Expressing Structural Hypertext Queries in GraphLog

Mariano P. Consens
Alberto O. Mendelzon

Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A4

ABSTRACT

GraphLog is a visual query language in which queries are formulated by drawing graph patterns. The hyperdocument graph is searched for all occurrences of these patterns. The language is powerful enough to allow the specification and manipulation of arbitrary subsets of the network and supports the computation of aggregate functions on subgraphs of the hyperdocument. It can support dynamically defined structures as well as inference capabilities, going beyond current static and passive hypertext systems.

The expressive power of the language is a fundamental issue: too little power limits the applications of the language, while too much makes efficient implementation difficult and probably affects ease of use. The complexity and expressive power of GraphLog can be characterized precisely by using notions from deductive database theory and descriptive complexity. In this paper, from a practical point of view, we present examples of GraphLog queries applied to several different hypertext systems, providing evidence for the expressive power of the language, as well as for the convenience and naturalness of its graphical representation. We also describe an ongoing implementation of the language.

INTRODUCTION

Hypertext systems are intended to support the organization and manipulation of networks of text nodes (or multimedia nodes, for hypermedia) connected by typed links. As current systems start getting more use, several limitations in the basic approach are becoming apparent. In a recent survey [Hala88], seven key issues are identified as requiring work for the next generation of hypertext systems. The first five of these are:

This work has been supported by the Information Technology Research Centre of Ontario and the Natural Science and Engineering Research Council of Canada. The first author was also supported by the PEDECIBA – United Nations Program for the Development of Basic Sciences, Uruguay.

1. **Search and query facilities.** Current systems are heavily oriented towards browsing and network navigation. They lack powerful query languages that allow the specification and manipulation of arbitrary subsets of the network.

2. **Augmenting the basic node and link model.** The directed graph model is too low level to support complex ways of organizing the information in a network.

3. **Virtual Structures.** Hypertext systems support only manual changes to the contents or structure of a network, making them relatively static in practice. It would be desirable to have dynamically defined structures that can make the network reconfigure itself automatically in response to changes.

4. **Computation over graphs.** Current systems are passive; for example, they do not include inference engines that may actively derive new information from what is explicitly stored.

5. **Versioning.** When hypertext technology is applied to the maintenance of large technical documents, or to computer assisted engineering, it is essential to have mechanisms for managing versions and configurations and to control concurrent access reliably.

In this paper we describe a powerful query language for hypertext, called GraphLog, that addresses all points 1 to 5 above. Point 3 is addressed by allowing the definition of virtual links, point 4, by supporting computation of aggregate functions on subgraphs of the document graph, and point 5, versioning, by using GraphLog to specify versioning policies. Finally, GraphLog can be easily extended to more elaborate object-oriented models that address point 2.

GraphLog queries are visually oriented; they are formulated by drawing with a graph editor the patterns that are to be searched for in the hypertext network. Halasz distinguishes in [Hala88] between *structural* and *content based* search. Content based searches will find nodes that contain certain patterns; structural searches look for whole subgraphs of the overall graph that have a certain structure. GraphLog emphasizes structural queries, although in the conclusions we suggest the integration of structural and content-based queries in a single language.

An important issue in the design of such a language is expressive power: too little power limits the applications of the language, while too much makes efficient implementation difficult and probably affects ease of use. We have two sorts of arguments for the adequacy of the expressive power of GraphLog. Elsewhere, we have used notions from deductive database theory and descriptive complexity to characterize from a theoretical point of view the class of queries that can be formulated in the language[1][Cons89]. In this paper, from a practical point of view, we survey several existing hypertext systems and queries described by their authors and show how they can all be expressed in GraphLog.

[1] In fact, the name of the language comes from its close relationship to Datalog, in turn a relative of Prolog.

THE QUERY LANGUAGE

The graph-based query language G^+ provided a starting point for GraphLog. We have extended G^+ by adding negation and changing the semantics to make the definition of the language simpler. In GraphLog, an interrelated collection of documents – a hyperdocument – is viewed as a directed labelled graph. A *query* is a graph pattern containing one distinguished edge. The effect of the query is to find all instances of the pattern that occur in the hyperdocument and for each one of them define the "virtual link" represented by the distinguished edge. Graph patterns are themselves graphs, and they can be specified by drawing them on a screen.

The formal semantics of the language is given in [Cons89]. Each query is given a precise meaning by associating it with a set of recursive Horn clauses defined on the relations that make up the graph. Instead of giving the full definition of the language here, we will introduce it by a series of examples and informal explanations.

Consider a hierarchical document where there are nodes for each chapter, section, subsection, etc., and edges labelled contains relate each part to its subparts. The query in Figure 1 defines a virtual link top that points from each component of the document directly to the top-level component. In this case the pattern is a pair of nodes connected by an arbitrary sequence of contains links such that the second node has no incoming contains link. For each such pattern, the query defines a top link between these nodes. Note the regular expression contains$^+$ labelling a dashed edge in the query. This means the pattern to be found is a path composed of any number of edges, each one labelled with contains. In general, any regular expression may be used; for example, if instead of contains, the graph used several different link types such as has-chapter, has-section, has-subsection, we could replace the regular expression contains$^+$ with (has-chapter | has-section | has-subsection)$^+$. The crossed-out edge in the query means that a node only qualifies as a top node if there is *no* contains edge coming into it.

This simple example already shows that GraphLog can express queries that are not expressible in conventional database languages such as relational algebra. The dashed edge, standing for a path of arbitrary length, corresponds to an arbitrarily long sequence of "join" operators, and there is no single relational algebra query equivalent to it, as is well known in database theory [Aho79].

What can we do with a virtual link such as top once we have defined it? We have three possibilities. First, we may wish to treat it simply as the answer to a query, that is, to display it to the user in some form and then forget it. Second, we may wish to incorporate it into the document, treating it as a *snapshot*. In this case, the link as computed by the query will remain in the document, but it will not be affected by future changes. Finally, we may make it into a *view*, meaning that not only does it get incorporated to the document, but it is also kept dynamically up to date, so that if, say, a new leaf node is added to the tree, its top link is automatically inserted. This choice is independent of the query language and would be made by the user interface in consultation with the user.

From the point of view of a user, once a link is created it can be manipulated in the same ways no matter whether it is a snapshot, a view, or a manually created set

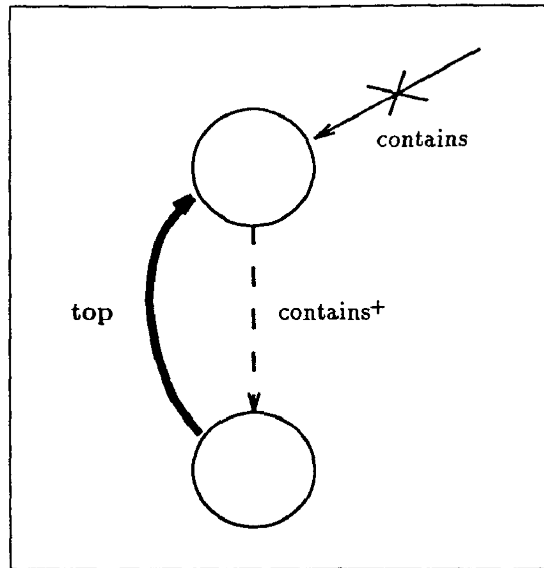

Figure 1: Defining a virtual link.

of edges. As an example of this, suppose that after creating the top link we realize that the network is not really a hierarchy, because there are nodes that belong to more than one document. We now want to link explicitly all top level nodes that share a component document. The query in Figure 2 generates a shares-with link between any two top nodes that have a common sub-document.

So far we have shown edges labelled only with a property name. This is the simplest case, and the label corresponds to the type of the link. In general, an edge may be labelled with a literal of the form $p(c_1, ..., c_n)$, associating n atomic values with the relationship between the endpoints.

