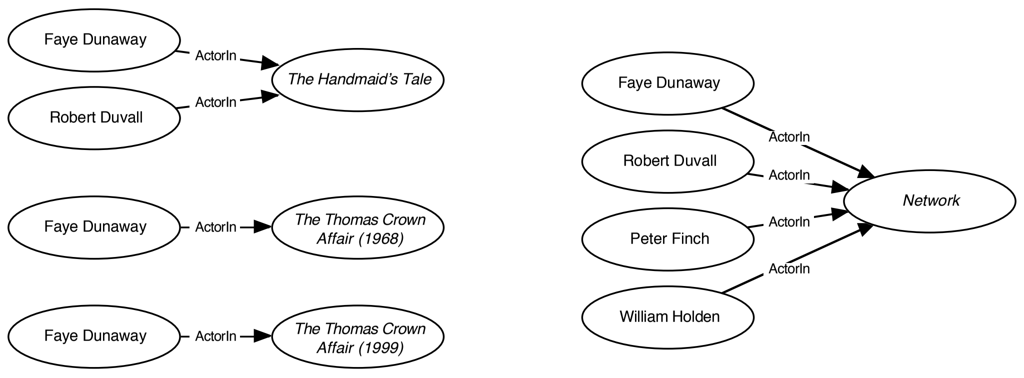

Figure 5: Matches for query in Figure 4

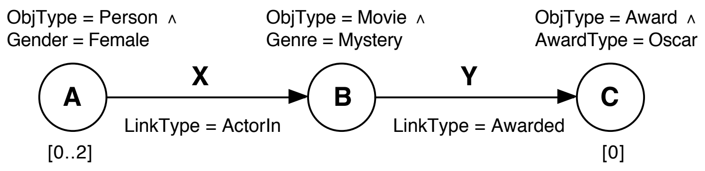

Figure 6: Mysteries with fewer than 3 female actors and no Oscar awards

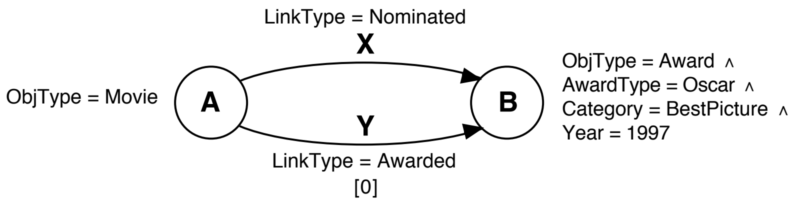

Figure 7: Movies nominated for Best Picture in 1997 that did not win

Figure 4 to be [1..2] instead of [1..], then the subgraph on the right-hand side of 5 would no longer be a match for the query. The subgraphs on the left-hand side would still be matches.

The edge **X** of Figure 4 also has an annotation [1..]. An edge incident to an annotated vertex must itself be annotated. The annotation on the vertex takes precedence over the annotation on the edge. We first find all the actors for a specific movie, then for each of those actors we find all the `ActorIn` links that connect the actor to the movie. For example, an actor who played multiple roles in a particular movie might have multiple `ActorIn` links to the same `Movie` object. The annotation [1..] groups all these links into a single match. To avoid clutter in the following examples, we have omitted the annotation [1..] from the edges adjacent to annotated vertices. The annotation [1..] is implicit unless some other annotation is specified on the edge.

The query of Figure 6 selects mystery movies that never received an Oscar and have fewer than three female actors. A movie that has won no awards at all, or has won awards that are not Oscars, could match this query. The movie *Sleuth* (1972) is a match. *Sleuth* had only one female actor (Eve Channing) and won no Oscars, although it did win an Edgar Allan Poe Award and a New York Film Critics Circle Award. If we wanted only movies that have won no awards at all, we would drop the conjunct `AwardType` = `Oscar` from the condition on node **C**.

A negated element (annotation [0]) does not show up in the results of a query, because a subgraph matches the query only if it has no object (or link) matching the negated vertex (or edge). For the query of Figure 6, no `Award` objects or `Awarded` links would appear in the results. `Person` objects and `ActorIn` links would appear only in matches for movies that had exactly one or two female actors, such as *Sleuth*. They would not appear in matches for movies that had no female actors.

The query of Figure 7 selects movies that were nominated for the Best Picture Oscar in 1997 but did not win. This query illustrates a numeric annotation on a link. The movies *As Good as It Gets*, *The Full Monty*, *Good Will Hunting*, and *L.A. Confidential* match this query.

2.3 Projecting over subgraph structure

For many queries, the user does not need to see the entire matching subgraph. For the query of Figure 7, there is no need to include the `Award` object and the `Nominated` link in every subgraph of the resulting collection. The focus of interest is the movie. To see only `Movie` objects in the results, we highlight vertex **A** in the query (leaving the other vertex and the edges unhighlighted). This highlighting is analogous to the projection operator in relational algebra. In QGRAPH, we project over structures by highlighting the elements that interest us. Highlighting does not change how the query is evaluated against the database. It changes how the matching subgraphs are displayed. Only those objects and links that match highlighted vertices and edges are displayed.

