Computer Science Department Faculty Publication Series

Computer Science

2001

A Visual Query Language for Relational Knowledge Discovery

H. Blau
University of Massachusetts - Amherst

Follow this and additional works at: http://scholarworks.umass.edu/cs_faculty_pubs

Part of the Computer Sciences Commons

A Visual Query Language for Relational Knowledge Discovery

H. Blau*   N. Immerman*   D. Jensen*

{`blau, immerman, jensen`}`@cs.umass.edu`
Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610

Abstract

QGRAPH is a visual query language for knowledge discovery in relational data. Using QGRAPH, a user can query and update relational data in ways that support data exploration, data transformation, and sampling. When combined with modeling algorithms, such as those developed in inductive logic programming and relational learning, the language assists analysis of relational data, such as data drawn from the Web, chemical structure-activity relationships, and social networks. Several features distinguish QGRAPH from other query languages such as SQL and Datalog. It is a visual language, so its queries are annotated graphs that reflect potential structures within a database. QGRAPH treats objects, links, and attributes as first-class entities, so its queries can dynamically alter a data schema by adding and deleting those entities. Finally, the language provides grouping and counting constructs that facilitate calculation of attributes that can capture features of local graph structure. We describe the language in detail, discuss key aspects of the underlying data model and implementation, and discuss several uses of QGRAPH for knowledge discovery.

1 Introduction

We have been investigating how to analyze large sets of relational data. As an example of such data, consider Figure 1, which shows a fragment from a database about movies. The figure uses *objects* to represent movies (e.g., *Network* and *The Thomas Crown Affair*), people (Faye Dunaway), organizations (MGM), and things (Oscars), and it uses binary *links* to represent relations between objects (e.g., ActorIn, DirectorOf). 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 gender of an actor, 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. The figure represents only a fragment of much larger database of movies, persons, organizations, and awards.

The data shown in Figure 1 exemplify a general data representation we have been exploring for knowledge discovery. Based on recent usage of the term in knowledge discovery and machine learning (e.g., De Raedt and Kramer 2000), we refer to this data representation as "relational", in contrast to data where objects are homogeneous, identically distributed, and statistically independent (often called "propositional data" or "iid data"). Specifically, our relational data sets consist of objects, binary links, and attribute-value pairs that record features of the object or link. An object or link can have zero or more attribute-value pairs, and all attributes are set-valued. That is, multiple values for the same attribute can be stored on an object. For example, a person can have multiple names. We are investigating how large data sets in this representation can be analyzed in an integrated knowledge discovery system called PROXIMITY (*http://kdl.cs.umass.edu/systems/proximity/*).

Relational data are an increasingly common target for research in knowledge discovery. Work in inductive logic programming (e.g., Muggleton 1992) and social network analysis (Wasserman and Faust 1994)
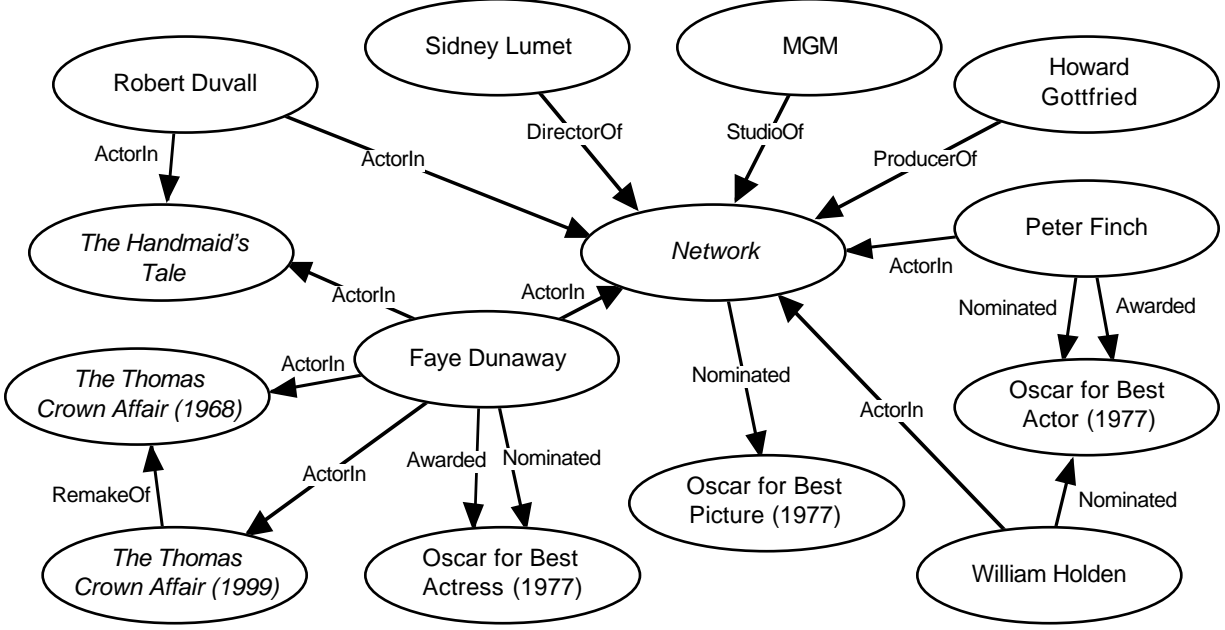

