Published in final edited form as:

AVI. 2016 June ; 2016: 272–279. doi:10.1145/2909132.2909246.

VISAGE: Interactive Visual Graph Querying

Robert Pienta,
Georgia Tech

Shamkant Navathe,
Georgia Tech

Acar Tamersoy,
Georgia Tech

Hanghang Tong,
Arizona State University

Alex Endert, and
Georgia Tech

Duen Horng Chau
Georgia Tech

Abstract

Extracting useful patterns from large network datasets has become a fundamental challenge in many domains. We present VISAGE, an interactive visual graph querying approach that empowers users to construct expressive queries, without writing complex code (e.g., finding money laundering rings of bankers and business owners). Our contributions are as follows: (1) we introduce *graph autocomplete*, an interactive approach that guides users to construct and refine queries, preventing over-specification; (2) VISAGE guides the construction of graph queries using a data-driven approach, enabling users to specify queries with varying levels of specificity, from concrete and detailed (e.g., query by example), to abstract (e.g., with "wildcard" nodes of any types), to purely structural matching; (3) a twelve-participant, within-subject user study demonstrates VISAGE's ease of use and the ability to construct graph queries significantly faster than using a conventional query language; (4) VISAGE works on real graphs with over 468K edges, achieving sub-second response times for common queries.

Keywords

Graph Querying and Mining; Visualization; Interaction Design

CCS Concepts

•**Human-centered computing** → **Visual analytics;** •**Information systems** → *Structured Query Language;*

1. INTRODUCTION

From e-commerce to computer security, graphs (or networks) are commonly used for capturing relationships among entities (e.g., who-buys-what on Amazon, who-called-whom networks, etc.). Finding interesting, suspicious, or malicious patterns in such graphs has been the core enabling technologies for solving many important problems, such as flagging "near cliques" formed among company insiders who carefully timed their financial transactions [27], or discovering "near-bipartite cores" formed among fraudsters and their accomplices in online auction sites [21]. Such pattern-finding process is formally called *graph querying* (or *subgraph matching*) [29, 28].

Many graph databases now support pattern matching and overcome the prohibitive costs of joining tables in relational databases [31]. Specifying graph patterns, unfortunately, can be a challenging task. Users often need to overcome steep learning curves to learn querying languages specific to the graph databases storing the graphs.

For example, many graph databases store graphs in the Resource Description Framework (RDF) format, which capture subject-predicate-object relationships among objects[1]. These systems support the SPARQL querying language, which is hard to learn and use [10]. The Cypher language, designed for the recent Neo4j graph database[2], is easier to work with since its syntax more closely resembles SQL [13, 15], but expressing relationships among nodes can still be challenging and may require writing many lines of code even for conceptually simple queries [14], as demonstrated in Figure 1, which seeks a "triangle" of three similar action films from the 1980's [23].

While there has been a lot of work in developing querying algorithms (e.g., [29, 28, 23]), there has been far less research on understanding and tackling the visualization, interaction, and usability challenges in the pattern specification process. Studying the user-facing aspects of subgraph matching is critical to fostering insights from interactive exploration and analysis. While early works suggested such potential [7, 3, 25], none evaluated their ideas with users. Hence, their usability and impact are not known.

We propose VISAGE, the **Vis**ual **A**daptive **G**raph **E**ngine[3], which provides an adaptive, visual approach to graph query construction and refinement, to simplify and speed up graph query construction (Figure 3). VISAGE performs exact graph querying on large graphs and supports a wide variety of different node types and attributes.

Our main contributions are:

- We introduce an interaction technique for graphs called *graph-autocomplete* that guides users to construct and refine queries as they add nodes, edges, and conditions (feature constraints). Adding too many nodes, edges, or conditions may result in over-specification (too few results) or even a null-result (no results

[1] http://www.w3.org/RDF/
[2] http://neo4j.com/
[3] Please see video-demo in supplementary material.

found) [1]. Graph-autocomplete stops the user from constructing null-result-queries and guides the query-specification process.

- We design and develop a system that utilizes recent advances in graph-databases to support a spectrum of querying styles, from *abstract* to *example-driven* approaches, while most other visual graph querying systems do not [7, 3]. In the abstract case, users start with a very abstract query and narrow down the possible results by providing feature and topological constraints. In the example-driven case, often called *query by example* (QBE) [32], users can specify an exact pattern and abstract from that pattern into a query of their choice. This technique allows users to start from an example or keep a value fixed in their query. In VISAGE, the user can *star* a node to fix its place in the query and across all of the results. We provide examples of both query-construction approaches in the Scenario Section.
- We demonstrate VISAGE's ease of use and the ability to construct graph queries significantly faster than conventional query languages, through a twelve-participant, within-subject user study.

2. SCENARIO

We provide two scenarios to illustrate how users may use VISAGE. The first scenario starts from a general question with a known structure and narrows the search through query refinement. The second scenario begins with a known example from which new similar results are found through abstraction.

The Rotten Tomatoes Movie Graph

Throughout this work, we use a Rotten Tomatoes[4] film-actor-director graph. The graph has 58,763 nodes: 17,072 films, 8,576 directors, and 33,115 actors. There are over 468,592 undirected edges of three types: (1) film to film edges, based on Rotten Tomatoes' crowd-sourced similarity; (2) film to actor edges, showing who starred in what; (3) film to director, showing who directed what.

Scenario 1: From Abstract to Detailed

Imagine our user Lana wants to find co-directors who have starred the same actor in two films. She can begin specifying her query starting with very general terms. She right clicks the background and chooses to add a new director node, she repeats this to add another director, and again to add two films and an actor. She attaches the director to the films and the films to the actor (see Figure 2.1), by clicking and dragging from one node to the other (one pair at a time). She clicks the search button. She gets the results in the results list, we show only the first result (in Figure 3.2) to save space. She likes the first result (in Figure 2.2) with the Coen Brothers, *The Big Lebowski, O' Brother Where Art Thou?*, and John Goodman. Realizing that she enjoys the work of the Coen brothers, she stars both director nodes and *O' Brother Where Art Thou?*, making them fixed values in the query. She

[4] A movie review website. http://www.rottentomatoes.com/

performs the search again with these values fixed. The query is now looking for any actor cast by the Coen brothers that was in *O' Brother Where Art Thou?* and any other Coen film. She receives the result, in Figure 2.3, showing George Clooney in *Intolerable Cruelty*.

