Interactive Searching and Visualization of Patterns in Attributed Graphs.

Pierre-Yves Koenig, Faraz Zaidi, D. Archambault

To cite this version:


HAL Id: hal-00499430
https://hal.archives-ouvertes.fr/hal-00499430
Submitted on 9 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
ABSTRACT

Searching for patterns in graphs and visualizing the search results is an active area of research with numerous applications. With the continual growth of database size, querying these databases often results in multiple solutions. Text-based systems present search results as a list, and going over all solutions can be tedious. In this paper, we present an interactive visualization system that helps users find patterns in graphs and visualizes the search results. The user draws a source pattern and labels it with attributes. Based on these attributes and connectivity constraints, simplified subgraphs are generated, containing all the possible solutions. The system is quite generic and capable of searching patterns and approximate solutions in a variety of data sets.

Index Terms: H.5.0 [Information Systems]: Information Interfaces and Presentation—General G.2.2 [Mathematics of Computing]: Discrete Mathematics—Graph Algorithms

1 INTRODUCTION

The problem of searching information in databases has been addressed by many researchers. Representing these search results in a compact form is an important area of research. In case where the query is not precise and an approximate solution is required, the number of solutions returned can be large. Using traditional List-based Result Visualization systems can hinder the efficiency of choosing an appropriate solution. As an alternative to the list-based systems, another way to represent solutions is to use a graphical representation of the search results. As argued by North and North [21], a visualization system can facilitate and empower users to perform more complex information retrieval tasks. Moreover, many real-world systems can be modeled as graphs, for example, Social Networks [31], Metabolic Networks [11], and Transport Networks [23]. A more natural way to search for information in these networks is to draw a pattern where the drawing corresponds directly to the query.

Several approaches exist to search for information in these networks. The most common is through a database query language. As an example, consider an airline network, where the nodes of the graph are airports and edges represent direct flights [23]. This graph can have multiple edges where an edge represents a unique flight on a particular date and time. Attributes are associated with the nodes and edges of the graph and could include information such as airline, flight date, departure time, flight duration, and the cost of the ticket. Attributes associated to nodes of the graph could include country and region. The attribute region represents the subcontinents like eastern and western Europe.

A simple query on this database would be to look for all the flights possible from some city to another, say, Bordeaux to Milano. The user might choose to take indirect flights with not more than one stop over. Generating this query is quite simple and the results can easily be analyzed by a list-based system. On the other hand, if the user is a tourist, who wants to plan a trip to Europe and visit its important cities, the user might ask a question such as: Starting from Bordeaux, find me a tour of four European cities such that the trip lasts two weeks, a stop in Milano to visit a relative on a particular date, and the return flight brings me back to Bordeaux. The table of possible search results can be very large and writing such a query can be quite cumbersome. Figure 1 represents such a graphical pattern where nodes represent airports, and the label Europe is associated to each airport as an attribute of the node.

The problem can also be modeled as a subgraph isomorphism [4] problem. The goal is to search for a graph $G_2$ in a larger graph $G_1$ such that there exists a bijection between the vertex set of $G_1$ and $G_2$. Although subgraph isomorphism is known to be NP-Complete [5], algorithms exist to solve it in a reasonable amount of time on real-world graphs with attributes [4]. The strict definition of subgraph isomorphism requires that all edges be present in the mapping of vertex set of $G_1$ to vertex set of $G_2$. Some researchers, however, relax this constraint. The goal of this inexact matching problem [25] becomes to find a graph that more or less matches the input graph. Moreover, due to the widespread use of graphs as models for representing data, the inexact graph matching problem has attracted lots of research activity [4].

Another approach to solve this problem is to model it as a constraint satisfaction problem (CSP) [24, 15]. Given a set of variables, a finite domain for each variable, and a set of constraints, CSP problems aim to find an assignment of values to the variables from the domain such that the assignment satisfies all the constraints. The graph matching problems has an analogy to CSP problems. The set of variables in CSPs can be considered as the nodes and edges in the graph along with their attributes. The constraints can be translated as restrictions on these nodes and edges. So, for example, a node of $G_1$ can be represented as a variable, the nodes in $G_2$ as the domain, and the connectivity between nodes in $G_1$ as restrictions applied to the variables.