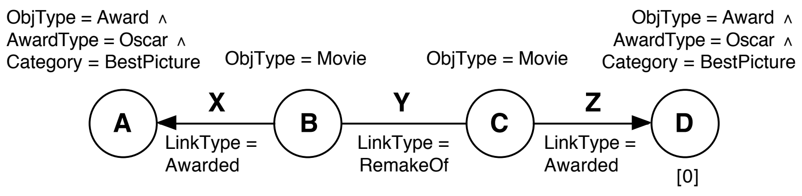

Figure 8: {Remake, original} pairs where one won Best Picture and the other did not

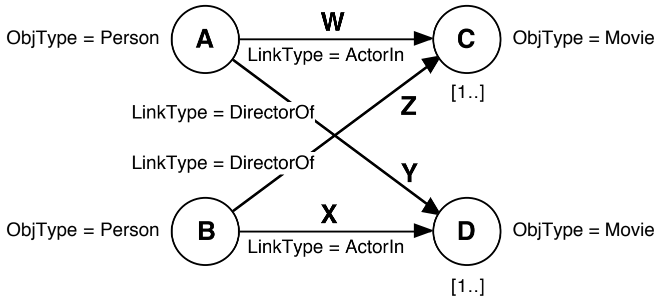

Figure 9: Pairs of people such that each has acted in movies directed by the other

2.4 Undirected edges

The data model underlying QGRAPH is a directed graph; it has no undirected links. Nevertheless, QGRAPH allows undirected edges for queries in which we do not know, or choose to ignore, the directionality of the relationship. For example, in the movie database the `RemakeOf` link goes from a new remake to the older original. Suppose we want to find {remake, original} pairs such that one of the two movies received an Oscar for Best Picture while the other did not. This query can be succinctly expressed with an undirected `RemakeOf` edge between the two `Movie` vertices (Figure 8). The silent classic *Ben-Hur* (1925) and the 1959 remake starring Charlton Heston match this query. The 1959 film won the Oscar for Best Picture; the original predated the Oscar awards.

2.5 Constraints

The query of Figure 9 selects pairs of people such that each has acted in one or more movies directed by the other. This query matches the database fragment shown in Figure 10. Burt Reynolds directed *The End* (1978) in which David Steinberg acted, and Steinberg directed *Paternity* (1981) in which Reynolds acted.

This query also matches any director who has acted in his own movies. Multiple vertices of a query can match a single database object provided the object satisfies the conditions on all the vertices. Likewise two or more edges having the same start- and endpoints can match a single link in the database. In the case of an actor-director, the vertices A and B match the same `Person`, C and D the same `Movie`, W and X the same `ActorIn` link, Y and Z the same `DirectorOf` link. For example, this query would match John Sayles and all the films he both directed

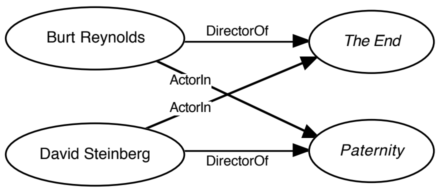

Figure 10: Database fragment for Burt Reynolds and David Steinberg

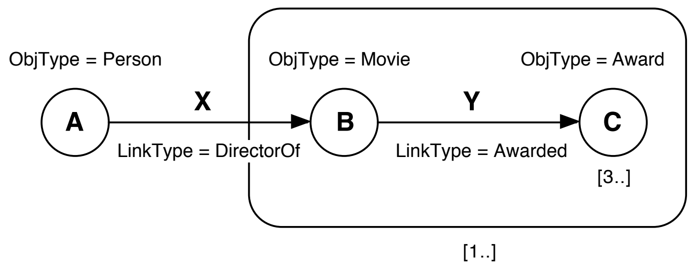

Figure 11: Directors of movies that have won three or more awards each

and appeared in: *Return of the Secaucus 7* (1980), *Lianna* (1983), *The Brother from Another Planet* (1984), *Matewan* (1987), *Eight Men Out* (1988), *City of Hope* (1991), *Passion Fish* (1992).

To eliminate the actor-director matches, we add two inequality *constraints* to the query: $\mathtt{A} \neq \mathtt{B}$ and $\mathtt{C} \neq \mathtt{D}$. (Inequality constraints on the vertices force the edges to be distinct as well.) Inequality constraints are necessary whenever we want to ensure that two vertices (or edges) map to distinct database objects (or links), unless the conditions on the two query elements are incompatible anyway. In addition to inequality constraints, a constraint can relate attribute values of one object or link to those of another in the matching subgraph. For example, suppose the `ActorIn` link has a `Salary` attribute recording the amount the actor earned for that appearance. With constraints, we can compare the salaries of two different actors, or the salaries of the same actor for two different movies.