Scenario 2: Building up From a Known Example

Now consider the example-driven approach, where a user, Barry, takes a known example and abstracts it into a new query. Barry knows that Matt Damon and Ben Affleck both starred in *Good Will Hunting*, so he draws a node for each person and one for the film, and connects each actor to the film (see Figure 2.4). Specific nodes can be added manually by searching for them in the node search menu (see Figure 4.1). When a specific node is added via search, it's automatically *starred* so that its value will remain fixed in the query. Nodes can be unstarred by clicking the star icon in the upper right corner (see the star by Matt Damon in Figure 2.4). Because Barry starred all the nodes in his query, searching only finds one result (if Barry was wrong and his initial example does not exist, no results will be shown and he will be alerted via text in the empty results panel). By specifying the exact value of the nodes, the query has become too specific and will need to be abstracted if Barry wants more results. Barry then *unstars* the specific film *Good Will Hunting*, to find any movie starring both actors (Figure 2.5). Barry can also leverage the visualized features of *Good Will Hunting* to specify new constraints based on the results (i.e., only selecting movies made in the 1990's co-starring the actor duo). He uses a visualization of the possible constraints discussed in Section 4.1 and shown in Figure 5. Barry reissues his search and finds *Dogma* (among others), a potentially exciting film for him to watch.

3. VISAGE OVERVIEW

The user interface for VISAGE is comprised of a force directed query-graph visualization (Figure 3), a context menu that provides an overview of features (Figure 3.2 in blue), a feature exploration pane (Figure 5), and a results list (Figure 3.2). The graph view shows the current state of the user's query. Matches are found in the background during interaction with the feature tree and query construction, but can be fetched manually using the “Find Matches” button at the top. Results are displayed in a popup list (Figure 3.3) which can be removed by clicking “Clear Results” at the top.

When users select a node, a blue border appears along with the node context menu (Figure 3.2). The context menu shows the current selected feature constraints in green (if the user wants the selected movie to only have good ratings then they can select this constraint in the feature tree in Figure 3.3). When a result is selected, a summary of the current node's features is shown in blue. If a particular node value from the data has been *starred*, its value in the query is fixed and can take only that specific value during the querying. Starred nodes have a golden star in the upper right (Matt Damon in Figure 2.4) and an additional context menu that reminds the user that the film is starred.

Adding new nodes is streamlined via our node tray, which is brought up by clicking the “+” icon on an existing node or right clicking on the background (see Figure 4.2). This menu displays the types of nodes that, if added, guarantee at least one match in the underlying network. Each node shows a pin, a star and a magnifying glass when moused over. The pin spatially pins the node and the star allows users to star the node, keeping it constant in the query. The magnifying glass opens the node search menu, in Figure 4.1, which allows users to search for particular nodes via text. Users can quickly and easily add known values and pin them; facilitating QBE-like query construction.

VISAGE Querying Language

VISAGE allows the user to form complex graph queries, where the nodes can be as abstract as a wildcard or as constrained as taking a single value (recall the scenario with the Coen brothers in Figure 2). Graphs with a known type, for example a film, can have any number of additional constraints added to them; limiting the possible matches in the underlying dataset. The feature-tree allows users to explore hierarchical and non-hierarchical features (for both categorical and continuous variables).

4. DESIGN RATIONALE

Supporting Expressive Querying: Abstract to Specific

Graph querying requires the user to specify a group of nodes and their relationships; however, the constraints placed on the nodes can range wildly, from specific to abstract (e.g., a wildcard node of any type). A key design goal was to allow users to express their queries ranging from abstract to very specific. Users may start from known examples and abstract based off of the features of their example.

We are able to leverage the internal capabilities of Neo4j in terms of query conditions and indexing to support more complex queries. We support true wildcards (which can take on any node type and value). We use indices to support constant-time lookup for all starred nodes. Conditions are added by clicking on that value in the feature hierarchy. Users are free to add as many as they like. Within each feature (whether flat or hierarchical), we employ the logical *or* operation for constraints (i.e., year = 1997 **or** year = 1998). Across the features, we use a logical *and* for the constraints (i.e., genre = horror **and** year = 1988).

4.1 Improving Visual Query Refinement: Autocomplete

Graph autocomplete has two primary goals (1) keeping the user from making queries with no results and (2) helping them understand the features of the matches of their query during refinement.

Structural Guidance—When a user submits an over-specified query (one that has too few or no matches), they must return to their query and refine it until they reach a suitable level of specification. To help avoid this, we adapt the query construction process based on the query the user is constructing. VISAGE directs the user's query construction towards results by providing critical information about possible nodes and their features. We have created the first graph-querying *autocomplete*, which works on **node types**. We want to limit the types a nodes users can add so that their query always has at least one result. This guides the user in the direction of queries that are rich in matches and away from over-specification and null-results [1].

We achieve type- or structure-autocomplete by constraining the possible-neighbor options in the new node menu. By querying in the background we determine which types of newly added nodes and edges will result in matches and which won't. VISAGE displays this data by desaturating the add node button. This way a user can immediately see which types of nodes are available to them. In the case of truly massive graphs, background querying for node-types may be too slow. In this case, we use first-*k*-sampling to guide the user. We use the first *k*-results of the current query to determine the feasibility and distribution of potential new nodes given the current query. The samples are visualized in the bar graph to the right of the node-buttons (Figure 4.1).

Feature Guidance—Graph-autocomplete also works in the feature space. We do this by visualizing the distributions of different node-attributes, from a sample of the results, of the current query. This approach provides users with detailed information about how the features (of their current queries) are distributed. With knowledge of different attributes, users are able to better understand how the results fill out the feature space. These data provide a visual cue that indicates how a new condition will change the number of results.