Regardless of the search method used to solve the search problem, our focus is the formulation of the query and representation of the solutions. Instead of finding and enumerating all approximate isomorphisms, which can be costly, we present a visualization of...
simplified areas of the large graph that contains all of these inexact matches. The algorithm takes as input a graph pattern drawn manually by the user containing optional and required edges. The only restrictions we impose on this pattern is that the nodes must be a single, connected component after all optional edges are removed. Our system presents all solutions in a series of subgraph visualizations where the user can easily browse the different possible solutions to pick the most suitable.

Our primary contribution is an interactive system that can be used to help solve this inexact graph matching problem. Secondly, our system maps attributes using spatial position, color, and shape to reduce visual complexity and facilitate the reading of patterns in the graph. Finally, our interactive system does not enumerate over all patterns found in the graph or attempt to find a globally minimal solution. Instead, it presents the set of solutions in an interactive system that can be browsed and modified manually, thus reducing the overall execution time of the search process. Note that a database query might execute faster to return the search results but browsing hundreds of records can be difficult. Currently, the system is implemented as a set of searching algorithms on a graph, but the search process could be replaced by a database query to decrease query time. This approach remains out of scope of this paper as generating database queries from visual interfaces has been addressed by other researchers such as Madurapperuma et al. [20].

The next section contains the related work. In section 3, we formalize the problem and explain the different types of attributes and constraints. Section 4 presents the proposed system. Finally, we present different case studies in section 5 to show the effectiveness of the proposed system.

2 Previous and Related Work

Related work can be classified into four areas. Pattern recognition places an emphasis on subgraph topology while CSP problems place an emphasis on graph attributes. Research has also been undertaken in the visual analytics and the graph drawing community to help illustrate patterns in the context of graphs. Finally, some work has also been done on visual query interfaces for databases.

2.1 Pattern Recognition

Pattern recognition algorithms for inexact matching find solutions that are global minima, but they suffer from high computational requirements [4]. Most of the algorithms are designed to calculate the matching cost on an explicit model of errors. These errors could be, for example, missing nodes or edges in the suggested solution. These models do not incorporate optional edges. Thus, allowing an indirect flight as an optional path to connect to a destination node even though a direct flight exists, would be difficult to input in this type of system.

Another approach to defining a matching cost is to use a set of graph edit operations, such as node insertion, and assign costs to each operation. The goal is to find the cheapest sequence of operations to match an input pattern [8]. These methods require a cost function for optional edges, which may vary between problems and users. For example, a user may prefer direct flights. Thus, high costs would be associated with indirect flights. On the other hand, a user may want to spend less money, even if the stop over of a connection is very long.

The objective of our system is the same as a pattern recognition algorithm with the exception that optional edges are allowed. Additionally, we do not attempt to find a single optimal solution, but several solutions which can be browsed by a user through a visualization system. As we do not attempt to compute this optimal solution, we gain in terms of time complexity over these types of approaches. However, the essence of our approach shares many commonalities with approaches in this research area. Our system uses attributes and connectivity constraints to reduce the set of possible solutions. Since we have several attributes that can be defined on the graph, either locally on particular nodes/edges or on the position of the node in the target graph, we use them to eliminate nodes or edges from the set of possible solutions.

2.2 Constraint Satisfaction Problems

Another way to solve the given problem is through CSP approaches. The classical method for solving CSP problems is backtracking [14]. This class of methods performs a depth-first search of the space of potential CSP solutions [26]. Despite the wide use of these methods, the time complexity for most nontrivial problems is exponential. Moreover, if CSP approaches are to be adapted to search for patterns in graphs, the node connectivity needs to be modeled as a constraint on pairs of nodes, further increasing the complexity of the backtracking method. Techniques such as arc consistency [18] and k consistency [7] can be used to reduce the number of solutions that these backtracking methods explore. However, the methods still remain computationally expensive.

These methods can be used to produce an optimal solution, subject to the constraints imposed by the user. However, mapping constraints to the edges that connect pairs of nodes in a graph is cumbersome, because each constraint must be declared separately for each attribute. As we do not attempt to find a globally minimal solution, our approach offers reduced computational complexity. Instead, the user can browse the solution space to select one or more inexact matches that conform to their needs.