Consider for example a rather different hyperdocument that might be used by a travel agency. The nodes contain textual and pictorial information about cities. One type of link between cities represents flights. We could then have edge labels of the form flight(Airline,Departure,Arrival). Another type of link gives distance information. Edge labels for this kind of links have the form dist(Distance). These values can be used in queries in many ways. Figure 3 shows a query that defines a link ind-dist(D) between Toronto and Vancouver by adding the Toronto-Calgary distance to the Calgary-Vancouver distance. Note that we are modelling node attributes such as city name by edges going to rectangular nodes. This is just to keep the data model as simple as possible; it is straigthforward to incorporate node attributes explicitly if we wish.

This last example was somewhat artificial; it would make more sense, if we do not know the distance between Toronto and Vancouver, to estimate it as the *smallest* sum of distances from Toronto to some city C and from C to Vancouver. Figure 4 shows how the *aggregate operator* min can be used in GraphLog to express this. The distinguished edge will be labelled by ind-dist(S) where S is the smallest of all sums defined above. The fact that the intermediate node is doubly circled is

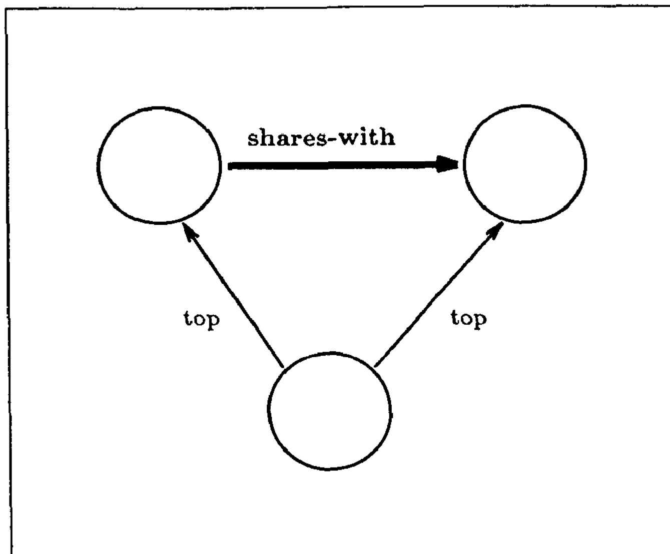

Figure 2: Documents that share a component.

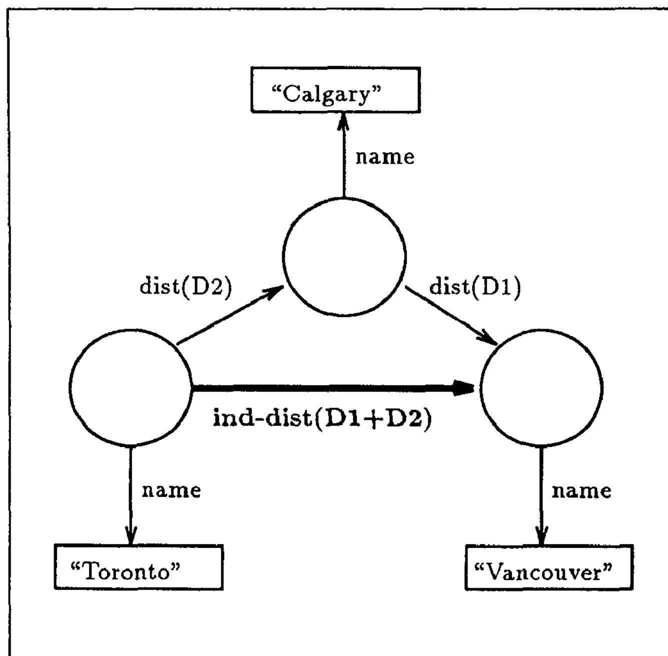

Figure 3: Distance between two cities via a third one.

meant to suggest that we are computing an aggregate over all possible choices of this intermediate node.

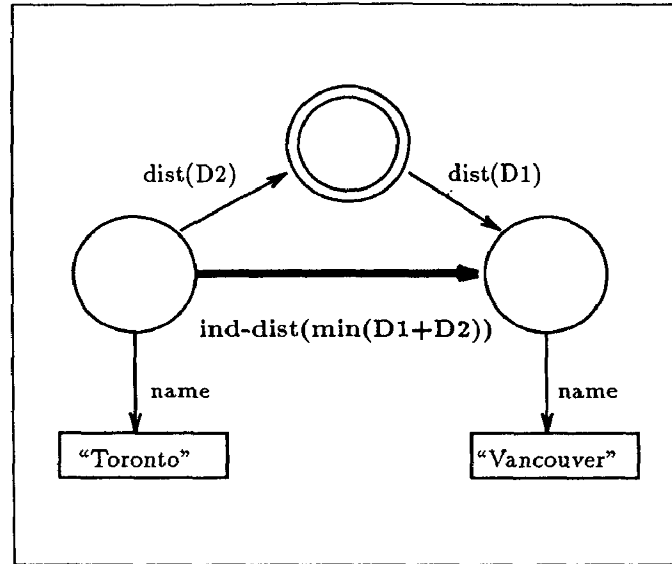

Figure 4: Distance between two cities via any third one.

Next, as the reader may expect, we are going to generalize Figure 4 to compute the *shortest distance* between Toronto and Vancouver independently of how many intermediate cities we go through. Instead of simply adding two distances, we now need to be able to add all distances appearing along a path of arbitrary length between Toronto and Vancouver. The dashed edge in Figure 5 between the two cities represents these paths. The distinguished edge will be labelled with the minimum over all paths of the sum of the distances along each path. Note the label on the path between Toronto and Vancouver is $\mathsf{dist}(\{\mathsf{D}\})^+$, not $\mathsf{dist}(\mathsf{D})^+$. This is because a label $\mathsf{dist}(\mathsf{D})^+$ would mean we are looking for paths such that, for some distance D, all hops along the path are of length D, which is not what we want. The notation {D} is meant to suggest that we want to collect the *set* of all distances because we are going to apply the *path summarization operator* sum to them.

EXPRESSIVE POWER

The expressive power of GraphLog can be characterized precisely from a theoretical point of view by relating it to the language of function-free Horn clauses called *Datalog*. Ignoring aggregate operators, GraphLog turns out to be equivalent to what is called *piecewise-linear, stratified Datalog*, a version of Datalog in which recursive rules are restricted to use the predicate being defined only once in its definition and negation is allowed in a controlled way [Ullm88]. Interestingly, the queries expressible in the language are exactly those than can be computed in space logarithmic in the size of the database. GraphLog with aggregate operators is more expressive than the relational algebra and calculus with aggregates of [Klug82]. We

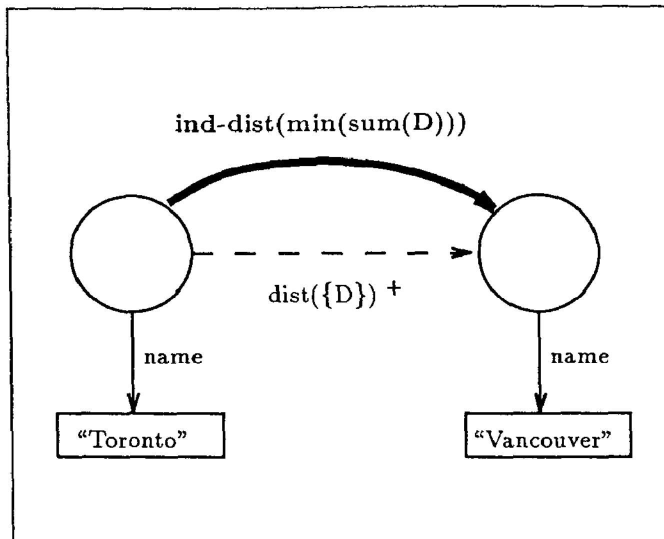

Figure 5: Shortest distance between two cities.

will not present these results here; see [Cons89] for the details.