We chose to visually encode the feature frequencies in the edges of the feature-tree with edge-width and saturation (Figure 5.2). By adding constraints with sparser features (thin, light lines in Figure 5.2), users will quickly decrease the number of matches. If users choose denser attributes they will constrain their search less, keeping more of the results. The feature tree also promotes abstraction in hierarchical attributes (Figure 5.3), because it is straightforward to trace from one constraint up to the parent constraint. For example, instead of looking only for films from 1993, the user can move up the hierarchy seeking films from the 1990's. Feature-autocomplete gives users the summary feature information needed to narrow their search without having to repeatedly go back and forth from query to results.

5. IMPLEMENTATION

VISAGE uses a client-server architecture (Figure 6) that separates the front-end interactive visualization (client) from the backend graph matching and storage (server). We have designed VISAGE to be independent of its backing graph database. VISAGE fully supports Neo4j [19]. Currently, it also partially supports SPARQL, with full support in the near future. VISAGE's web client (Javascript and D3) and server (Python) can run smoothly on the same commodity computer (e.g., we developed VISAGE on a machine with Intel i5-4670K 3.65GHz CPU and 32GB RAM). Optionally, for larger graphs, the server may be run on a separate, more powerful machine.

To fetch results of a user's query, we convert and parse the visual query into a compact format that we pass off to the DB modules which convert the parsed query into the necessary languages for each graph database. Once results are returned, we calculate summary statistics with the metadata extractor in Figure 6, which are the input for graph-autocomplete and represent the results of the current query.

Parsing A Graph Query

When looking for matches of a query, if the starting node has very few matches in the graph, the search space is reduced and fewer comparisons are needed. The effect can be enormous, reducing a multiple minute query down to sub-second times. Because the node-constraints can vary from completely abstract (like a wildcard) to a single specific node, we have designed VISAGE to partition the graph queries into pieces. Our first step is to rank the nodes by the number and severity of their constraints. Starred nodes are parsed into subqueries first. VISAGE then ranks the remaining nodes by number of constraints. The entire parsing can often be completed in a few milliseconds or less.

6. USER STUDY

To evaluate VISAGE's usability, we conducted a laboratory study to assess how well participants could use VISAGE to construct queries on the Rotten Tomatoes movie graph discussed earlier. We chose a movie graph, because the concept of films, directors, etc., would be familiar to all participants, so that they could focus on VISAGE's features. We asked participants to build queries to find interesting graph patterns derived from prior graph mining research [17, 23, 8]. We compared the time taken forming queries between VISAGE and Cypher; we chose Cypher for its resemblance to SQL and ease of use. We chose Participants were not informed which system, if either, was developed by the examiner. We are not able to compare with GRAPHITE [7], as it is not publicly available.

6.0.1 Participants—We recruited 12 participants from our institution through advertisements posted to local mailing lists. Their ages ranged from 23 to 41, with an average age of 27. 5 participants were female the rest were male. All participants were screened for their familiarity with SQL. Participants ranged in querying skills; three had prior experience with Cypher. Each study lasted for about 60 minutes, and the participants were paid $10 for their time.

6.0.2 Experiment Design—Our study uses a within-subjects design with two main conditions for completing tasks: **VISAGE** and **Cypher**. The test consisted query tasks which were divided into two sections (see the *Task* section). Every participant completed the first section of tasks in one condition, and the second set of tasks in the remaining condition. The order of the conditions was counterbalanced. We generated matched sets of tasks, Set A and B, each with 5 tasks to complete. The tasks ranged from easy to hard and were counterbalanced with each condition, to even out unintended differences in difficulty among the tasks. We used two sets of tasks to ensure that participants did not remember their solutions between sets.

6.0.3 Tasks—We created the tasks based on an informal survey of interesting patterns from common questions people formed when exploring Rotten Tomatoes data and from prior graph mining research [17, 23, 8]. The tasks in Task Set A (shown in Figure 7) were:

1. Find films similar to any film from *1993*.
2. Find an actor and a director for any *drama* film.

3. Find an actor starred in 3 films: *romance*, *comedy*, and *action*.
4. Find 3 similar *action* films, where one is from the 80's, one from the 90's, and the last from the 00's.
5. Find co-directors who made at least three films together, starring the same actor, where one of the films was *from the 90's*.

The difficulty of each query increases from task 1 through 5. We ranked the difficulty of each task based on the amount of Cypher code and number of nodes, edges, and constraints needed. The italic values shown above were the elements that differed between the two task sets (the order remained the same across both sets). Our hypothesis was that Cypher and VISAGE would achieve approximately similar performance for easy queries and VISAGE would achieve shorter task completion times for harder queries.

Task completion time was our dependent measure. Task completion time could be affected by: (1) Software – VISAGE or Cypher; (2) Task Set – the Task Set A or B; (3) Software Order – which software was used first. Using a Latin square design, we created 4 participant groups (since all subjects would do both software systems). We randomly and evenly assigned the participants to the groups, e.g., one group is *(*VISAGE + *Task Set A) then (Cyper + Task Set B)*.

6.0.4 Procedure—Before the participants were given the tasks, they were provided with instructions on the software that they would be using as well as information about the data set they would be exploring. For the Cypher querying language, we offered a tutorial for starting Cypher tailored to our dataset. For VISAGE we provided an overview of VISAGE's interface, how to construct queries, and how our tool would work. The participants were welcome to ask clarifying questions during these introductory periods.

Once demoed we moved on the the first block of tasks, where we instructed the participants to work quickly and accurately. They had 5 minutes to perform each task and could only move to the next one if they correctly completed the current task or ran out of time. After each task, the participant was given the next task's instruction while the system was reset. Each task was timed separately. If a participant failed to finish a task within the allotted 5 minutes (300 seconds), the experimenter stopped the participant, marked that task as a failure, and recorded 300s as the task completion time (to prevent participants from spending indefinite amounts of time on tasks).

Once participants had completed the first set of tasks, they were provided the next set. At the end participants completed a questionnaire that asked for subjective impressions about each software system.

6.1 Results

6.1.1 Quantitative Results—The task completion times were analyzed using a mixed-model analysis of variance with fixed effects for *software, software order, task set, and a random effect across participants.* This technique is used to analyze within-subject studies and improves over conventional ANOVA, because error-terms are also calculated per-subject [18].