Figure 1: Graphical data fragment from a movie database

have explored this topic for years. More recently, work in learning statistical models of relational data has yielded several practical techniques (De Raedt and Kramer 2000; Getoor and Jensen 2000). Finally, work on analyzing data about the Web and other computer networks has produced useful algorithms for Web page classification (Kleinberg 1999; Craven et al. 1998).

Our own work with several large data sets has indicated the need for a query language with special features that support relational knowledge discovery. We have designed a new query language — QGRAPH— that has many of these features. QGRAPH queries can identify *subgraphs* of a larger graph, and allow variation in the number and types of objects and links that form the subgraph. Queries can identify and name the particular roles that specific objects and links serve in a matched subgraph. Finally, queries can transform matched subgraphs by adding and removing objects, links, and attributes. Added attributes can be the result of evaluating mathematical expressions that include the values of attributes on the objects and links in a matched subgraph.

The feature set of QGRAPH distinguishes it from other query languages such as SQL, Datalog, and Lorel, and make it well-suited for several phases of the knowledge discovery process, including ad hoc exploration, data transformation, sampling, and mining. We have found QGRAPH to be both an expressive and intuitive medium for expressing the queries necessary for these operations.

The first section below describes the language and gives examples of using QGRAPH to query the database from which Figure 1 was drawn. Next, we examine why the language is useful for knowledge discovery. We explain the multiple roles played by QGRAPH for PROXIMITY, and compare the expressiveness of this language to that of other languages such as SQL and Datalog. Finally, we describe our implementation to date and our plans for future work.

2

2 Language description

QGRAPH is a visual language for querying and updating a database of relational structures. 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 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.

2.1 Conditions

The query shown in Figure 2 finds all subgraphs where an `ActorIn` link exists 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. Labeling query elements is useful for talking about the query and for writing constraints (see Section 2.5). 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 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
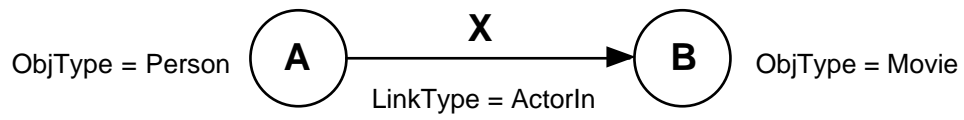
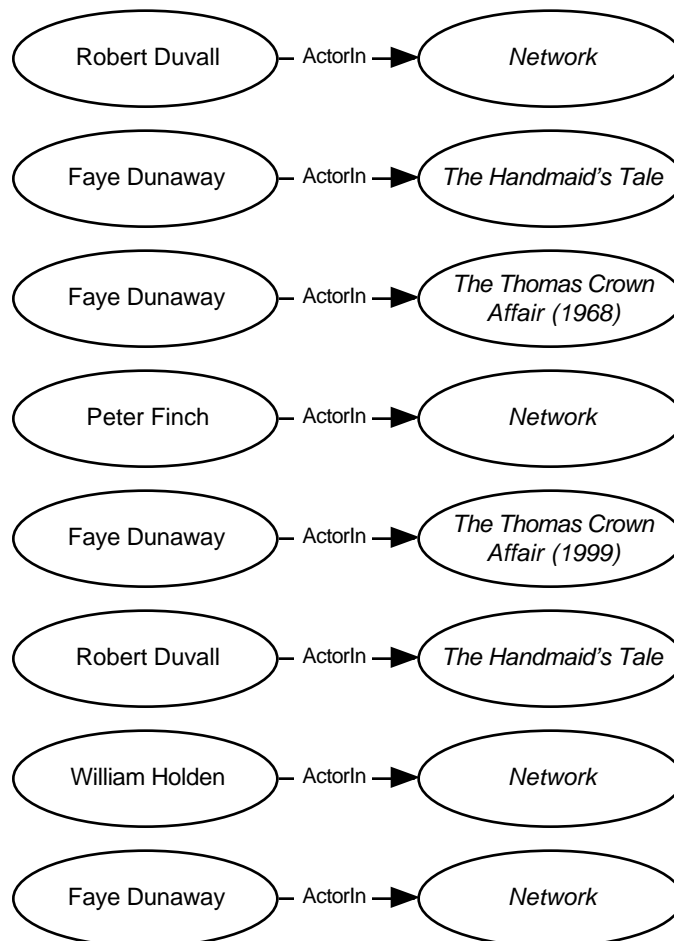Figure 2: Find all `Person`, `ActorIn`, `Movie` subgraphs