From a practical point, we will now present examples of how **GraphLog** could be used in the context of several different hypertext systems, providing evidence for the expressive power of the language, as well as for the convenience and naturalness of its graphical representation.

NoteCards

NoteCards [Hala87] is an "idea processing" hypertext system. Nodes in NoteCards are the electronic analog of the 3×5 familiar paper notecard. A web of typed links interconnect the note cards in a hyperdocument.

An application of the NoteCards system described in [Hala87] consisted in authoring a public policy research paper. One kind of link used by the author, *supports* links, connected notecards with supporting arguments. The next two examples illustrate the possibilities of GraphLog in an idea processing hypertext system.

Example 1: Assume that in addition to linking notecards with supporting arguments, the author indicated her belief in the strength of the support by assigning it a number between 0 and 1. The strength could be represented by an edge label of the form supports(S).

The graphical query in Figure 6 defines a virtual link most-reliable(R) that connects notecard N1 to the notecard N2 containing its most reliable unsupported argument, and also gives the reliability R of this argument. A notecard N2 not supported by any other card is defined to be the most reliable unsupported argument for N1 if the weakest link in the chain of arguments from N2 to N1 is stronger than the weakest

link in any other such chain. Making this connection into a virtual link would keep it updated as the author's beliefs change and new arguments are incorporated or old ones deleted.

Note there are two boxes in this query. The first one defines a property of nodes called supported; the bottom one uses this property to find most reliable unsupported arguments. The two queries could have been combined into one box in this case; in general, it is convenient to be able to break a query down into several steps, defining at each step intermediate links that can be used in subsequent steps. The user need not be concerned with the exact order in which the different steps are executed; the system will determine an ordering that ensures each link is computed before it needs to be used, as long as there are no cycles in this ordering. Sets of queries with cyclic orderings are syntactically forbidden. □

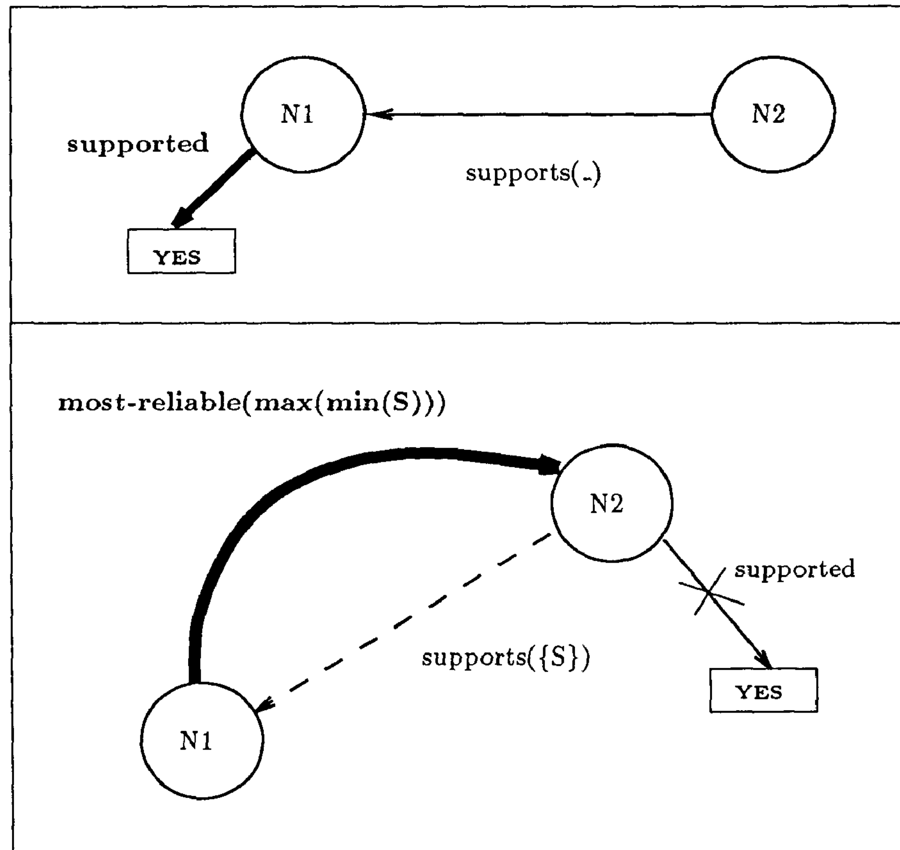

Figure 6: Creating a virtual link to the most reliable unsupported evidence.

Example 2: Suppose an author is writing a collaborative research paper and need to see what cards have been created recently by her co-authors. Figure 7 shows the query graph that defines a set of nodes with the notecards created by someone other than the author in the last three days (the constant TODAY is a "system provided" value for the current date). Note that, instead of defining any new links, we simply

highlight one of the nodes in the query to indicate that we just want the resulting set of nodes as the answer. □

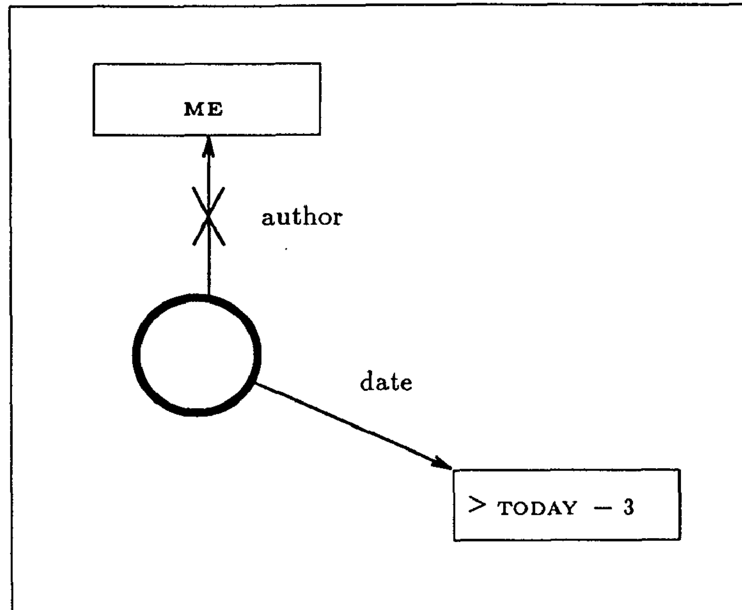

Figure 7: Nodes created by someone else in the last three days.

gIBIS

The gIBIS hypertext system has a specific objective: "to provide a systems design team with a medium in which all of their work can be computer-mediated and supported" [Bege88]. It provides a hypertext environment for the IBIS design methodology. IBIS supports the constructive discussion of the issues that arise during the design process by presenting positions that respond to the issues and arguments that support or object to the positions.

Nodes in gIBIS are of three kinds: they hold either an issue, a position or an argument. There is also a fixed number of link types. For example, a position responds-to an argument and an argument either supports or objects-to its position.