Both conditions and constraints restrict the matches to a query. Conditions on a vertex (or edge) involve only the attributes of the corresponding object (or link). Constraints relate one vertex (or edge) of the query to another vertex (or edge), by asserting that the two are distinct or by comparing their attribute values. No inequality or other constraint is allowed between two vertices that both have numeric annotations, for the same reason that two vertices joined by an edge cannot both be annotated. Likewise no constraint may mention two annotated edges. However, a constraint may mention an annotated vertex and an annotated edge that is incident to that vertex.

2.6 Subqueries and Union

A subquery is a connected subgraph of vertices and edges that can be treated as a logical unit. It has one or more edges that leave the subquery box and attach the subquery to some vertex or vertices of the main query (or a higher level subquery). A subquery enables the user to attach a numeric annotation to a connected group of vertices and edges, instead of just a single vertex or edge. A query or subquery may also be written as the union of two or more queries as long as they all have identically named unannotated vertices and edges.

Figure 11 shows a query that finds people who have directed very successful movies, where a movie is considered "very successful" if it has won three or more awards. The numeric annotation [1..] on the subquery box will group together all the successful movies for a given director into one match for the query. Without the subquery box, one match would be returned for each successful movie of each director. The director Steven Spielberg matches this query. His very successful movies include *Raiders of the Lost Ark* (1981), 4 Oscars; *E.T. the Extra-Terrestrial* (1982), 4 Oscars; *Jurassic Park* (1993), 3 Oscars; *Schindler's List* (1993), 7 Oscars; and *Saving Private Ryan* (1998), 5 Oscars. The entire subgraph shown in Figure 12 constitutes one match for the query in Figure 11.

2.7 Data transformation

In addition to its convenient features for data extraction, QGRAPH is a flexible language for data transformation in graph databases. We can add new objects, links, and attributes, or delete existing ones. Conceptually, QGRAPH query processing comprises two phases: match and update. The match phase determines which subgraphs of the database are selected by the query's match vertices and edges (with their associated conditions, constraints, and numeric annotations). The update phase performs all indicated updates in parallel to the selected subgraphs. Applying one update cannot create a new match for another update within the same query.

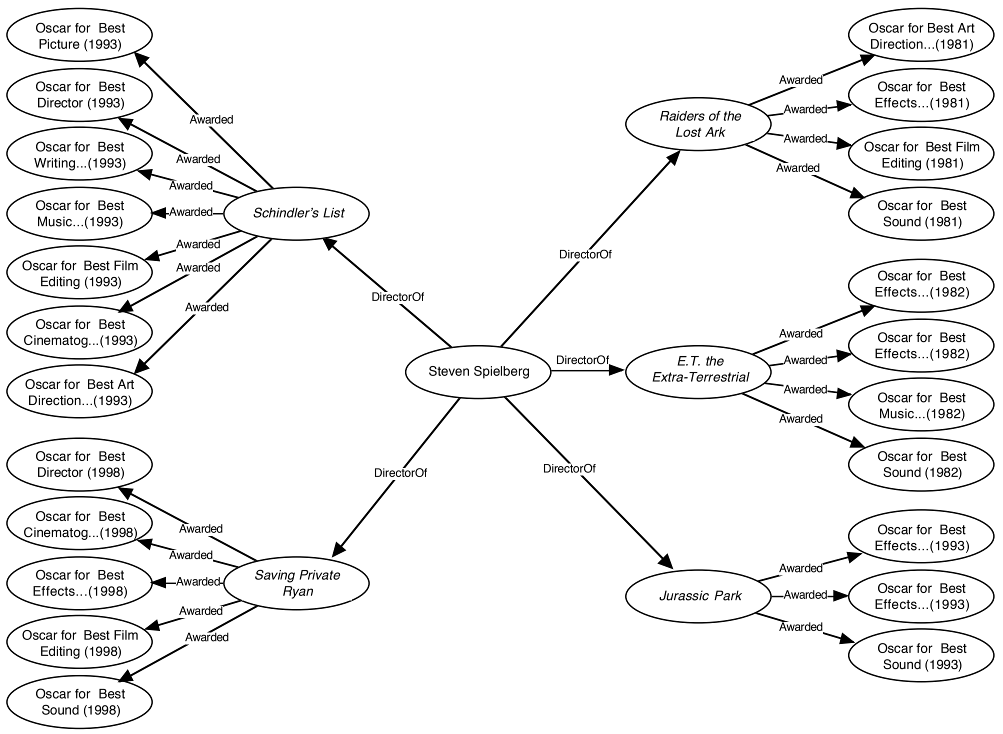
Steven Spielberg
DirectorOf
Schindler's List
Saving Private Ryan
Raiders of the Lost Ark
E.T. the Extra-Terrestrial
Jurassic Park
Awarded
Oscar for Best Picture (1993)
Oscar for Best Director (1993)
Oscar for Best Writing...(1993)
Oscar for Best Music...(1993)
Oscar for Best Film Editing (1993)
Oscar for Best Cinematog...(1993)
Oscar for Best Art Direction...(1993)
Oscar for Best Director (1998)
Oscar for Best Cinematog...(1998)
Oscar for Best Effects...(1998)
Oscar for Best Film Editing (1998)
Oscar for Best Sound (1998)
Oscar for Best Art Direction...(1981)
Oscar for Best Effects...(1981)
Oscar for Best Film Editing (1981)
Oscar for Best Sound (1981)
Oscar for Best Effects...(1982)
Oscar for Best Effects...(1982)
Oscar for Best Music...(1982)
Oscar for Best Sound (1982)
Oscar for Best Effects...(1993)
Oscar for Best Effects...(1993)
Oscar for Best Sound (1993)