We measured the task completion time for the effects over all possible combinations of software, software order, and task set. The only statistically significant effect was software, suggesting a successful counterbalancing of software order and the equality of the difficulty of the two task sets. Figure 8-left demonstrates the average time per task for the study. The software effect was significant across all tasks: task 1 ($F_{1,5} = 27.16$, $p < 0.0004$), task 2 ($F_{1,5} = 49.76$, $p < 0.0001$), task 3 ($F_{1,5} = 33.23$, $p < 0.0002$), task 4 ($F_{1,5} = 25.88$, $p < 0.0005$), task 5 ($F_{1,5} = 42.84$, $p < 0.0002$). Participants were significantly faster when constructing queries in VISAGE than in Cypher. Only one participant failed to complete task 5 in the allotted 5 minutes (using the Cypher software); the rest succeeded in all tasks. This datum is partially responsible for the high variance in the task 5 (see Figure 8-left - Task 5). Using VISAGE, participants were able to construct task 5 slightly faster than task 4 (see Figure 8-left). Adding new nodes in VISAGE is faster than specifying feature constraints; task 4 has a large number of constraints while task 5 has a large number of nodes and edges. We do not see this in Cypher task 4 and 5, because adding new edges, nodes, and constraints all take similar amounts of time. Overall, the average difference in task times between VISAGE and Cypher was statistically significant ($F_{1,5} = 37.38$, $p < 0.0005$); this represents an average speedup of about 2.67× when using VISAGE.

6.1.2 Subjective Results—We measured several aspects of both conditions using 7-point Likert scales filled out at the end of the study. Participants felt that VISAGE was better than Cypher for all the aspects asked about (see Figure 8-right). The participants enjoyed using VISAGE more than a querying language and additionally found that our system was easier to learn, easier to use and more likeable overall; although this is a common experimental effect, we find the results encouraging. Several participants found that the visualization of the query greatly improved the overall completion of the tasks.

6.2 Discussion and Limitations

The results of our user study were positive, both qualitatively and quantitatively. This suggests that VISAGE's visual representation of graph queries using graph autocomplete is faster than typed querying languages. We believe that VISAGE achieves these better times by: (1) streamlining the process of adding nodes and edges; (2) autocompleting partially-complete queries, which adaptively guides the user away from null-results; (3) shielding users from making typos and mistakes during the construction of their queries.

Adding nodes and edges in traditional querying systems often requires creating a variable for them, which must be remembered in order to specify the structure and attributes related to it. The user may have to type the name a single node repeatedly in order to specify the actual structure and in the case of large queries may confuse the names of nodes. VISAGE's visual representation simplifies this considerably. Typos and mistakes are common when writing a long and complicated query by hand. By programmatically generating queries based on users' constructions, VISAGE avoids the delay incurred by typos.

We observed two general strategies that participants employed when constructing queries: (1) entities first, then relationships, and (2) iterative construction. Participants in the first group would often add all the entities from the task first, then wire up the relationships. Other participants followed a more iterative approach, wherein they would start with a single entity and build up from it (reminiscent of a breadth-first search). No statistical significance was found in the time taken for each general strategy.

When users construct queries with null-results, a traditional system requires the user to wait while the search is performed. During the study two of the users stated that the autocomplete helped remind them about the underlying structure of the network, saving time during their tasks. We help guide the user away from this case with our graph-autocomplete, so that users spend less time debugging queries that do not produce results. Because we sample results for graph-autocomplete, VISAGE may be able to retrieve a small sample of the possible results in real time, leading to a potentially skewed samples. This limit is dictated by the underlying graph database and scales accordingly.

While the results of our evaluation was positive, the need for the participants to build queries was created by the tasks; however, in real-world scenarios, such needs would be ad hoc. For example, what kind of exploratory query patterns do people create? We plan to study such needs, as well as how VISAGE can handle those kinds of tasks, in less controlled situations.

7. RELATED WORK

Graph Visualization and Query Languages

Many tools and techniques have been developed to facilitate discovery in graphs; Herman et al. [11] cover much of the initial work in graph visualizations. Static [30] and dynamic [2] graph visualizations are quickly growing areas; as well as graph sensemaking [22]. Our work extends this body of research by providing an adaptive, visual approach to graph query construction and refinement.

Database researchers have proposed graphical query languages to help users specify queries against various forms of databases. The seminal work by Zloof [32] introduces the Query By Example language (QBE). QBE allows users to formulate queries by filling out relational templates, constructing “example queries”, rather than writing traditional SQL queries. Other examples include PICASSO [16], which allows users to pose complex queries without knowing the details of the underlying database schema, and the concept of dynamic queries [1, 26], which allow users to create relational queries with graphical widgets to provide visual display of actions. Catarci et al. [5] provide a survey of the body of work focusing on relational databases. More recently, researchers proposed graphical query languages for XML [6, 20] and RDF [12] databases as well. Our work builds upon much of this previous work; however, our focus is on the visual aspect of query construction and refinement as well as displaying results without requiring familiarity with a data-model like XML or RDF.

Approaches for Graph Querying

The problem of querying a large graph given a subgraph of interest, also known as subgraph matching, has been investigated in several recent works. Tong et al. propose G-Ray [29], which is a best-effort inexact subgraph matching approach that supports node attributes. The MAGE algorithm [23] improves G-Ray by exhibiting lower latency, supporting attributed edges, wildcards, and multiple attributes. Tian and Patel propose TALE [28], which is an index-based method that incorporates the local graph structure around each node into an index structure for efficient approximate subgraph matching. Other systems have solved this problem, like OntoSeek [9], which utilizes inexact graph matching based on linguistic ontologies over a large collection of keywords called WordNet. While we currently utilize Neo4j's graph querying functionality, our approach can very well work with any of the advanced techniques in this line of research. Recently Cao et al. introduced g-Miner, an interactive tool for graph mining that supports template matching and pattern querying [4]. VISAGE bridges querying and pattern matching, by offering multiple levels of abstraction, where g-Miner does not.