Example 3: Figure 8 shows the graphical query that finds the issues with at least two positions without arguments (another example query from [Hala88]). □

Example 4: This example illustrates the use of GraphLog in a typical "ad-hoc" query: how popular are an author's positions in a gIBIS hyperdocument? We will break this query down into three steps. The first two associate with each position P two links called count-s and count-o that point to nodes containing respectively

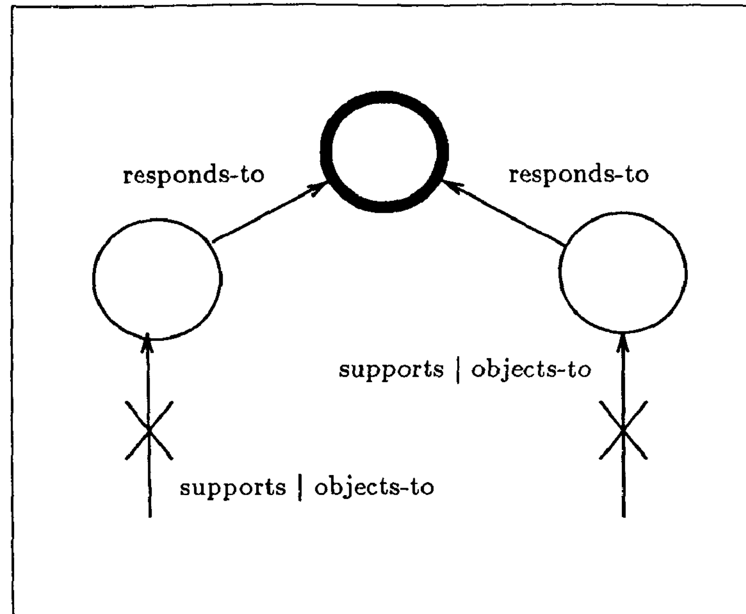

Figure 8: Issues with at least two positions without arguments.

the total number of arguments that support P and the total number of arguments that object to P. The third step computes the average of the support-to-objection ratio for the author's positions and stores this average as the "popularity" of the author. Figure 9 shows the graphical query that results. □

In gIBIS, a user can also group an issue together with its positions and corresponding arguments into what constitutes an IPA (issue-position-argument) composite node. This IPA composite node is used to record the decision reached on an issue. GraphLog would allow a flexible selection of what nodes should be grouped in a composite.

History Mechanism

A very useful aid to avoid dissorientation when browsing a hyperdocument is the history mechanism. Systems that support a history mechanism (like ZOG/KMS and HyperCard) provide the user with virtual links that connect in sequence the last nodes visited.

A slightly more sophisticated history mechanism would create attributes for the nodes in the history trace, recording for each node the time at which the user opened and closed the node. The following example illustrates a GraphLog query that uses the history mechanism to locate previously neglected relevant information.

Example 5: After a couple of hours of working within a hyperdocument, a user realizes that while avoiding distractions by not following links named digression he

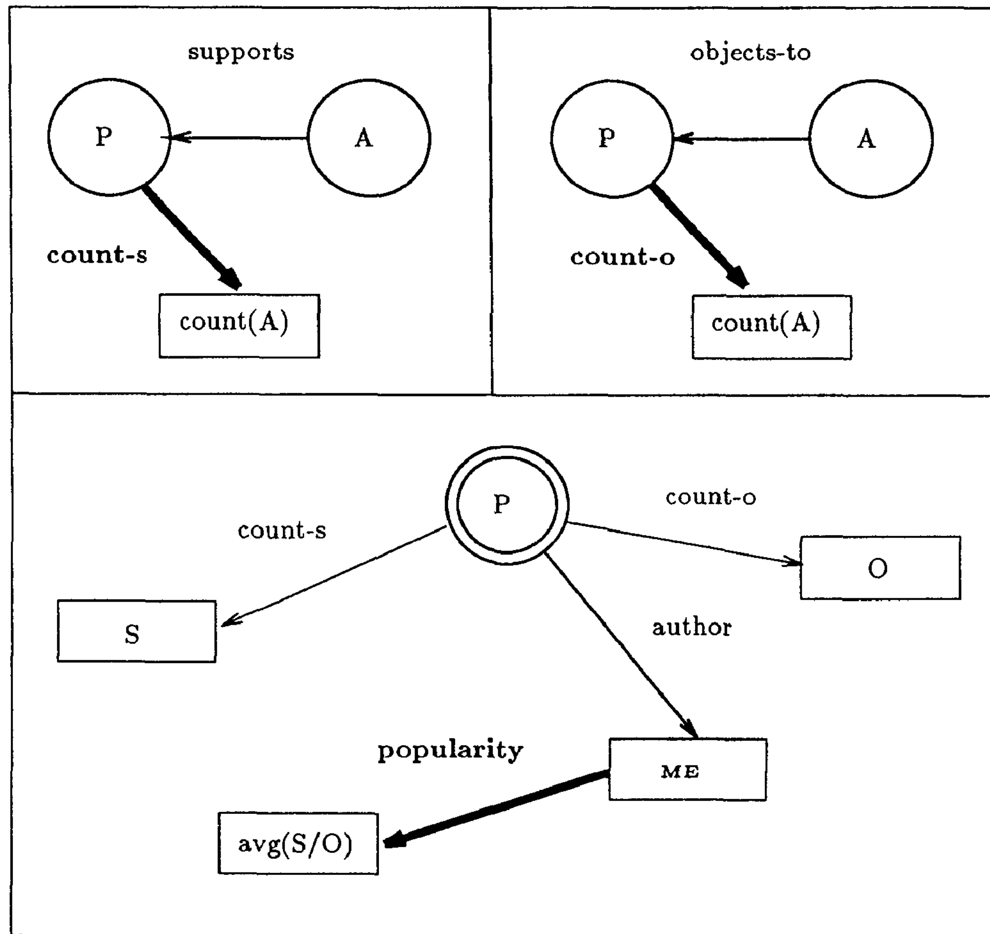

Figure 9: Finding out how popular are an author's positions.

probably missed some important point. He wonders:

> "Where did I see a link labelled "digression"? I remember that it was between 2 and 3 hours ago."

Figure 10 shows the query graph that helps the user to trim down the search. The constants HERE and NOW are "system provided" values for the currently open node and the current time, respectively. Note how the arithmetic comparisons "<" and ">" are represented by links like any other relationship between nodes, although both the links and the nodes to which they point are virtual in this case. □

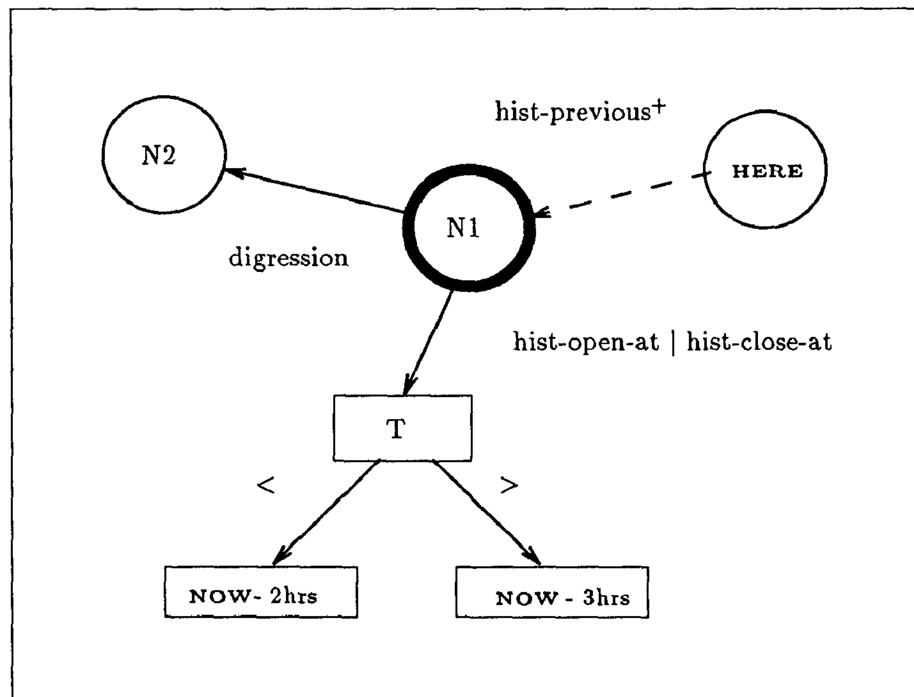

Figure 10: Searching back the history mechanism.

DynamicDesign

DynamicDesign [Bige87, Bige88] is a CASE (Computer-Aided Software Engineering) environment for the C programming language implemented as a front-end to the HAM hypertext storage system.

The nodes store all the components of a software engineering project. A node attribute, project-component, takes values that indicate the kind of component stored in the node: requirement, specification, object-code, source-code (one C function is stored per node), library, comment, dictionary (i.e., symbol table), etc.