Figure 12: A match for query in Figure 11

2.8 Conditions and updates on set-valued attributes

In the QGRAPH data model, all attributes are set-valued. In many cases, the semantics of the domain represented by the database constrain the values of some attributes to be singleton sets. In the movie database, the `ObjType` attribute is a singleton set: no object is both a person and a movie, or an award and a production studio. But in other domains, a single object might have several different `ObjType`s.

The notation $\texttt{attribute} = \texttt{value}$ is shorthand for $\texttt{value} \in \texttt{values(attribute)}$. Likewise, $\texttt{attribute} \neq \texttt{value}$ is shorthand for $\texttt{value} \notin \texttt{values(attribute)}$. Note that any object or link for which `attribute` is undefined (that is, $\texttt{values(attribute)} = \emptyset$) satisfies the condition $\texttt{attribute} \neq \texttt{value}$. If these matches are undesirable, we can eliminate them with a compound condition that first tests if the attribute is defined: $\texttt{(attribute} \neq \texttt{EMPTY} \land \texttt{attribute} \neq \texttt{value)}$.

Because attributes are set-valued, QGRAPH provides for three types of attribute updates:

- replace the existing values of the attribute with the new value, written `attribute := newValue` which means $\texttt{values(attribute)} \leftarrow \{\texttt{newValue}\}$
- add the new value to the existing ones, written `attribute += newValue` which means $\texttt{values(attribute)} \leftarrow \texttt{values(attribute)} \cup \{\texttt{newValue}\}$
- remove an existing value from the set, written `attribute -= oldValue` which means $\texttt{values(attribute)} \leftarrow \texttt{values(attribute)} - \{\texttt{oldValue}\}$

We can add, remove, or replace multiple values at once. For example,
`attribute := newValue1, newValue2, newValue3` means
$\texttt{values(attribute)} \leftarrow \{\texttt{newValue1, newValue2, newValue3}\}$.
To remove all values for an attribute, set it to empty: `attribute := EMPTY`
which means $\texttt{values(attribute)} \leftarrow \emptyset$.

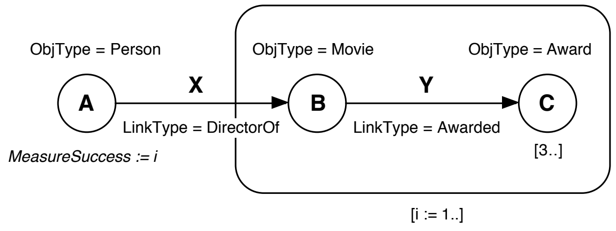

Figure 13: Add measure of success as attribute of director

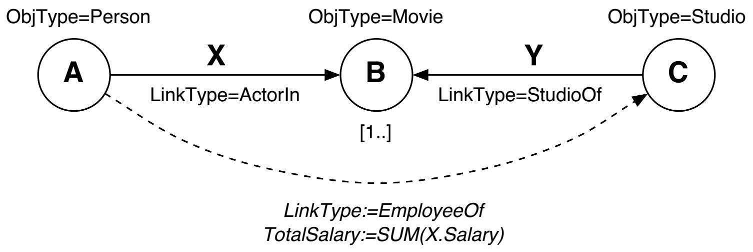

Figure 14: Add link from actor to studio with total salary

2.9 Counter variables in attribute updates

Figure 13 shows a variation on the query from Figure 11 in which we store the number of very successful movies as a new attribute of the director. The numeric annotation $[i := 1..]$ on the subquery box illustrates the use of a *counter variable* that is set to the number of matches for the subquery. Any or all of the numeric annotations in a query may be augmented with counter variables, so long as the variable names are unique within the query. The variable i counts the number of movies by this director that have received three or more awards. This value is copied into a new attribute `MeasureSuccess` on the `Person` object. The italic font and assignment operator indicate an attribute update.

2.10 Adding a link

The query of Figure 14 creates an `EmployeeOf` link between an actor and a studio if the actor has appeared in movies made by that studio. The query calculates the total salary the actor earned from all his appearances in the studio's movies and records the figure as as an attribute of the new link. This example illustrates the use of an aggregation function to calculate the actor's total salary. Aggregation functions such as SUM, AVG, MIN, and MAX may be used in a QGRAPH constraint or attribute update. The expression `SUM(X.Salary)` calculates the sum of the `Salary` attribute for all the `ActorIn` links X connecting the actor to a movie made by the studio. The numeric annotation on the movie vertex is essential for the calculation of `TotalSalary`. The annotation groups together into one match all the movies for a given {actor, studio} pair. Without the numeric annotation, a separate link from actor to studio would be created for each {actor, movie, studio} triple, and the value of the `TotalSalary` attribute on the link would be the salary for that particular movie.