Visual Graph Querying

A few systems have been proposed for visually querying large graphs. One example is GRAPHITE [7], which allows users to visually construct a graph query over categorically attributed graphs. It uses approximate subgraph matching and visualizes the results. GRAPHITE proved that visual graph querying is possible; however, our focus is on the query refinement process (with richer querying possibilities than GRAPHITE, which only supports a single categorical attribute per node). More recently, researchers proposed VOGUE [3], which is a query processing system with a visual interface that interleaves visual query construction and processing. VOGUE exploits GUI latency to prune false results and prefetch candidate data graphs through special indexing and query processing schemes. Our work differs from this body of work by enabling users to explore the feature space with a tree-based view and guiding users as they construct their graph query with graph-autocomplete. Previous works on visual graph querying [3, 7, 24], did not focus on addressing the interaction and visualization challenges, which is another focus of our work here.

Graph Summarization

Another line of research focuses on "summarizing" a given graph. Koutra et al. [17] propose VoG, which constructs a vocabulary of subgraph-types like stars and cliques. Dunne and Shneiderman [8] present motif simplification, which is a technique for increasing the readability of node-link network visualizations by replacing common repeating network motifs with easily understandable motif glyphs (e.g. fans and cliques). Schreiber et al. used this idea with MAVisto, a tool for the exploration of motifs in biological networks [25]. We do not focus on graph summarization in this paper; however, many of the patterns or motifs serve as the basis for our user study tasks.

8. CONCLUSION & FUTURE WORK

In this work we presented VISAGE, a system built using recent innovations in graph-databases to support the visual construction of queries, from abstract structures to highly conditioned queries. VISAGE relies on an interaction technique for graphs called *graph-autocomplete* that guides users to construct and refine queries, preventing null-results. We hypothesized that visual graph querying with VISAGE would be faster for generating queries than the Cypher querying language. We demonstrated this with a twelve-participant, within-subject user study. The study showed that VISAGE is significantly faster than conventional querying for participants with or without familiarity to Cypher.

VISAGE offers users a visually supported, code-free solution to graph querying that helps guide the user towards queries with results. Currently we do not support graph subqueries, unions and intersections (of graph results), aggregations, shortest-paths, and edge attributes. This work has revealed additional challenges and potential new questions for the community. How can inexact or approximate querying be used to aid query construction and refinement; and how best to visualize the uncertainty inherent in approximate results [22, 30]? We hope VISAGE will spur continued interest in visual graph querying.

References

1. Ahlberg, C., Williamson, C., Shneiderman, B. Proc CHI. ACM; 1992. Dynamic queries for information exploration: An implementation and evaluation; p. 619-626.
2. Beck F, Burch M, Diehl S, Weiskopf D. The state of the art in visualizing dynamic graphs. EuroVis - STARs. 2014:83–103.
3. Bhowmick SS, Choi B, Zhou S. Vogue: Towards a visual interaction-aware graph query processing framework. Proc CIDR. 2013
4. Cao, N., Lin, YR., Li, L., Tong, H. Proc CHI. ACM; 2015. g-miner: Interactive visual group mining on multivariate graphs; p. 279-288.
5. Catarci T, Costabile MF, Levialdi S, Batini C. Visual query systems for databases: A survey. Journal of Visual Languages & Computing. 1997; 8(2):215–260.
6. Ceri S, Comai S, Fraternali P, Paraboschi S, Tanca L, Damiani E. Xml-gl: A graphical language for querying and restructuring xml documents. Proc SEBD. 1999:151–165.
7. Chau, DH., Faloutsos, C., Tong, H., Hong, JI., Gallagher, B., Eliassi-Rad, T. Proc ICDM. IEEE; 2008. Graphite: A visual query system for large graphs; p. 963-966.
8. Dunne, C., Shneiderman, B. Proc CHI. ACM; 2013. Motif simplification: Improving network visualization readability with fan, connector, and clique glyphs; p. 3247-3256.
9. Guarino N, Content-based O, Vetere G, Masolo C. Ontoseek: Content-based access to the web. IEEE Intelligent Systems and Their Applications. 1999; 14(3):70–80.
10. Hakeem, A., Lee, MW., Javed, O., Haering, N. Proc Multimedia. ACM; 2009. Semantic video search using natural language queries; p. 605-608.
11. Herman I, Melançon G, Marshall MS. Graph visualization and navigation in information visualization: a survey. IEEE Transactions on Visualization and Computer Graphics. 2000; 6(1): 24–43.
12. Hogenboom, F., Milea, V., Frasincar, F., Kaymak, U. Emergent Web Intelligence: Advanced Information Retrieval. Springer; 2010. Rdf-gl: a sparql-based graphical query language for rdf; p. 87-116.
13. Holzschuher, F., Peinl, R. Proc Joint EDBT/ICDT Workshops. ACM; 2013. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j; p. 195-204.
14. Jindal, A., Madden, S. Proc Big Data. IEEE; 2014. Graphiql: A graph intuitive query language for relational databases; p. 441-450.
15. Kaur K, Rani R. Modeling and querying data in nosql databases. Proc Big Data IEEE. 2013
16. Kim H, Korth HF, Silberschatz A. Picasso: A graphical query language. Software: Practice and Experience. 1988; 18(3):169–203.
17. Koutra, D., Kang, U., Vreeken, J., Faloutsos, C. Proc SDM. SIAM; 2014. Vog: Summarizing and understanding large graphs; p. 91-99.
18. Littell RC, Milliken GA, Stroup WW, Wolfinger RD. SAS System for Mixed Models. 2006

19. Montag D. Understanding neo4j scalability. Technical report, Neo Technology. Jan.2013

20. Ni W, Ling TW. Glass: A graphical query language for semi-structured data. Proc DASFAA. 2003:363–370.

21. Pandit, S., Chau, DH., Wang, S., Faloutsos, C. Proceedings of the 16th international conference on World Wide Web. ACM; 2007. Netprobe: a fast and scalable system for fraud detection in online auction networks; p. 201-210.