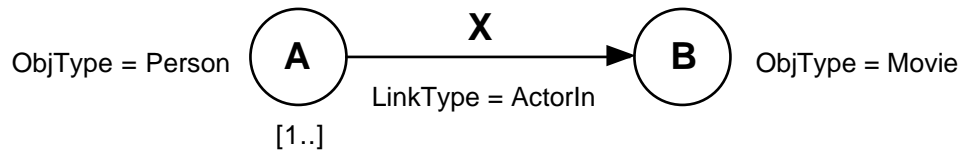Figure 3: Matches for query in Figure 2

Figure 4: For each movie, find all its actors
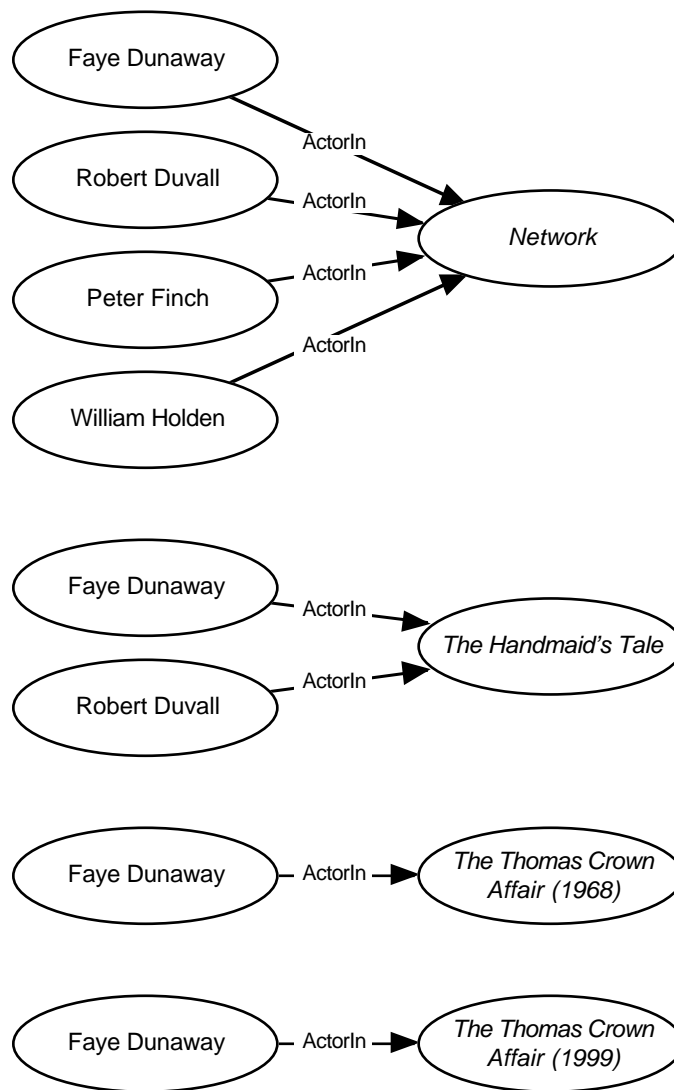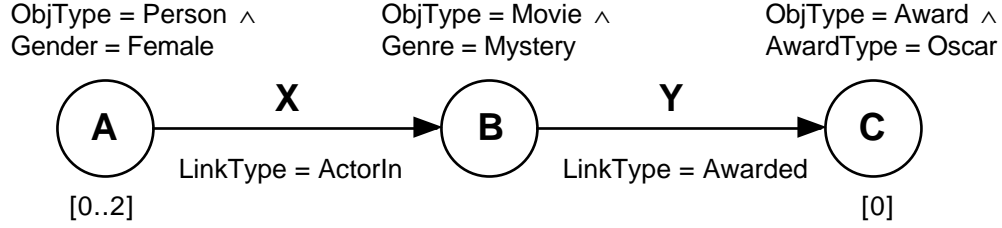

Figure 5: Matches for query in Figure 4

5

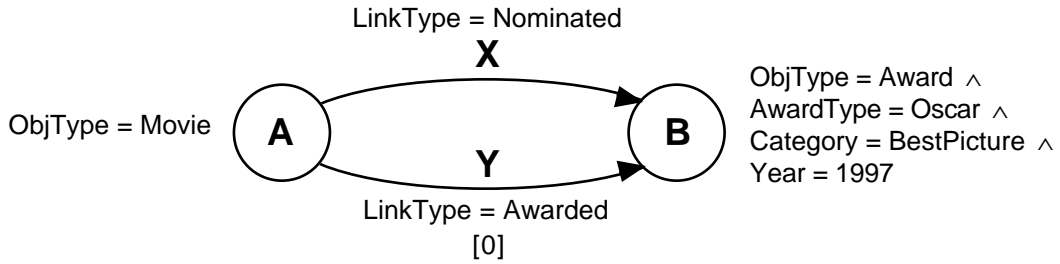Figure 6: Mysteries with fewer than 3 female actors and no Oscar awards

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

$[1..j]$.) 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).