A new `EmployeeOf` link is created for each {actor, studio} pair that matches the query. The salary is summed over just the movies involving that {actor, studio} pair. If the actor has worked for several different studios, the query creates an `EmployeeOf` link to each studio with a corresponding value for the `TotalSalary` attribute. If we wanted to create the new link only in cases where the actor had earned one million dollars or more working for the studio, we would add a constraint to the query: `SUM(X.Salary)` $\geq 10^6$.

2.11 Adding an object

QGRAPH also allows the creation of new objects. To add an object to the database, we must also add one or more links to connect the new object to existing ones. Figure 15 shows how the manager of a cinema complex would create an

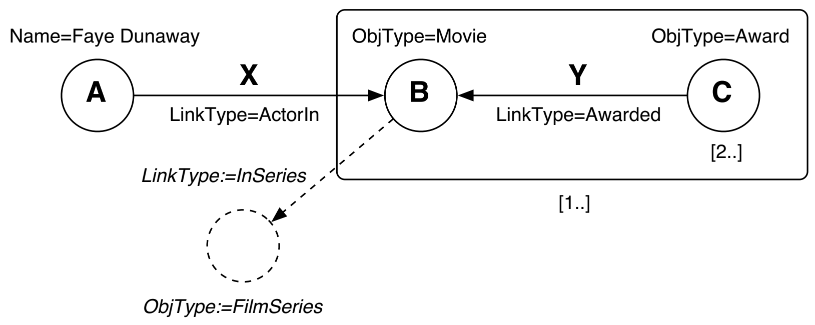

Figure 15: Film series of Faye Dunaway movies with two or more awards

object representing a film series. The manager wants to present a retrospective of Faye Dunaway's work. To weed out the less appealing films, she selects only those that received at least two awards. Each of the Faye Dunaway movies is connected to the `FilmSeries` object by a `InSeries` link. This query adds one new object but as many links as there are movies in the database that satisfy the awards criterion. If we augmented the database fragment of Figure 1 with award information for all its movies, evaluating this query would result in a film series with three items: *Network*, and both versions of *The Thomas Crown Affair* (1968 and 1999). *The Handmaid's Tale* won only one award.

3 QGRAPH semantics

We now present a complete formal semantics for the match portion of the QGRAPH language in first order logic with two sorts: objects and links. Having a formal semantics is essential for QGRAPH to be understood consistently by a wide variety of users. The diagrams of QGRAPH and other visual languages seem intuitive, but different users might have divergent intuitions and interpret the same diagram in different ways. The formal semantics of QGRAPH establishes the authoritative interpretation for each query.

We use a two-sorted logic: A, B, C are object variables; X, Y, Z are link variables. A query has

$A_1, \ldots, A_a$	unannotated vertices	$B_{1[i_1..j_1]}, \ldots, B_{b[i_b..j_b]}$	annotated vertices
$X_1, \ldots, X_x$	unannotated edges	$Y_{1[k_1..l_1]}, \ldots, Y_{y[k_y..l_y]}$	annotated edges
$Q_{1[m_1..n_1]}, \ldots, Q_{q[m_q..n_q]}$	subqueries.		

Edges that cross a subquery boundary are considered part of the subquery. These edges must be annotated because the subquery is annotated.

Higher-order semantics is the most natural way to represent the meaning of a QGRAPH query. Subqueries are defined inductively and the meaning of a subquery is the same as if it were a query all by itself. However, the answer to a QGRAPH query is a set of graphs, so the real semantics of QGRAPH is simply the flattening of the higher-order semantics. μ_h stands for higher-order semantics and μ_f for flat semantics. Here "flatten" replaces each subquery, $\mu_h(S_Q)$, recursively with the sets $S_{A_{Q,1}}, \ldots, S_{A_{Q,a_Q}}, \ldots, S_{Y_{Q,1}}, \ldots, S_{Y_{Q,y_Q}}$ of all vertices and edges that occur in any instance of $\mu_h(S_Q)$. The meaning of the top level query Q_0 is $\mu_f(Q_0) = \text{flatten}(\mu_h(Q_0))$.

$$\mu_h(Q) = \{\langle A_1, \ldots, A_a, X_1, \ldots, X_x, S_{B_1}, \ldots, S_{B_b}, S_{Y_1}, \ldots, S_{Y_y}, S_{Q_1}, \ldots, S_{Q_q}\rangle \mid \varphi(A_1, \ldots, A_a, X_1, \ldots, X_x),$$
$$S_{B_i} = \{B_i | \varphi_{B_i}(B_i)\},\ S_{Y_i} = \{Y_i | \varphi_{Y_i}(Y_i)\},\ S_{Q_i} = \mu_h(Q_i; A_1, \ldots, A_a, X_1, \ldots, X_x)\}$$

The semantics of a QGRAPH query is defined inductively as follows:

1 Add unannotated vertex A: Add A to the result tuple, and replace φ by $\varphi \wedge \rho_A(A)$. $\rho_A(A)$ is the default condition, initially just $true$.