22. Pienta R, Abello J, Kahng M, Chau DH. Scalable graph exploration and visualization: Sensemaking challenges and opportunities. Proc BigComp. 2015:271–278.

23. Pienta, R., Tamersoy, A., Tong, H., Chau, DH. Proc Big Data. IEEE; 2014. Mage: Matching approximate patterns in richly-attributed graphs.

24. Pienta, R., Tamersoy, A., Tong, H., Endert, A., Chau, DHP. Proc IUI. ACM; 2015. Interactive querying over large network data: Scalability, visualization, and interaction design; p. 61-64.

25. Schreiber F, SchwÃűbbermeyer H. Mavisto: a tool for the exploration of network motifs. Bioinformatics. 2005; 21(17):3572–3574. [PubMed: 16020473]

26. Shneiderman B. Dynamic queries for visual information seeking. IEEE Software. 1994; 11(6):70–77.

27. Tamersoy A, Khalil E, Xie B, Lenkey SL, Routledge BR, Chau DH, Navathe SB. Large-scale insider trading analysis: patterns and discoveries. Social Network Analysis and Mining. 2014; 4(1):1–17.

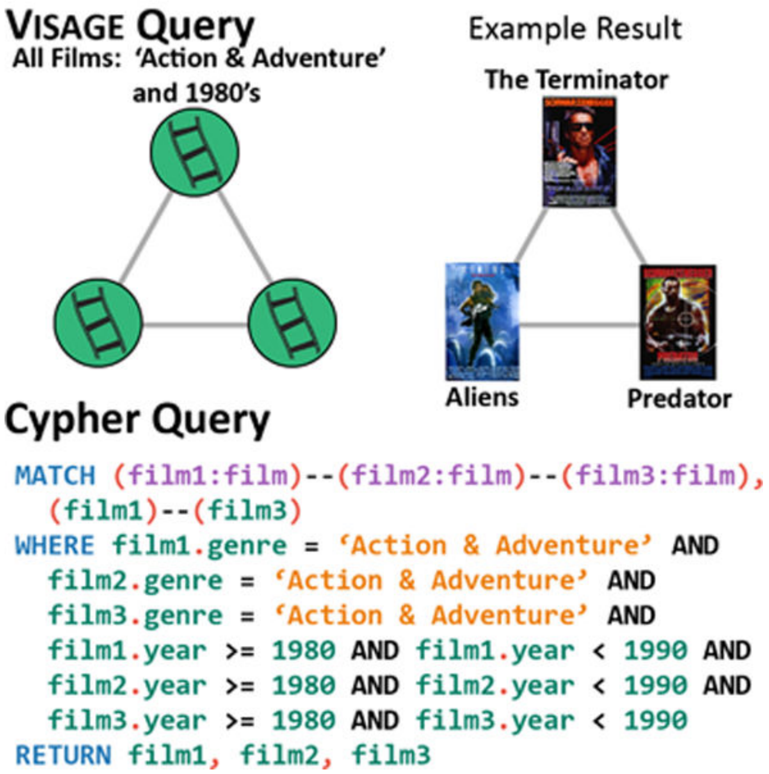
28. Tian, Y., Patel, J. Proc ICDE. IEEE; 2008. Tale: A tool for approximate large graph matching.

29. Tong, H., Faloutsos, C., Gallagher, B., Eliassi-Rad, T. Proc KDD. ACM; 2007. Fast best-effort pattern matching in large attributed graphs; p. 737-746.

30. von Landesberger T, Kuijper A, Schreck T, Kohlhammer J, van Wijk J, Fekete JD, Fellner D. Visual analysis of large graphs: State-of-the-art and future research challenges. Computer Graphics Forum. 2011; 30(6):1719–1749.

31. Wong, PC., Haglin, D., Gillen, D., Chavarria, D., Castellana, V., Joslyn, C., Chappell, A., Zhang, S. Proc LDAV. IEEE; 2015. A visual analytics paradigm enabling trillion-edge graph exploration.

32. Zloof MM. Query-by-example: A data base language. IBM Systems Journal. 1977; 16(4):324–343.



```
MATCH (film1:film)--(film2:film)--(film3:film),
  (film1)--(film3)
WHERE film1.genre = 'Action & Adventure' AND
  film2.genre = 'Action & Adventure' AND
  film3.genre = 'Action & Adventure' AND
  film1.year >= 1980 AND film1.year < 1990 AND
  film2.year >= 1980 AND film2.year < 1990 AND
  film3.year >= 1980 AND film3.year < 1990
RETURN film1, film2, film3
```

Figure 1.

Top: a VISAGE query seeking three similar action films from the 1980's along with a result, found from the Rotten-Tomatoes movie-similarity graph (an edge connects two movies if they are similar). *Bottom*: the equivalent query written in the Cypher querying language. VISAGE's interactive graph querying approach significantly simplifies the query writing process.

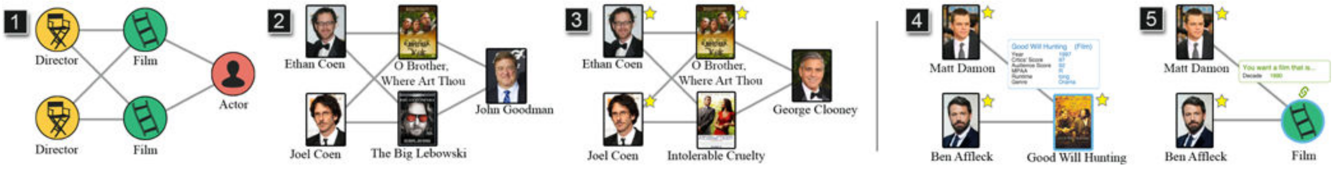
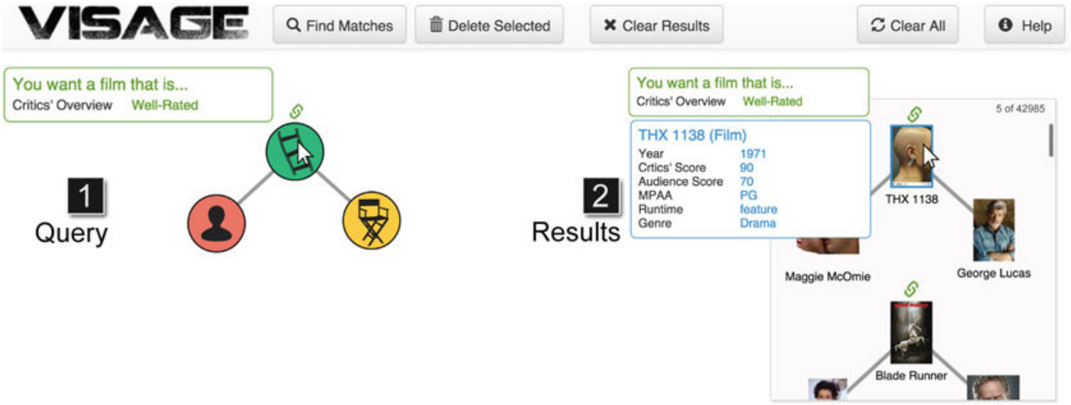