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. 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.

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 garnered several nominations. *Sleuth* 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**, leaving just `ObjType = Award`.

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.

To be well-formed, a query must remain a connected graph when any optional or negated structures
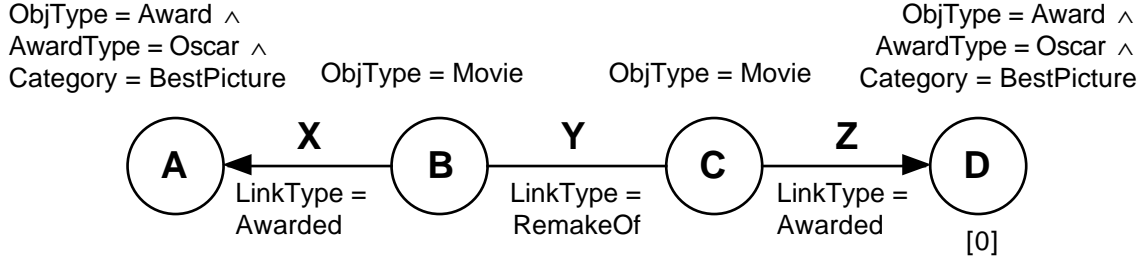
Figure 8: {Remake, original} pairs where one won Best Picture and the other did not

(annotations [0], [0..], or [0..n]) are removed. To avoid ambiguities of interpretation, only one of any two adjacent vertices can be annotated. An edge incident to an annotated vertex can itself be annotated. If the edge incident to an annotated vertex has no explicit annotation, it bears an implicit annotation of [1..]. The annotation on the vertex takes precedence over the annotation (implicit or explicit) on the edge. We first find objects that match the annotated vertex, then for each matching object we find links that match the annotated edge.

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.

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. Either the remake or the original received the award, but not both. 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.
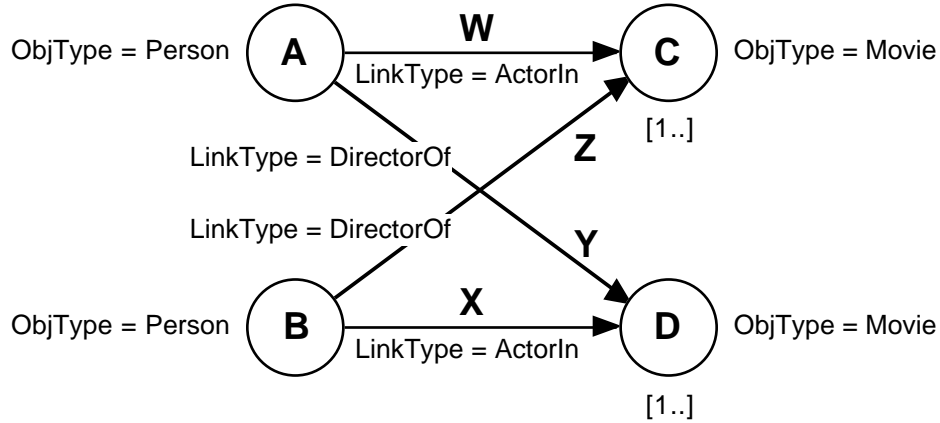
7

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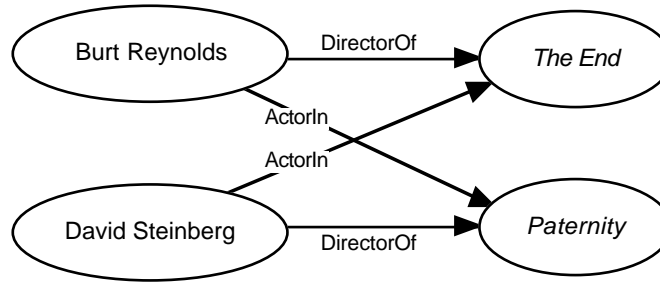