2 Add annotated vertex $B_{[i..j]}$: Add S_B to the result tuple, and replace φ by $\varphi \wedge (\exists\, [i..j]\, B)(\varphi_B(B))$ where $\varphi_B(B) \equiv \rho_B(B)$. We use the notation $(\exists\, [i..j]\, B)(\varphi_B(B))$ to mean that the number of B's such that $\varphi_B(B)$ is between i and j.

3 Add directed, unannotated edge X from unannotated vertex A_t to unannotated vertex A_h: Add X to the result tuple, and replace φ by $\varphi \wedge \rho_X(X) \wedge tail(X, A_t) \wedge head(X, A_h)$. An undirected edge requires a trivial change. If X is an undirected edge between vertices C_1 and C_2 (annotated or not), replace $\mathit{tail}(X, C_1) \wedge \mathit{head}(X, C_2)$ by $((\mathit{tail}(X, C_1) \wedge \mathit{head}(X, C_2)) \vee (\mathit{tail}(X, C_2) \wedge \mathit{head}(X, C_1)))$ throughout the semantics. Everything we write hereafter for directed edges applies equally to undirected edges.

4 Add directed, annotated edge $Y_{[k..l]}$ from unannotated vertex A_t to unannotated vertex A_h: Add S_Y to the result tuple, and replace φ by $\varphi \wedge (\exists\,[k..l]\,Y)(\varphi_Y(Y))$ where $\varphi_Y(Y) \equiv \rho_Y(Y) \wedge \mathit{tail}(Y, A_t) \wedge \mathit{head}(Y, A_h)$.

5 Add directed, annotated edge $Y_{[k..l]}$ from unannotated vertex A to annotated vertex B: Add S_Y to the result tuple, replace φ_B by $\varphi_B \wedge (\exists\,[k..l]\,Y)(\rho_Y(Y) \wedge \mathit{tail}(Y, A) \wedge \mathit{head}(Y, B))$, and let $\varphi_Y(Y) \equiv (\exists B)(\varphi_B(B) \wedge \rho_Y(Y) \wedge \mathit{tail}(Y, A) \wedge \mathit{head}(Y, B))$. The occurrence of $\varphi_B(B)$ in the second formula refers to the new φ_B. (If Y is from B to A then just switch *tail* and *head* in the above.)

6 Replace any condition ρ_c by ρ'_c: Just replace ρ_c throughout semantics by ρ'_c.

Constraints between two annotated vertices or two annotated edges are illegal in QGRAPH. In general, a constraint can involve at most one annotated vertex or annotated edge. However, a constraint can mention an annotated vertex and an *adjacent* annotated edge. This is the only form of constraint that involves more than one annotated element.

7 Add constraint θ that does not refer to any annotated edge or vertex: Replace φ by $\varphi \wedge \theta$.

8 Add constraint θ that depends on annotated vertex B and no annotated edge: Replace ρ_B by $\rho_B \wedge \theta$.

9 Add constraint θ that depends on annotated edge Y (and perhaps on adjacent annotated vertex B): Replace ρ_Y by $\rho_Y \wedge \theta$.

10 Add subquery $Q_{[m..n]}$: Add S_Q to the result tuple, and replace φ by

$$\varphi \wedge (\exists\,[m..n]\,\langle A_{Q,1}, \ldots, A_{Q,a_Q}, X_{Q,1}, \ldots, X_{Q,x_Q}\rangle)(\varphi_Q(A_{Q,1}, \ldots, A_{Q,a_Q}, X_{Q,1}, \ldots, X_{Q,x_Q}))$$

$A_{Q,1}, \ldots, A_{Q,a_Q}, X_{Q,1} \ldots, X_{Q,x_Q}$ are the unannotated vertices and edges within the subquery Q.

11 Union of queries: If two queries (or subqueries) have the same unannotated vertices and edges, i.e., Q_1 and Q_2 with $\mu_h(Q_i) = \{\langle A_1, \ldots, A_a, X_1, \ldots, X_x, S_{B_{i,1}}, \ldots, S_{B_{i,b}}, S_{Y_{i,1}}, \ldots, S_{Y_{i,y}}, S_{Q_{i,1}}, \ldots, S_{Q_{i,q}}\rangle \mid \varphi_i\}$, then

$$\mu_h(Q_1 \cup Q_2) = \{\langle A_1, \ldots, A_a, X_1, \ldots, X_x, S_{B_{1,1}}, \ldots, S_{B_{1,b}}, S_{Y_{1,1}}, \ldots, S_{Y_{1,y}}, S_{Q_{1,1}}, \ldots, S_{Q_{1,q}}, S_{B_{2,1}}, \ldots, S_{B_{2,b}}, S_{Y_{2,1}}, \ldots, S_{Y_{2,y}}, S_{Q_{2,1}}, \ldots, S_{Q_{2,q}}\rangle \mid \varphi_1 \vee \varphi_2\}$$

If the same annotated vertex or edge, say B, occurs in both Q_1 and Q_2, then the new φ_B would be $(\varphi_1 \wedge \varphi_{B_1}) \vee (\varphi_2 \wedge \varphi_{B_2})$.