Figure 2.

VISAGE supports many query refinement approaches like abstract querying (1–3) and example-driven querying (4–5). A broad query (1) with only node types and structure, with the first resulting match in (2). The Coen Brothers and the film *O Brother, Where Art Thou?* are starred, fixing these nodes. With the nodes starred, only matches with those nodes are displayed like (3). Bottom-up querying or query by example starts with an example of a known pattern. The known pattern (4) coveys lots of detailed information but is too specific to offer any other matches. In (5), *Good Will Hunting* is abstracted to form a new query based off the example (for only films from the 90s).



You want a film that is...
Critics' Overview Well-Rated

You want a film that is...
Critics' Overview Well-Rated

THX 1138 (Film)
Year 1971
Crtics' Score 90
Audience Score 70
MPAA PG
Runtime feature
Genre Drama

Figure 3.

A screenshot of VISAGE showing an example graph query of films and actors related to George Lucas' films. VISAGE consists of: (1) a *query construction area*, where users construct graph queries by placing nodes and edges; (2) an *overview popup window* that summarizes the desired features (constraints or conditions) of a query node (in green), and the features of a selected node in a match (e.g., the film *THX 1138* in blue); a *results pane*, which shows a list of the results returned by the query. In this example, a user has specified a condition that the film must have a critics' overview of "Well-rated". The matches' layouts (general shape) mirror that of the original query.

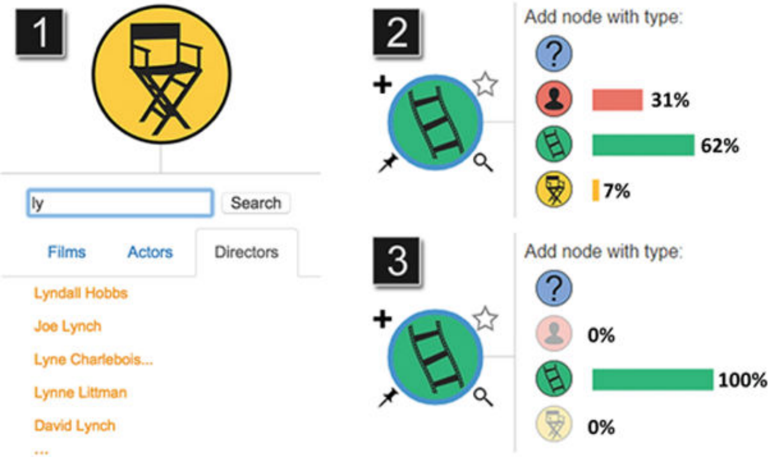


Figure 4.

(1) VISAGE supports conventional text search for finding a node to star. (2) Node controls and the add-node menu; the pin button fixes the node's position in the visualization; the star button (available only when results exist) allows users to keep that particular node in future results; the magnifying glass opens the node-search menu (at 1) that allows users to search for particular nodes. (3) The distribution of each potential neighbor node type is plotted to the right of each node-button; neighbors that will lead to over-specification are grayed out.

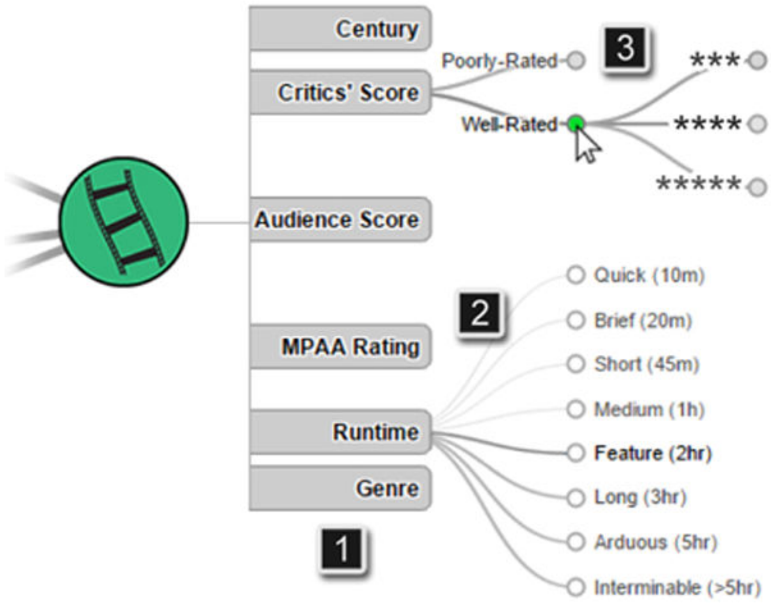


Figure 5.

To investigate the feature space, VISAGE visualizes node features with a tree view. Hierarchical node features can be clicked to hide or show levels of the hierarchy (3). The edges denote the density of the target feature in the current results. The darker edges in (2) mean more results have that attribute value. When a user adds a condition by clicking a node it is highlighted in green, as in (3). If the current node is a result or a starred node, that nodes attributes will be highlighted in blue.