Figure 10: Database fragment for Burt Reynolds and David Steinberg

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 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: A $\neq$ B and C $\neq$ D. (Inequality constraints on the vertices force the edges to be distinct as well, since one edge cannot have two different endpoints.) 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.
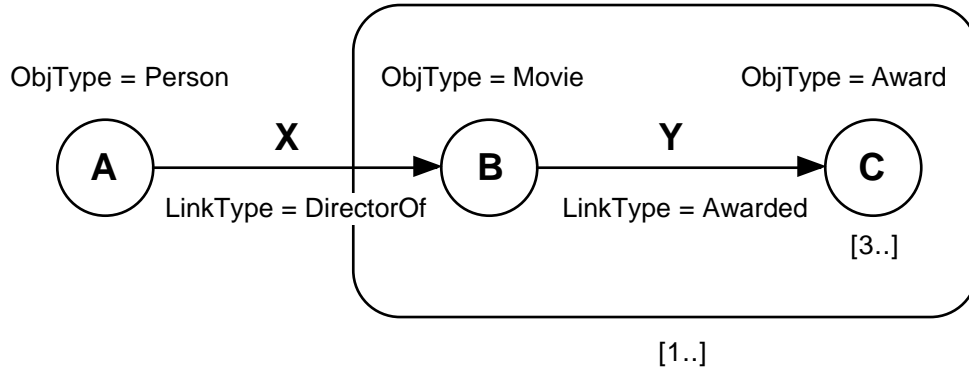
8

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

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.

2.6   Subqueries

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 another 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.

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 many convenient features for data extraction, QGRAPH is a flexible data transformation language for graph databases. We can add new objects, links, and attributes, or delete existing ones. We can calculate new attribute values by applying simple arithmetic operations or aggregation functions (sum, average, *etc.*) to the values of known attributes.

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. Within a query having more than one update element, the result of applying one update cannot create a new match for another update from the same query.
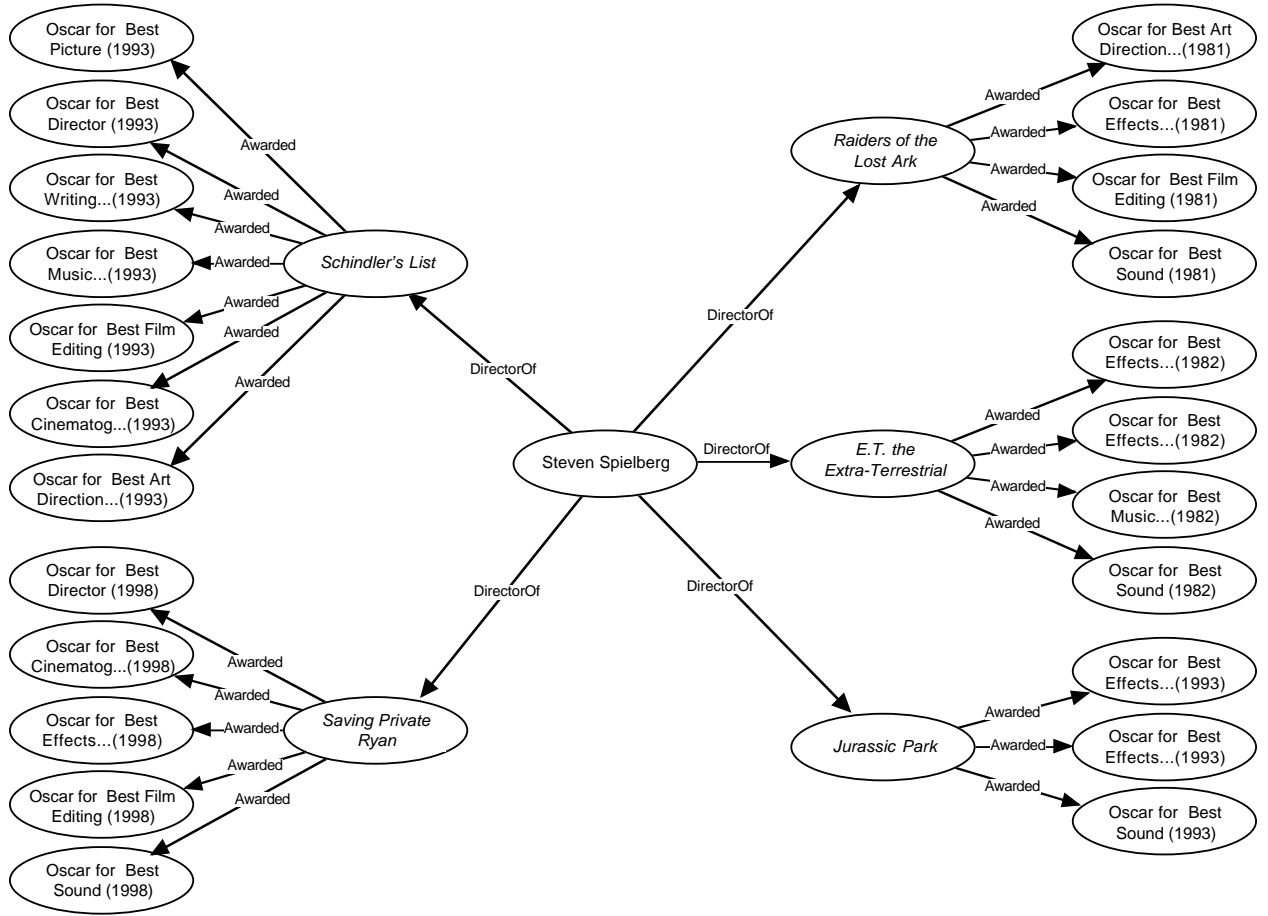
Figure 12: Match for query in Figure 11

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 ObjTypes.