The links are used to relate the different components. A link attribute, relation, describes the kind of relationship between nodes: calls (between functions stored in source code nodes), refers-to (from a function to a dictionary; it describes the C variable referred to in the function and has an additional attribute with the name of the C variable), in-library (from functions to libraries), implements (from functions to specifications), follows-from (describing the linear order between nodes when printed or compiled), etc.

DynamicDesign answers some queries on the structure of the hyperdocument of a software engineering project:

- What does a function do? (follows comments, implements)
- Who (directly) calls this function? (follows calls)
- What is this variable used for? (follows comments)
- Who (directly) uses this variable? (follows refers-to(V))

The importance of these queries is clear. It is also clear that there are several other relevant queries (e.g., who directly or indirectly uses this function?), and that not all of them can be anticipated and "built into" the environment.

Adding GraphLog to DynamicDesign allows dynamic specification of queries that were not anticipated by the system designers. The different values of the relation link attribute can be used to define GraphLog virtual links calls, follows-from, in-library, implements and refers-to(V), where V is a C variable name. The example below illustrates one application of GraphLog in the DynamicDesign hypertext system. It also introduces a new feature of the language, link inversion.

Example 6: Figure 11 shows the query graph that finds the functions F1 that share a variable with some function F2 implementing (directly or indirectly) the io-spec specifications, but not belonging to the syncio library nor calling any function in it. The refers-to(V) links from nodes F1 and F2 to node D means that variable V, defined in dictionary D, is referenced in both nodes. Note the "– calls" label on the path from F2 to io-spec. We are looking for a path from function F2 to some specification whose name is io-spec. The path may be a direct one, in which case we do not use the "– calls" part, or it may be that F2 is at the end of a chain of functions G_1, G_2, ..., G_n, such that G_1 implements io-spec and each G_i calls G_{i+1}, and G_n calls F2. The edges from G_i to G_{i+1} go in the opposite direction to the path; this is the purpose of the inversion operator "–". Similarly, the in-library link is inverted in the path from F2 to syncio. Notice that the node labelled with variable D is not strictly necessary; it could have been omitted by using a link from node F1 to node F2 labelled "refers-to(V) – refers-to(V)". □

The HAM versioning mechanism is particularly useful for a CASE application. The next example illustrates a query that uses a link next-version to locate a specific piece of code.

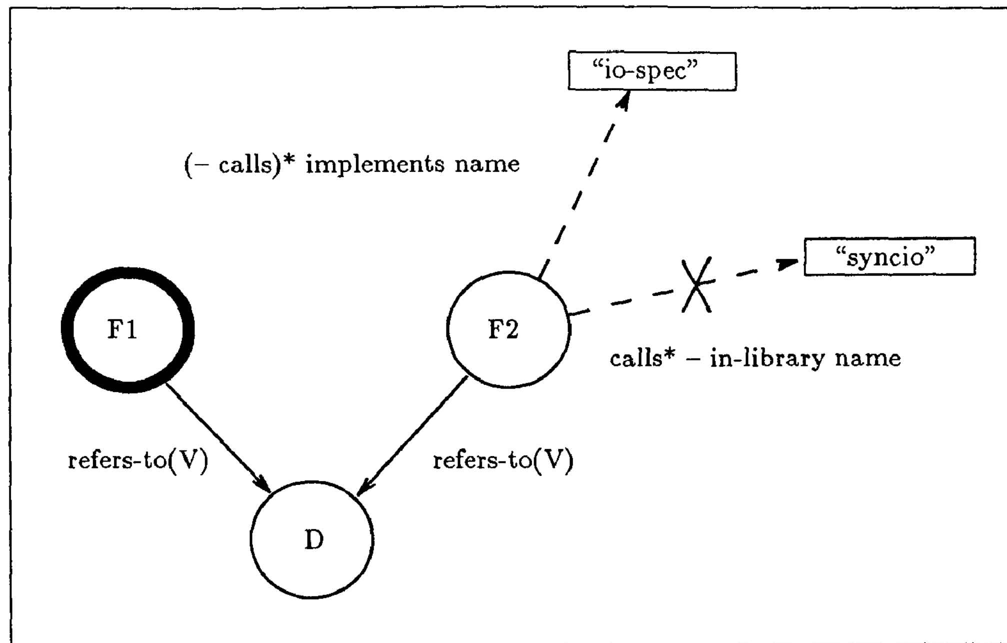

Figure 11: Finding code in hypertext CASE.

Example 7: Suppose a programmer has to find the code for a particular function, and she gives the following information:

> "I am looking for the last version of a function that implements the security-spec; the first versions were done by myself; then Dennis took charge of it; I do not remember the authors that followed him; maybe Dorothy or Chris were in charge before Jeff, who I am sure wrote the current version."

The graphical query of Figure 12 finds the functions satisfying the above description. The first graph simply adds to the existing link between two successive versions a new link indicating who is the author of the second version. The second graph looks for the current version of functions F1 that implement, directly or indirectly, the security-spec specification and are preceded by a chain of versions satisfying the rather vague criteria the user has in mind. Note that, even though the system does not support approximate search, the flexibility of regular expressions does provide some of the power of approximate matching. In particular, the ocurrence of an underscore in a regular expression involving closure means that we are looking for an arbitrary sequence of values along a path. □

HAM Versioning

Versioning is an important feature in hypermedia systems [Hala88, Garg88]. The specific details of versioning mechanisms differ from system to system. An advantage

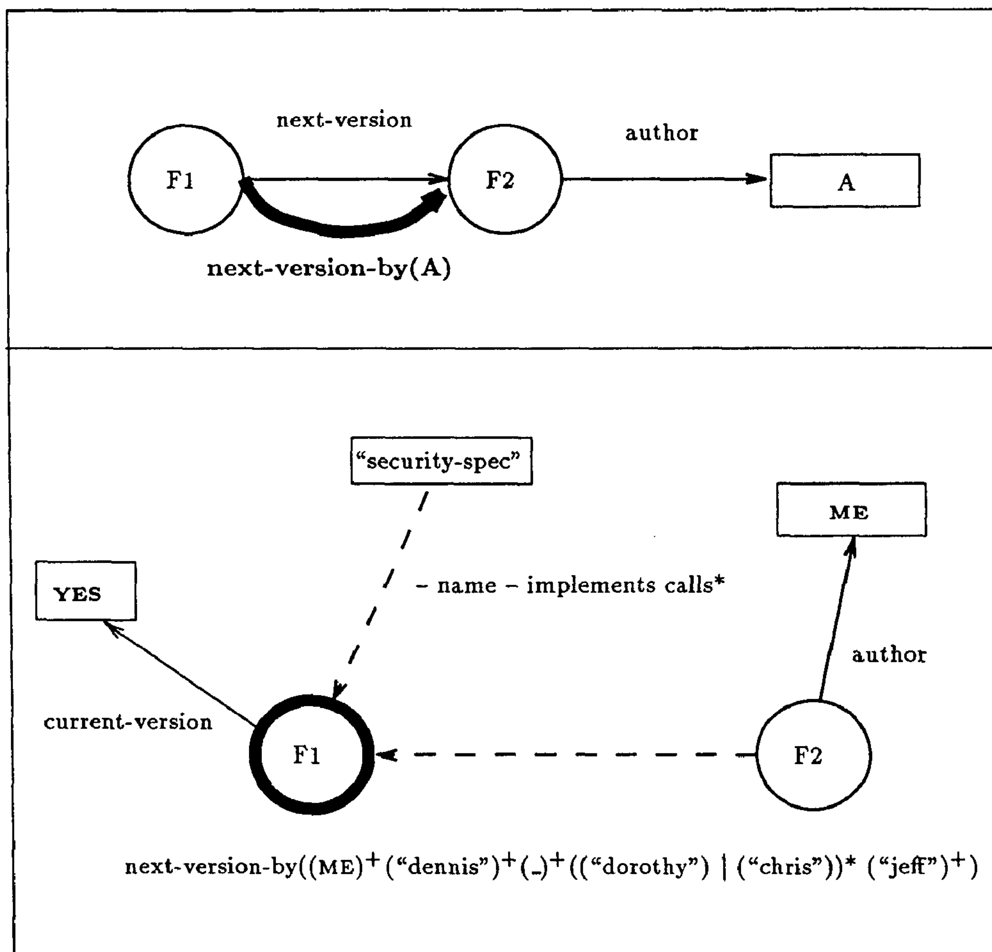

Figure 12: Querying the versioning mechanism.

of a query language like GraphLog, that has the capability to describe revision sequences, is that it can specify quite clearly these details as well as implement variations of the versioning policies originally provided by the hypertext system.