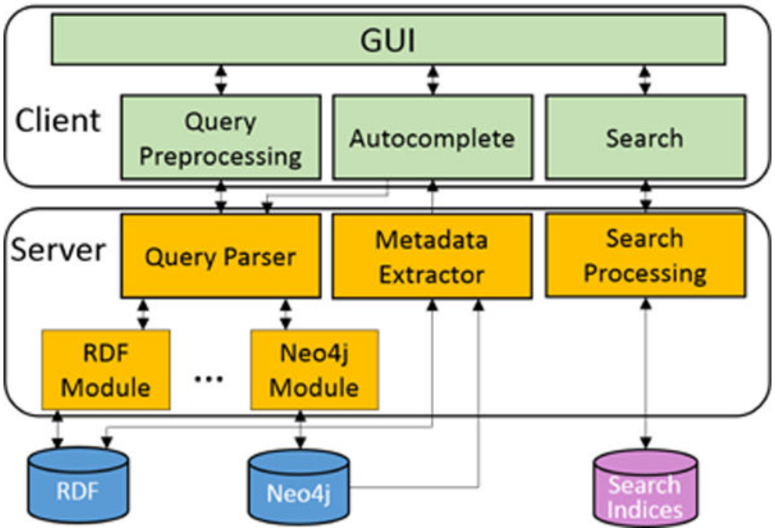
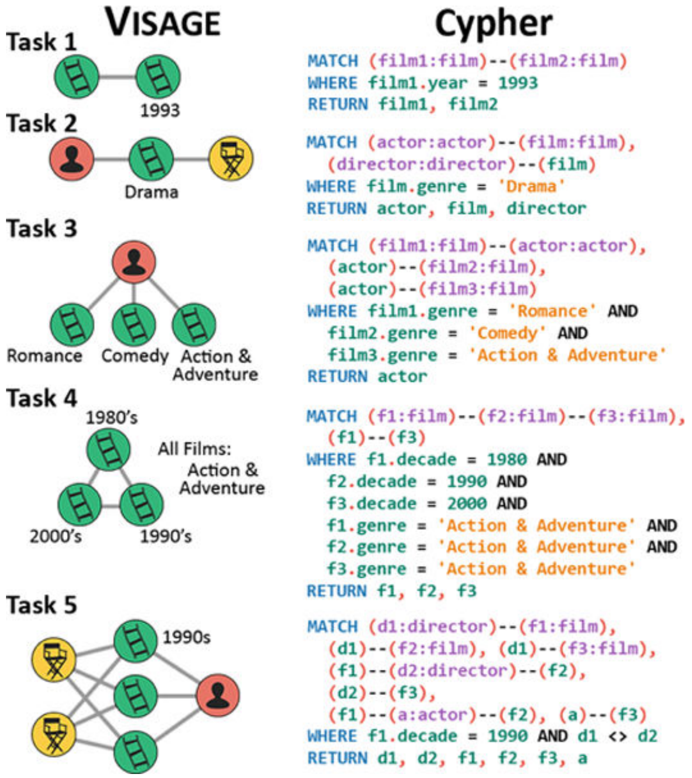


Figure 6.

VISAGE uses a client-server architecture. The client visualizes the query and results. The server wraps a graph database, e.g., Neo4j, RDF database; additional databases can be added via new parser output modules. The *metadata extractor* creates summarization statistics for autocomplete. VISAGE's search functionality for finding specific nodes is sped up using full-text-search indices.



```
MATCH (film1:film)--(film2:film)
WHERE film1.year = 1993
RETURN film1, film2

MATCH (actor:actor)--(film:film),
  (director:director)--(film)
WHERE film.genre = 'Drama'
RETURN actor, film, director

MATCH (film1:film)--(actor:actor),
  (actor)--(film2:film),
  (actor)--(film3:film)
WHERE film1.genre = 'Romance' AND
  film2.genre = 'Comedy' AND
  film3.genre = 'Action & Adventure'
RETURN actor

MATCH (f1:film)--(f2:film)--(f3:film),
  (f1)--(f3)
WHERE f1.decade = 1980 AND
  f2.decade = 1990 AND
  f3.decade = 2000 AND
  f1.genre = 'Action & Adventure' AND
  f2.genre = 'Action & Adventure' AND
  f3.genre = 'Action & Adventure'
RETURN f1, f2, f3

MATCH (d1:director)--(f1:film),
  (d1)--(f2:film), (d1)--(f3:film),
  (f1)--(d2:director)--(f2),
  (d2)--(f3),
  (f1)--(a:actor)--(f2), (a)--(f3)
WHERE f1.decade = 1990 AND d1 <> d2
RETURN d1, d2, f1, f2, f3, a
```

Figure 7.

VISAGE user study tasks. VISAGE queries shown on the left with their corresponding multi-line Cypher queries on the right.

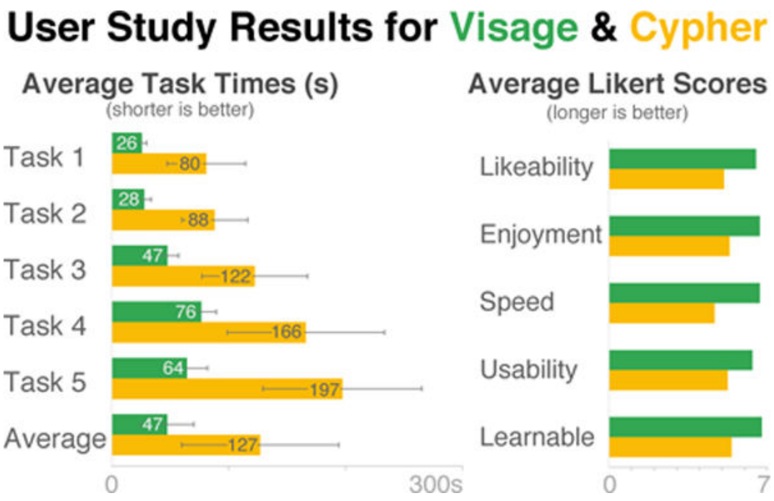


Figure 8.

Average task completion times and likert scores for VISAGE (green) and Cypher (yellow). VISAGE is statistically significantly faster across all tasks. The error bars represent one standard deviation.