The example in Figure 16 illustrates the semantics of a simple query with a constraint. ρ_A, ρ_B and ρ_Y are conditions on A, B, and Y, respectively. The constraint θ involves both vertices and the edge.

$$\begin{gathered}\mu_h(Q_0) = \{\langle A, S_B, S_Y\rangle \mid \varphi(A),\ S_B = \{B|\varphi_B(B)\},\ S_Y = \{Y|\varphi_Y(Y)\}\} \\ \text{where } \varphi(A) = \rho_A(A) \wedge (\exists\,[i..j]\,B)(\varphi_B(B)) \\ \varphi_B(B) = \rho_B(B) \wedge (\exists\,[k..l]\,Y)(\rho_Y(Y) \wedge \theta(A, B, Y) \wedge \mathit{tail}(Y, A) \wedge \mathit{head}(Y, B)) \\ \varphi_Y(Y) = (\exists B)(\varphi_B(B) \wedge \rho_Y(Y) \wedge \theta(A, B, Y) \wedge \mathit{tail}(Y, A) \wedge \mathit{head}(Y, B))\end{gathered}$$

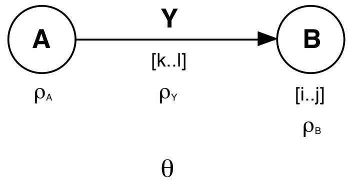

Figure 16: Simple query with constraint θ involving **A**, **Y**, and **B**

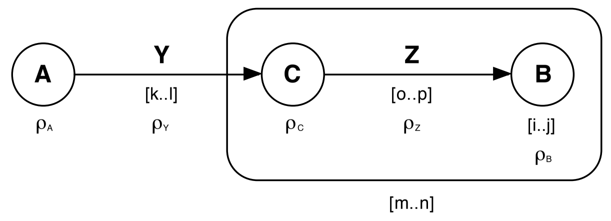

Figure 17: Query with subquery

The example in Figure 17 illustrates the semantics of a subquery.

$$\mu_h(Q_0) = \{\langle A, S_Q\rangle \,|\varphi_0(A), S_Q = \mu_h(Q;A)\}$$
$$\mu_h(Q;A) = \{\langle C, S_B, S_Y, S_Z\rangle \mid \varphi_Q(C),\, S_B = \{B|\varphi_B(B)\},\, S_Y = \{Y|\varphi_Y(Y)\},\, S_Z = \{Z|\varphi_Z(Z)\}\}$$
$$\text{where } \varphi_0(A) = \rho_A(A) \wedge (\exists\,[m..n]\,C)(\varphi_Q(C))$$
$$\varphi_Q(C) = \rho_C(C) \wedge (\exists\,[i..j]\,B)(\varphi_B(B)) \wedge (\exists\,[k..l]\,Y)(\varphi_Y(Y))$$
$$\varphi_B(B) = \rho_B(B) \wedge (\exists\,[o..p]\,Z)(\rho_Z(Z) \wedge \mathit{tail}(Z,C) \wedge \mathit{head}(Z,B))$$
$$\varphi_Y(Y) = \rho_Y(Y) \wedge \mathit{tail}(Y,A) \wedge \mathit{head}(Y,C)$$
$$\varphi_Z(Z) = (\exists B)(\varphi_B(B) \wedge \rho_Z(Z) \wedge \mathit{tail}(Z,C) \wedge \mathit{head}(Z,B))$$

4 The expressive power of QGRAPH

In this section we characterize the expressive power of QGRAPH as the subset of FO(COUNT) in which queries are restricted to local neighborhoods, i.e., having a bounded diameter, and numbers are not quantified, i.e., the only quantifiers are of the form "$\exists[i..j]x$" for constants i, j. Recall that the language FO(COUNT) consists of first-order logic, plus counting quantifiers: $(\exists i\, x)\varphi(x)$ means that there exist at least i distinct elements x such that $\varphi(x)$. Here i is a number constant or variable. In FO(COUNT) it is also permissible to count tuples of elements, e.g., $(\exists i\,\langle x,y\rangle)\varphi(x,y)$ means that there exist at least i distinct pairs $\langle x,y\rangle$ such that $\varphi(x,y)$. In the presence of ordering, FO(COUNT) captures the complexity class ThC^0 consisting of bounded depth, polynomial-size threshold circuits [Imm].