Example 8: In the HAM, a link may or may not keep linking the most recent versions of the nodes it connects. This is a user's choice, selected by setting to YES or NO a keep-up-to-date link attribute defined by the system. Keeping links up to date is the default policy, while not doing so is useful to retain links between previous (fixed) node versions.

Figure 13 describes the "keep up to date" policy for link versions in the HAM. To simplify the graphical query, current-version is defined simultaneously for links and nodes in the first query graph. The second query graph defines the from relation between a link and its start-point node to be kept up to date with the last node (and link) version if the keep-up-to-date attribute is set to YES. The third query graph contemplates the situation where no "keeping up to date" is desired. □

Dynamic Medical Handbook

In the Dynamic Medical Handbook [Fris88b, Fris88a] a content-based query is given as a set of keywords and the sectioning structure of the hyperdocument is used to help find the best starting points (i.e., either a chapter, section, subsection, and so on) for the interested reader. This search mechanism constitutes an interesting and non-trivial example of the combination of content-based and structural search. The algorithm first assigns an intrinsic weight for each card and keyword that is directly proportional to the number of keyword occurrences in the card and inversely proportional to the total number of occurrences of a keyword in the whole hyperdocument. Then a total weight is recursively propagated from the leaves to the root of the sectioning structure. The contributions of subsections to sections decrease exponentially with their distance in the sectioning hierarchy.

Example 9: The selection of the best starting points for the content-based query (i.e., those nodes with highest total-weight) can be expressed in GraphLog as shown in Figure 14.

We assume that a weight is associated with each node by the link weight and that the relation section-of describes a tree. Note the notation [C] following the section-of* expression. This means that variable C will store the length (number of edges) of each path that matches the regular expression. With this in mind, the query can be interpreted as follows. For each document component N2, for each section or sub-section N1 of N2 that is at distance C from N2 in the document tree and has weight W, add $W/(2^C)$ to the total weight of N2. □

Note that, as our queries become more ambitious, their representation in GraphLog becomes more complex. We do not envision an end user composing queries like the ones in Figures 13 or 14; rather, GraphLog could be used as a tool to allow a system

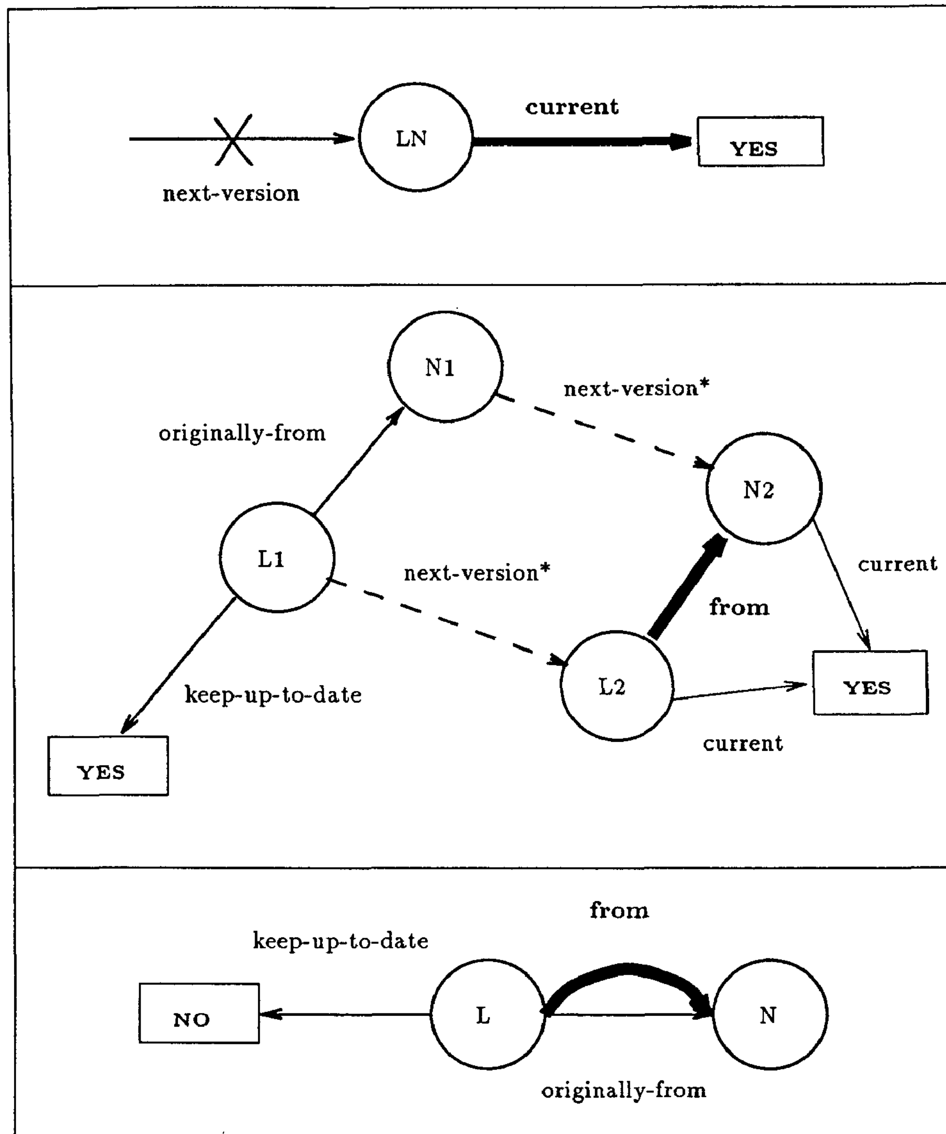

Figure 13: Keep up to date link version policy.

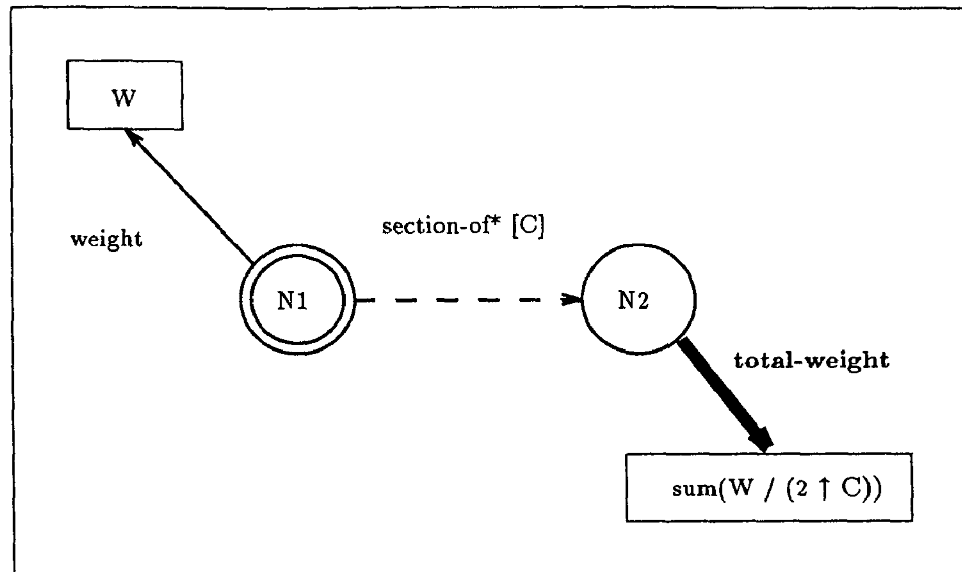

Figure 14: Selecting the optimal starting points in content-based search.

designer to provide a useful repertoire of "canned" queries, in much the same way that database query languages are used in large information systems.