2.3 Graph Pattern Visualization and Interactive Graph Mining

Research in the domain of interactive and visual graph mining is attracting lots of interest as data sets are rapidly growing in size. Systems like [22, 29, 27] help users to visualize large size graphs, organize these graphs, and interact with them for mining purposes. These systems are not designed to address the inexact graph matching problem, and, thus, would require some customization to present solutions to these problems.

A number of contests have focused on the problem of finding graph patterns in a larger graph. The 2003 Graph Drawing Contest [3] focused on how to display a smaller subgraph pattern, or graph motif, in the context of a larger graph. Two systems, both of which are described below, claimed a prize in the contest.

Holleis et al. [10] creates what they call a summary graph, similar to a quotient graph in the graph drawing and visualization literature, that presents the patterns found and their interconnections. Their system places each pattern found into its own node of the summary graph. Two nodes of the summary graph are connected if the patterns are connected. Each found pattern and the summary graph is drawn independently, presenting an overview of the motifs and how they are interrelated.

Klukas et al. [12] describe an interactive system where the user draws a small graph pattern on a panel to the right of the screen. The pattern is found in the larger target graph and emphasized in context using spatial position and color. The emphasis via spatial position is accomplished through a modified force-directed algorithm.

In this contest, the example graphs were quite small, consisting of a few hundred nodes and edges. Additionally, the nodes and edges had relatively few attributes, consisting of one or two pieces of information at most. Our system differs from the solutions presented above, as we consider larger graphs with more attributes. Thus, showing context is more complicated without greater levels of simplification to the large graph. Additionally, we focus on approximate matching of edges attributing them as optional or required to handle the inexact graph matching problem.

The Social Network and Geospatial Challenge of the 2009 VAST Challenge [9] focused on finding a small pattern representing the
communication between actors in an espionage organization in a larger graph. The fictitious scenario involved confirming a pattern of communication between the actors in the organization. These actors communicated using a social networking tool named Flitter. Many solutions to this task were submitted, including ours [28], but most of these systems were tailored directly to the problem posed by the challenge committee and were not general systems. Also, attributes beyond topological information, such as geospatial constraints, were not really considered when finding the pattern. It was only afterward, when multiple solutions were found, that these constraints were applied by the participants as a post-process to filter the solution set.

2.4 Interactive Visual Query Generation

Research in the domain of generating queries for databases through visual interfaces has interested many researchers. The most important reason being that visual interfaces seem to be a more natural way for humans to query databases as compared to using some query language. The readers are recommended to refer to several papers on the topic [16, 17, 19, 20] for further information. However, these papers generally do not focus on searching for approximate patterns in graphs.

3 Problem Formalization

The input to the visualization system consists of two graphs. The source graph, $G_s = (N_s, E_s)$, is the graph pattern sought by the user. The target graph, $G_t = (N_t, E_t)$, is the graph in which the source graph will be found. Both the source and target graphs can be modeled as $G = (N, E, A)$ consisting of:

- a finite set of nodes $N$.
- a finite set of edges $E \subseteq N \times N$ with loops and multiple edges.
- a finite set of attributes $A$ associated with $N$ and $E$, having a finite set of domain.

The source graph has an additional parameter $T$:

- type $T$, for each edge, which can take boolean values representing if the object is required or optional.

Required elements of the source graph are nodes and edges that must appear in the pattern. Optional edges of the source graph may be present in a solution, but do not eliminate the solution as a candidate. Any edge that is not present in $G_t$ may be present in $G_s$, but it is filtered out by the system when presenting solutions. In our system, the sets $T$ and $A$ apply locally on the source graph. Each attribute value represents a unary constraint on either a node or an edge. As an example, in Figure 1, a node of the pattern must have the value Bordeaux (departure and arrival node). In our approach, we constrain our source graphs to consist of a single, connected component using only the elements of $T$ labeled Required.

4 Proposed System

The proposed system comprises of four steps as shown in Figure 2. Each of these steps is explained in detail in the following sections.

4.1 Interactive Query Generator

The interactive query generator serves three main purposes. The first purpose is to enable the user to generate queries through a visual interface. The idea is to let the user draw a pattern of nodes and edges annotated with attributes. The second purpose is to allow the user to specify a visual encoding of attributes on nodes (like geographical position mapped to color) and the desired layout (position of nodes on the screen to create a mental map). This visual encoding is used to display various search results that can be further explored by the user. The third objective is the automatic extraction of node-connectivity constraints to determine how the nodes must be connected to each other defining the pattern to be searched. These constraints eventually help to reduce the number of candidate solutions of a given pattern.