The notation attribute = value is shorthand for value ∈ values(attribute). Likewise, attribute ≠ value is shorthand for value ∉ values(attribute). Note that any object or link for which attribute is undefined (that is, values(attribute) = ∅) satisfies the condition attribute ≠ value. If these matches are undesirable, we can eliminate them with a compound condition that first tests if the attribute is defined: (attribute ≠ NULL ∧ attribute ≠ 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
  values(attribute) ← {newValue}
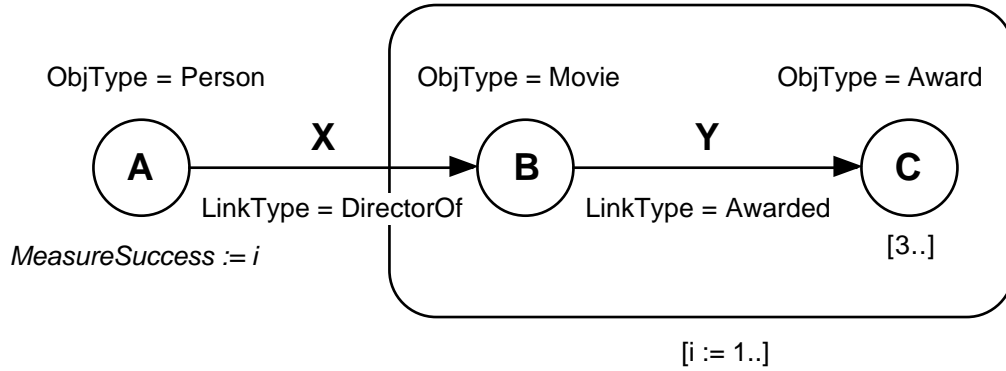
10

Figure 13: Add measure of success as attribute of director

- add the new value to the existing ones, written
  `attribute += newValue`
  which means
  `values(attribute)` ← `values(attribute)` ∪ `{newValue}`

- remove an existing value from the set, written
  `attribute -= oldValue`
  which means
  `values(attribute)` ← `values(attribute)` − `{oldValue}`

We can add, remove, or replace multiple values at once. For example,
`attribute := newValue1, newValue2, newValue3`
means
`values(attribute)` ← `{newValue1, newValue2, newValue3}`.
To remove all values for an attribute, set it to null:
`attribute := NULL`
which means
`values(attribute)` ← ∅.

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.

11

ObjType = Person   ObjType = Movie   ObjType = Studio

X   **Y**

A   **B**   **C**

LinkType = ActorIn   LinkType = StudioOf

[1..]

LinkType := EmployeeOf
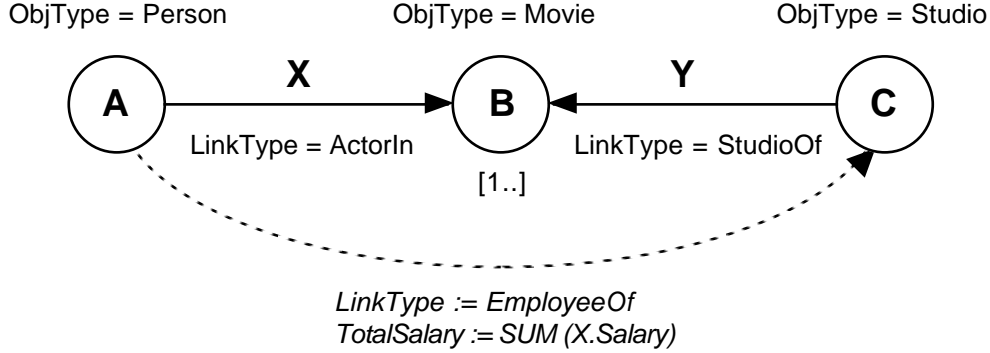*TotalSalary := SUM (X.Salary)*

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

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: $SUM(X.Salary) \geq 10^6$.

3 Query Languages and Knowledge Discovery

Querying and updating are key operations for several stages of knowledge discovery. Querying is necessary for effective data exploration, data sampling, and data mining. Updating is central to data transformation, in addition to being necessary for operations such as sampling and data mining.

Despite these uses, query languages for knowledge discovery have not been widely investigated. One reason is that many knowledge discovery methods address propositional data, and querying and updating propositional data is relatively simple. Nearly all queries can be expressed in simple SQL, and many data transformation are functions applied to single attributes (e.g., $\log x$) or simple mathematical expressions combining the values of several attributes from a single instance (e.g., $patient\text{-}height/patient\text{-}weight$).