HyperCard

We conclude this section by considering the application of **GraphLog** to the extremely simple model of Apple HyperCard. Stacks are composed of cards, and cards are related by links whose only property is the icon associated with the "button" at the start-point of the link. An interesting version of **GraphLog** can be adapted to this model that, although limited, will be a convenient improvement to the information retrieval capabilities of HyperCard. Instead of using symbolic labels on the edges, we can label them with the iconic button corresponding to the link. Regular expressions can be used as before. In addition, nodes in query graphs denoting stacks can be represented by the corresponding icons rather than by simple circles or rectangles.

This simple "**GraphLog** on HyperCard"query language has potential for two important extensions. The first is combining HyperCard content search with the structure search represented by a query graph; i.e., strings that must be present in the text fields of cards can be associated with the corresponding nodes in the query graph. The second extension is to allow, in addition to the specification of strings that must be present in cards, the invocation of an arbitrary HyperTalk script[2] in association with a node in a query graph that will allow the specification of more complex conditions that must be satisfied by the corresponding cards.

[2]HyperTalk is a special purpose programming language that provides extensibility to the HyperCard system.

PROTOTYPE IMPLEMENTATION

This section describes an ongoing implementation of **GraphLog**. The prototype is actually based on the earlier language G^+ and is being extended to handle full **GraphLog**. We refer to this system as the G^+ Prototype.

The original effort consisted in the specialization of a Smalltalk-80™ [Gold83, Gold84] graph editor product (NodeGraph-80 [Adam87]) for editing query graphs and displaying database graphs. The resulting editor supports graph "cutting and pasting", as well as text editing of node and edge labels, node and edge repositioning and reshaping, storage and retrieval of graphs as text files, etc.

Once the Graph Editor was available, the Query Evaluation component was developed to support G^+ edge queries. These are simple queries containing two nodes with one (possible dashed) edge connecting them, labelled with an arbitrary regular expression. The algorithms used to search the database for answers are discussed in [Mend89].

In Figure 15 the small G^+ GraphEditor window at the top of the screen contains an edge query. The large G^+ Graph Editor window shows the flights hypertext database mentioned in the section that introduced **GraphLog**.

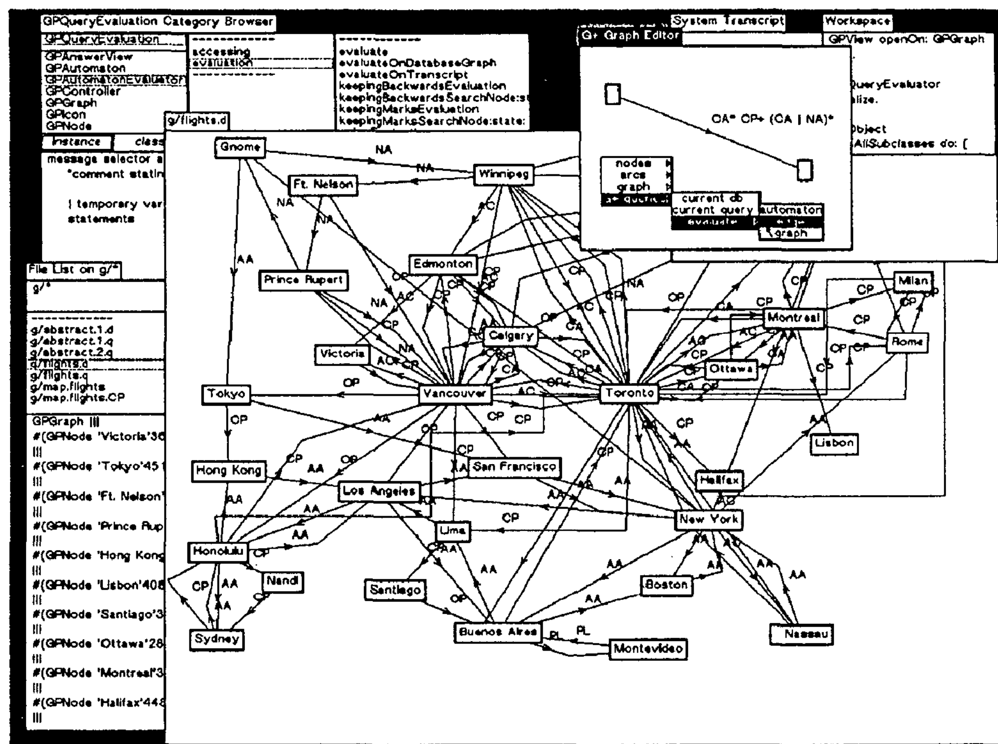

Figure 15: Invoking the evaluation of a G^+ edge query.

Figure 16 shows a screen dump displaying one of the answers of the query in Fig-

ure 15 by highlighting one path described by the query on the database graph.

Figure 16: Displaying the answers of a G+ edge query.

Rather than viewing the answers superimposed on the database graph, the user may choose to view them in a separate window called a graph collection browser. Figure 17 shows one such window. The left pane contains a list of paths that were found to satisfy the query, represented by their end points, and one such path is being displayed in the right pane.

Finally, the user may select to collect all the answers together into one new graph. This graph in turn may be queried, providing a mechanism for iterative filtering of irrelevant information until a manageable subgraph is obtained. Figure 18 shows an answer graph in which only American Airlines flights have been retained.

The current state of the prototype consists of the ongoing implementation of a query interface for the GraphLog query language on top of the Neptune hypertext system front-end to the Hypertext Abstract Machine (HAM) [Deli86]. The HAM is a general-purpose, transaction-based, multi-user server for a hypertext storage system. The HAM model is general enough to implement any current hypertext system [Camp87].

Figure 19 is a diagram of the G+ Prototype architecture. The G+ Query Interface and the G+ Query Evaluation constitute the implementation of the G+ Prototype. So far, the evaluation is being prototyped as a Smalltalk-80 program running as a client of the HAM within the Neptune environment. A more mature implementation

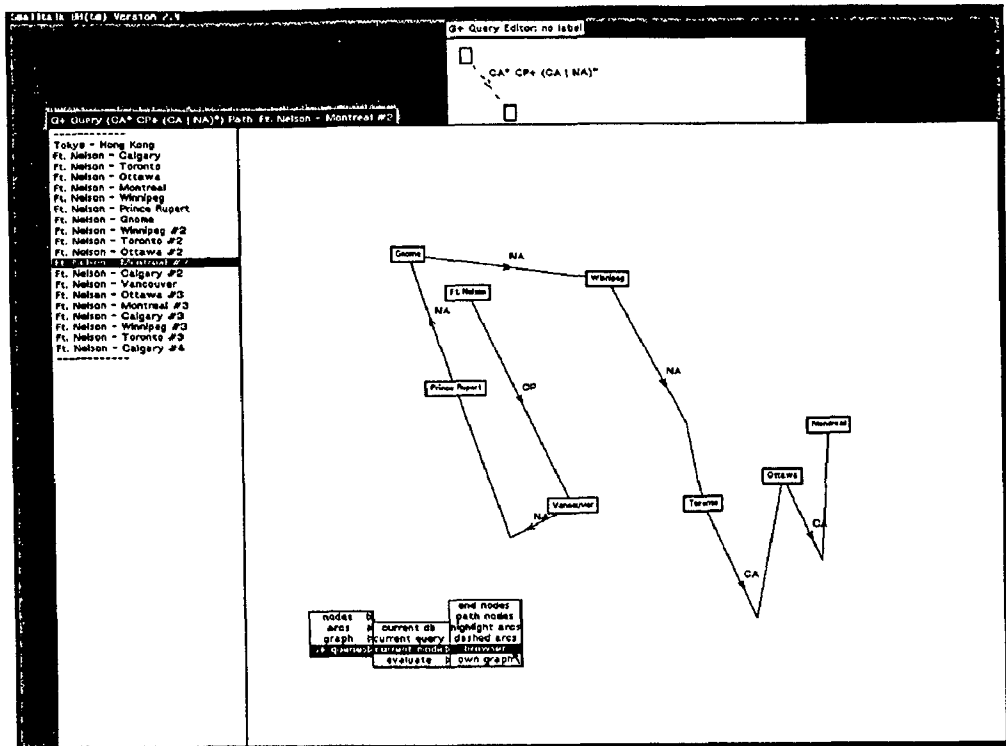

Figure 17: Displaying the answers in a browser.