The interactive query generator is a graph drawing tool, actually the Tulip software interface [1], allowing the user to draw a pattern. The user can further assign desired attribute values to the nodes and the edges. The interface of the query generator is shown in Figure 3. The visual encoding includes shape, position, and color and is used when presenting different possible solutions to the user.

The next step is to extract constraints from the graph. We define two types of constraints on $N$ and $E$ of graph $G_t$: connectivity constraints and attribute constraints. Connectivity constraints are imposed by the edges of source graph where they define how the nodes of this source graph must be connected to each other. For example, looking for an edge between two nodes is an example of a connectivity constraint. In the air traffic network, this edge might be a constraint that a direct flight must exist between two cities for the solution to be considered. Attribute constraints are the conditions defined on either edges or on nodes of $G_t$. An attribute is required to have a certain value or be within a range of values. Continuing with our airport example, possible attributes for nodes are the city names, chosen as departure and arrival cities or ticket fares associated to edges representing particular flights. Connectivity and attribute constraints play an important role in reducing the number of possible solutions. The next section presents how our algorithm uses this information to find candidate solutions.

4.2 Constructing and Filtering Candidate Sets

Our solution involves creating a set of possible candidates for each node of the source graph, and then iteratively removing elements from these sets based on connectivity and attribute constraints.

![Figure 2: Block diagram of the proposed system. The interactive query allows the user of the system to draw the source graph. Candidate set construction finds potential candidates in the target graph. Candidate layout draws the collection of solutions in a way that accentuates structures similar to the drawn source graphs and maps visual attributes. Finally, the solutions are presented to the user and they can be edited manually using the Tulip interface.](image-url)
4.2.1 Constructing Node Candidate Sets

In the first step of the algorithm, for each node in $G_1$, we construct a set of candidate nodes from $G_1$ that fulfill all the attribute constraints of that particular node in $G_1$. For example, as shown in Figure 1, the node labeled Bordeaux represents an attribute constraint where the nodes label should be Bordeaux. While constructing the candidate set for this node, only the nodes having this label will be retained. In this case, the airport labeled Bordeaux will be the only node in the first candidate set. Similarly, the candidate set for the node labeled Milano will contain only one node. The candidate sets for nodes labeled 1,2,3 will contain airports from the region Europe as given in the example, and this region constraint is the only attribute specified by the user for the nodes. Initializing the candidate sets requires $O(|A||N_s||N_s|)$ time in the worst case, as we need to check to see if every element of $N_s$ belongs to one of the candidate sets of $N_s$ and $A$ is the number of attributes associated to node and/or edges. As $|A|$ and $|N_s|$ are usually small, this step of the algorithm is fairly efficient.

4.2.2 Finding Patterns from $S$

The next step is to iteratively eliminate nodes from these candidate sets based on connectivity constraints. First we find a minimum spanning tree using Prim’s algorithm from $G_1$, considering only the nodes present in candidate sets of required elements, which we call $RT$. In the European tour example, $RT$ would be a pass of the cycle.

Continuing with the European tour example, the nodes and edges are required in this example. But it may be the case that the user might need to cut their trip short after Milano due to financial constraints. In this case, they might also be interested in return flights from Milano directly to Bordeaux. The edge representing this flight is an optional edge as the pattern searched does not necessarily require this edge, but the edge brings additional information to the user which might be useful. This optional edge would never be present in $RT$.

Next, we select the candidate set with the smallest cardinality and call this set $S$, which in our current example, can be either Bordeaux or Milano as the candidate sets of both these nodes contain exactly one element. For each node in the smallest candidate set, we construct a subgraph as described below.

Starting from the root of $RT$, we perform a breadth-first search in $G_1$ to find all possible instances of $RT$. From each element of $S$, we look at the adjacent neighboring vertices in $G_1$. All nodes that are part of a candidate set directly adjacent to a node in $S$ are added to the subgraph associated with the node in $S$. We repeat the process for all the other nodes of $RT$. In the current example, if we start with the candidate set for the node Bordeaux, we look at the nodes adjacent to Bordeaux in the candidate set of node 1 (see Figure 7). The nodes that are not directly connected to Bordeaux, are removed from the candidate set of node 1. We continue for all nodes in the order defined by $RT$.