Proposition 4.1 *The expressive power of* QGRAPH *is the subset of FO(COUNT) in which queries are restricted to local neighborhoods and there are no number variables.*

Proof: Recall that we have defined the meaning of a QGRAPH query to be a set of tuples $\langle A_1,\ldots,A_a,X_1,\ldots,X_a,$ $S_{B_1},\ldots,S_{B_b},S_{Y_1},\ldots,S_{Y_y}\rangle$ of unannotated objects and links together with sets of annotated objects and links. We will show that the formula expressing the tuple of unannotated objects and the formulas expressing membership in the sets of annotated objects are all expressible in the specified subset of FO(COUNT). Conversely, we will show that every appropriate formula in FO(COUNT) is the meaning φ_Q of a query in QGRAPH.

In Section 3 we have written the formal semantics of QGRAPH in this subset of FO(COUNT). Note that the locality property comes from the requirement of QGRAPH that all queries consist of a connected graph, even when those vertices and edges whose annotation includes 0 in the range are deleted.

We next show that any such formula of FO(COUNT) is expressible in QGRAPH. First suppose that φ is a quantifier-free, local query. We can thus write φ in disjunctive normal form, where each clause represents a connected relationship between its vertices and edges. Thus, each such clause can be represented by a QGRAPH query. Thus φ is represented by the union of these queries.

Suppose inductively that the local FO(COUNT) formula ψ is represented by a QGRAPH query, Q. Let $\varphi = (\exists[i..j]A)\varphi$. Then φ is represented by the following modification of Q: draw a new subquery box around A, and annotate it with "$[i..j]$". The new subquery box should include all annotated vertices that are adjacent to A as well as all subqueries that contain a vertex adjacent to A. Thus it follows by induction that every appropriate FO(COUNT) query is expressible in QGRAPH. $\square$

It is interesting to note that a QGRAPH query that has no subqueries has depth of nesting of quantifiers at most two, corresponding to the existence of an annotated vertex, B, and within that an annotated edge, Y. The only way to write queries with a greater depth of quantification is by nesting subqueries. For this reason, a relatively simple looking query tends to be relatively simple to evaluate. We next discuss the evaluation of QGRAPH queries.

5 Evaluation of QGRAPH queries

It is relatively straight-forward to evaluate a QGRAPH query, Q, using the semantics that we described in Section 3. We start by choosing an unannotated vertex A. We find all the objects in the database that match A. We next choose some unannotated edge, X, adjacent to A, and the other adjacent vertex A'. For each object in our current match for A, we follow its adjacency list to find all links satisfying ρ_X that are also adjacent to an object satisfying $\rho_{A'}$. We continue in this way until all tuples of unannotated vertices and edges have their potential matches.

Next, for each edge, Y, with annotation $[i..j]$, adjacent to an unannotated vertex, A, and another vertex C, we collect the set of links satisfying ρ_Y whose other endpoints satisfy ρ_C. These should be maintained in a B-tree, sorted by the pair of endpoints, with the current count of such links for each pair of endpoints. If any of these counts are less than the lower limit i, then the associated links are removed. After all remaining links and constraints for the endpoints have been evaluated, those links with counts greater than j are also removed.

Edges from A into a subquery, Q_k, are evaluated in a similar way, with a B-tree maintaining the counts of the tuples of unannotated objects in Q_k, for each tuple of annotated vertices including A that have edges into Q_k.

In future work, we plan to maintain statistical information concerning our database. We will use this information to automatically estimate the size of QGRAPH queries, and report these sizes to the user. We will make use of these estimates to decide in which order to evaluate the queries.

Acknowledgments

We thank Jen Neville for her timely assistance with the examples presented in this paper, and Matt Cornell for his skillful implementation of the QGRAPH query processor. We benefited from their insightful comments in the early design phase of QGRAPH. Our examples are drawn from queries on the Internet Movie Database (`www.imdb.com`) and the Movies database from the UCI KDD Archive (`kdd.ics.uci.edu`). This research is supported by DARPA/AFOSR under contract No. F30602-00-2-0597 and DARPA/AFRL under contract No. F30602-99-C-0061. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained in this paper are solely those of the authors.

References

[ABS] S. Abiteboul, P. Buneman, and D. Suciu (2000). *Data on the Web*. Morgan Kaufmann.

[AHV] S. Abiteboul, R. Hull, and V. Vianu (1995). *Foundations of Databases*. Addison-Wesley.

[AQM] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener (1997). The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68-88.

[CM] M. Consens and A. Mendelzon (1990). GraphLog: a Visual Formalism for Real Life Recursion. PODS 1990, 404–416.

[Imm] N. Immerman (1999). *Descriptive Complexity*. Springer Graduate Texts in Computer Science.

[JN] D. Jensen and J. Neville (2002). Schemas and Models. To appear in *Proceedings of the KDD 2002 Workshop on Multi-Relational Data Mining*.

[NJ] J. Neville and D. Jensen (2002). Supporting Relational Knowledge Discovery: Lessons in Architechture and Algorithm Design. *Proceedings of First International Workshop on Data Mining Lessons Learned*, ICML 2002.