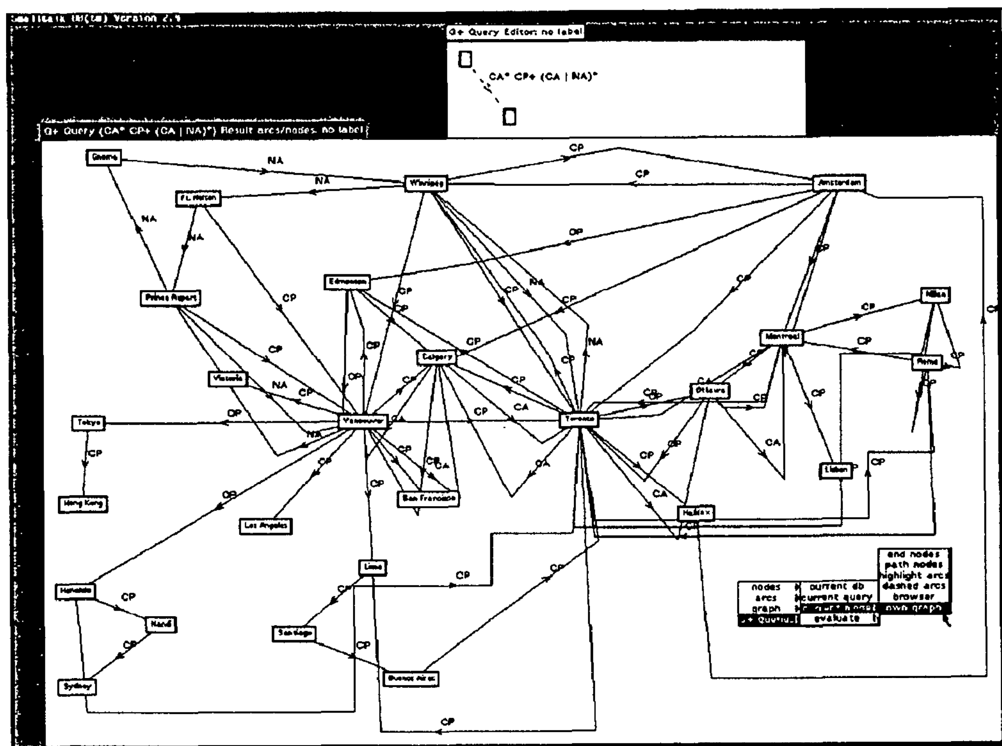

Figure 18: Displaying the answers in a single graph.

will have the evaluation components down-loaded to the HAM server.

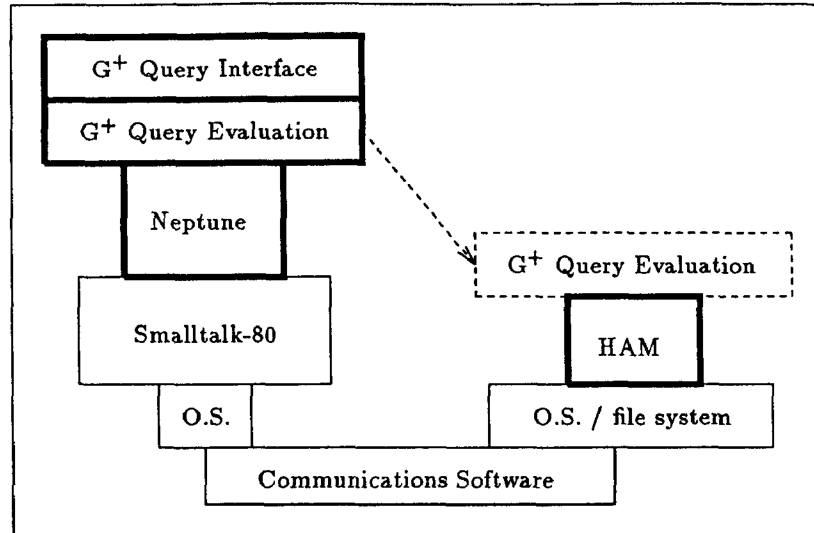

Figure 19: G^+ Prototype architecture.

CONCLUSIONS

We have described a powerful structural query language for hypertext. The language can express a large variety of queries that arise naturally in several different hypertext systems. It has a sound theoretical basis taken from the theory of database query languages and logic programming, but is visually oriented and avoids explicit use of logic formulae or recursion.

The next step in the development of GraphLog should be to integrate structure-based search with content-based search. Since regular expressions are already an essential part of the language, it is natural to do this by allowing each node in a query graph to be qualified by searching its contents for substrings matching a regular expression. This can be combined with value-based queries on node attributes for a completely general query language.

ACKNOWLEDGMENTS

The authors are grateful to Fred Lochovsky for his helpful comments and to Christine Knight and Frank Eigler for their contributions to the prototype.

REFERENCES

[Adam87] Sam S. Adams. *NodeGraph-80 Version 1.0.* Knowledge Systems Corporation, 1987.

[Aho79] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proc. 6th ACM Symp. on Principles of Programming Languages*, pages 110–120, 1979.

[Bege88] Michael L. Begeman and Jeff Conklin. The right tool for the job. *BYTE*, pages 255–266, October 1988.

[Bige88] James Bigelow. Hypertext and CASE. *IEEE Transactions on Software Engineering*, pages 23–27, 1988.

[Bige87] James Bigelow and Victor Riley. Manipulating source code in DynamicDesign. In *Hypertext'87 Workshop*, pages 397–408, 1987.

[Camp87] Brad Campbell and Joseph M. Goodman. HAM: A general-purpose hypertext abstract machine. In *Hypertext'87 Workshop*, pages 21–31, 1987.

[Cons89] Mariano P. Consens. Graphlog: "real life" recursive queries using graphs. Master's thesis, Department of Computer Science, University of Toronto, 1989.

[Deli86] N. Delisle and M. Schwartz. Neptune: A hypertext system for CAD applications. In Carlo Zaniolo, editor, *Proceedings of ACM-SIGMOD 1986 International Conference on Management of Data*, pages 132–142, 1986.

[Fris88a] Mark Frisse. From text to hypertext. *BYTE*, pages 247–253, October 1988.

[Fris88b] Mark Frisse. Searching for information in a hypertext medical handbook. *Communications of the ACM*, 31(7):880–886, 1988.

[Garg88] Pankaj K. Garg. Abstraction mechanisms in hypertext. *Communications of the ACM*, 31(7):862–879, 1988.

[Gold84] Adele Goldberg. *Smalltalk-80: The Interactive Environment.* Addison-Wesley, 1984.

[Gold83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Hala88] Frank G. Halasz. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM*, 31(7):836–852, 1988.

[Hala87] F.G. Halasz, T. P. Moran, and H.R. Triggs. NoteCards in a nutshell. In *ACM Conference of Human Factors in Computer Systems*, pages 45–52, 1987.

[Klug82] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[Mend89] A.O. Mendelzon and P.T. Wood. Finding regular simple paths in graph databases. In *Proc. 15th International Conference on Very Large Data Bases*, 1989.

[Ullm88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Potomac, Md., 1988.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 089791-339-6/89/0011/0292 $1.50