In contrast, our work with relational data has highlighted the importance of querying and updating complex graphical structures such as those discussed in the previous section.

Specifically, we have developed QGRAPH to assist with:

- Data exploration — We frequently need to ask questions about the existence and frequency of specific structures present in the data. For example:

  Q: Does the database contain movies linked to zero actors?
  A: Yes, but most cases appear to be errors.
  Q: Do actors ever have multiple roles in the same movie?
  A: Yes, Tony Randall had seven roles in *The Seven Faces of Dr. Lao*.

  Such ad hoc exploration of large data sets virtually mandates a query language, and our needs in this area were a strong motivation in developing QGRAPH.

- Attribute creation — We often need to calculate attributes that capture properties of local graph structure. Such attributes can be calculated based on the elements of a subgraph, and then stored as an attribute of an object or link. For example: How many award-winning movies has each actor starred in? How many studios has each actor worked for? What is the first year that each actor starred in any movie? In contrast to standard attribute calculations for propositional data, calculating attributes in relational data requires querying both the structure and attributes of the data. Such attributes are important to many relational learning and inference methods, including some ILP techniques (Muggleton 1992), probabilistic relational models (Getoor et al. 2001), and iterative classification (Neville and Jensen 2000).

- Structural transformations — We have often found it convenient to make fundamental alterations in the structure of databases by adding and deleting objects and links. For example, some calculations about the relationships among movies can be made substantially more efficient by adding links that directly connect movies that share a common actor or director. Similarly, objects and links might be added to represent "families" of movies (e.g., the multiple Star Wars or James Bond movies) that share a common studio, producer, and set of characters. Such structural transformations allow fundamental aspects of a particular data representation to be altered dynamically in ways that assist analysis, and we have found such alterations to be important for practical knowledge discovery.

- Sampling — Our recent work on sampling relational data (Jensen and Neville 2001) shows the importance of sampling entire subgraphs rather than individual objects or links. For example, one type of subgraph in the movie data might include movies and all linked actors, directors, and producers. Sampling individual objects can produce biased parameter estimates in statistical models, and lead to incorrect estimates of accuracy. Sampling entire subgraphs can overcome such biases, but constructing such subgraphs requires an appropriate query language.

- Data mining — Our work on data mining in relational data (Neville and Jensen 2000) uses Bayesian classifiers, but queries and updates can also be thought of as a knowledge representation. Methods have been developed to learn Datalog queries that accurately classify relational data, and we conjecture that methods could also be developed to learn QGRAPH queries. Though we have not yet developed such methods, we plan to investigate them in the future.

4   Related Work

Researchers and practitioners of knowledge discovery already have access to many query languages, including SQL, Datalog, Lorel, OQL, and GOOD. Our own work has made extensive use of SQL, and other researchers in knowledge discovery and machine learning make use of Datalog. Lorel (Abiteboul, Quass, McHugh, Widom, and Wiener 1997), OQL (Alashqur, Su, and Lam 1989), and GOOD (Gyssens, Paredaens, Van den Bussche, and Van Gucht 1994) are three of the better known languages for querying semi-structured data. QGRAPH differs from these other query languages because of its design goals and the degree to which the language achieves those goals. Specifically, we designed QGRAPH to be visual, intuitive, and useful for knowledge discovery.

QGRAPH is a truly *visual* language. In contrast to SQL and Datalog, the primary elements of QGRAPH queries and updates are expressed as graphs with textual annotations. While graphic user interfaces and visualizers have been developed for SQL and Datalog, these tools are only adjuncts to the primary textual representation. QGRAPH queries have a structure that closely approximates the data, making it easier for users to imagine the structure of potentially matching subgraphs.

Many of our design decisions were intended to preserve syntactic clarity in ways that make the language more *intuitive*. Specifically, we have avoided language features that have several reasonable and conflicting interpretations, rather than make arbitrary choices about language semantics. For example, an annotation of the form $[..3]$ could be interpreted either as $[0..3]$ or $[1..3]$. We designed QGRAPH to require users to specify a lower bound, rather than assigning an arbitrary meaning when a lower bound was not provided.

Similarly, we have made several design decisions because of overall conceptual clarity and ease of interpretation. For example, queries containing updates are processed by first finding all matches to the match portion of the query and then executing all parts of the update portion in parallel. This requires that some types of updates (e.g., those with interdependent calculations) be executed as multiple queries, but it greatly simplifies the language semantics.

The decision to represent attributes textually illustrates a tradeoff between our goals of a visual language and an intuitive one. Some visual query languages, such as GOOD, represent attributes as nodes in the query graph. For queries involving even a small number of attributes, the visual elements of the query that represent attributes dominate, and obscure the structural elements of objects and links. In contrast, it is easy to distinguish between structural elements and attributes in QGRAPH queries.

A third design goal was to make QGRAPH *useful* for knowledge discovery. We wanted a language for easily expressing common queries needed for our work with PROXIMITY. These include queries for ad hoc exploration, data sampling, and attribute calculation. Essentially all of these queries involve small subgraphs surrounding core objects (e.g., all people related to a given studio through movies). Because of this goal, we focused on queries that capture the characteristics of local graph structure, rather than global characteristics of graphs (e.g., diameter, mean shortest path, etc.). We decided against language features that would allow queries to express transitive closure or calculate distances between arbitrary nodes. Given our current experience, we feel that these features would be complex to implement and would be rarely used.

For similar reasons, we designed the language to query and update graphs containing links with multiple attributes. Many existing languages either assume that links contain no data or that they have a fixed attribute structure (e.g., a single type attribute). For example, languages for querying XML data often have these restrictions. Our data model allows links to have an arbitrary number of set-valued attributes, and QGRAPH provides the means to query and update such attributes.

5 Implementation and Future Work

We are currently implementing QGRAPH within PROXIMITY, our system for relational knowledge discovery. Our current implementation of PROXIMITY stores all data in relational database tables. Separate tables are used for objects, links, and individual attributes. Collections of subgraphs are stored in an additional table. When PROXIMITY processes a QGRAPH query, it is compiled to a set of SQL queries that insert rows into the subgraph table based on the query and the contents of the object, link, and attribute tables.

Our current implementation processes only simple queries that contain a single core vertex connected to one or more edges and vertices. Each edge and non-core vertex can be annotated or unannotated. Vertices and edges can be constrained via boolean expressions on attributes. Several advanced language features remain to be implemented, including subqueries and global constraints. Despite the limitations of our current implementation, many useful queries can be expressed within this subset of QGRAPH. For example, many of the queries we make for sampling and attribute calculation can be made with this subset of the language. Our work on the implementation continues, and we expect to complete all features of the current version of the language in the next three months.

In addition, we are currently characterizing the complexity of the QGRAPH queries. This work aims to improve the efficiency of query processing and provide feedback to users as they construct and edit queries. We hope to develop a small number of statistics about databases that can be easily maintained and used to improve the efficiency of query processing. Finally, we hope to examine the effect of particular graph topologies on the efficiency of QGRAPH queries. Recently, attention has focused on so-called "small world" topologies (Watts 1999), which exhibit both small diameter and a high degree of clustering. We conjecture that such topologies will create substantially different demands on query processing than more uniform topologies.

Acknowledgments

References

Abiteboul, S., 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.

Alashqur, A. M., Su, S. Y. W., and Lam, H. (1989). OQL: A query language for manipulating object-oriented databases. *Proceedings of the VLDB Conference* (pp. 433-442).

Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T., Nigam, K., and Slattery, S. (1998). Learning to extract symbolic knowledge from the World Wide Web. *Proceedings of the Fifteenth National Conference on Artificial Intelligence* (pp. 509-516). Menlo Park: AAAI Press.

De Raedt, L., and S. Kramer (Eds.) (2000). *Proceedings of the Workshop on Attribute Value Learning and Relational Learning: Crossing the Boundaries*, Workshop held at the 17th International Conference on Machine Learning, Stanford.

Getoor, L. and D. Jensen (2000). *Learning Statistical Models from Relational Data: Papers from the AAAI 2000 Workshop*. Menlo Park: AAAI Press. Technical Report WS-00-006.

Getoor, L., N. Friedman, D. Koller, and A. Pfeffer (2001). Learning probabilistic relational models. In *Relational Data Mining* (S. Dzeroski and N. Lavrac, Eds.). Springer-Verlag.

Gyssens, M., J. Paredaens, J. Van den Bussche, and D. Van Gucht (1994). A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering* 6(4): 572-586.

Jensen, D. and J. Neville (2001). Subgraph Sampling for Relational Data. In *Computing Science and Statistics: Proceedings of the Thirty-Third Symposium on the Interface*. Volume 33 (forthcoming).

Kleinberg, J (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM 46*.

Muggleton, S. (Ed.) (1992). *Inductive Logic Programming*. San Diego: Academic Press.

Neville, J., and D. Jensen (2000). Iterative classification in relational data. In *Learning Statistical Models from Relational Data: Papers from the AAAI 2000 Workshop* (L. Getoor and D. Jensen, Eds.). Menlo Park: AAAI Press. Technical Report WS-00-006.

Wasserman, S. and K. Faust (1994). *Social Network Analysis: Methods and Applications*. Cambridge: Cambridge University Press.

Watts, D. (1999). *Small Worlds*. Princeton, NJ: Princeton University Press.