While considering the connectivity of nodes, we filter out edges that violate any attribute specified by the user on the edges. For example, if the user does not want to take a flight that costs more than some specified amount of money, the edge will not be considered. Returning to the example, when we look for nodes directly connected to Bordeaux in the candidate set of node 1, we only consider the edges that conform to all the attributes associated with the edges of $G_2$.

As a result of this step, we obtain a subgraph of $G_1$ as shown in Figure 8. In this subgraph, for each node in $G_1$, we have identified the possible candidate nodes from $G_1$. A number of solutions can be extracted from this subgraph as all the possible routes are presented to the user. Recall that if the cardinality of set $S$ is more than 1, we will obtain several subgraphs, each representing a set of possible solutions. The number of subgraphs generated for the three different data sets are mentioned in Figure 13 where each subgraph contains several solutions. The breadth-first search of $G_1$ is the most expensive step as every node of $|N_s|$ can be a part of $|N_s|$ candidate sets. Thus, this step can take $O(|S|(|N_s||N_s| + |A||E_s|))$ time.

4.2.3 Filtering Solutions and Node Splitting

Once we obtain subgraphs containing sets of possible solutions, we filter by local connectivity and attribute constraints. For example, each node in Figure 7 must have at least degree two. Not only do the candidate nodes need this degree, but they also need to be connected to the right two candidate sets of $G_2$. The algorithm begins by computing, for each node in $G_2$, the required connections for all candidate sets. Next, for each node in each subgraph, the algorithm checks that it is adjacent to all sets required by $G_2$. If it is not adjacent, then the node is removed from the candidate set for this subgraph. The process is repeated until no more nodes can be removed.

Let $\mathcal{G}_{\text{max}} = (N_{\text{max}},E_{\text{max}})$ be the largest subgraph in $S$. As it is possible to remove a constant number of nodes from the candidate sets in each iteration on a particular subgraph, the loop can execute at most $O(|N_s||N_{\text{max}}|)$ times. Thus, in the worst case, this step can take $O(|S|(|N_s|^2N_{\text{max}}^2+|S||N_s|^2N_{\text{max}}^2||A||E_{\text{max}}|))$ time. It is possible that $|N_s| = |N_{\text{max}}|$, but this would imply that nearly all nodes in $G_2$ are candidates in a subgraph of $S$, or that hardly any filtering occurred. Frequently, we have observed that $|N_s| >> |N_{\text{max}}|$. Also, $|S|$ and $|N_s|$ are generally small.

It is possible to have a single node belonging to candidate sets of more than one node. An example is the city of Paris and Dublin in Figure 8 as both of them are connected to Bordeaux and are present in candidate sets 1 and 2. To handle the nodes that belong to more than one candidate set, we split the nodes such that we make multiple copies of the node. As a result of this step, each node belongs to a single candidate set. Splitting allows us to place nodes of the graph into multiple candidate sets. These candidate sets will be represented as metanodes, during candidate layout.

4.3 Candidate Layout

As mentioned previously, we do not iterate over all solutions to the subgraph isomorphism problem. Rather, we present a visualization.
where many solutions that originate from a single element of $S$ can be visualized simultaneously. In order to facilitate visualization, we map the candidate sets to a layout that resembles the user generated input layout of $G_s$. As an example, in Figure 4, all nodes of graph in Figure 4(a) are mapped to the layout in Figure 4(b) giving a final layout presented in Figure 4(c).

For each candidate set, we create a metanode. The diameter of this metanode is proportional to the number of elements present in its candidate set. We map the position of each metanode in this metagraph to the position of its corresponding node in $G_s$ that was determined by the user. The layout of this metagraph is uniformly scaled by a factor equal to the largest metanode diameter radius. Subsequently, we apply the algorithm of Dwyer et al [6] to ensure that no two metanodes overlap.

The algorithm places each node of the candidate set on the diameter of the metanode created in the previous step. Candidate edges span the gaps between metanodes. As we are free to choose the orientation of this diameter, we choose a line $\ell$, as shown in Figure 5 and defined by the following equation:

$$\ell = p + t v_{ave}$$

where $p$ is the position of the metanode and $t$ is a scalar value. The vector $v_{ave}$ is the vector perpendicular to $v_{ave}$ where vector $v_{ave}$ is the average of the vectors between adjacent metanodes and the current one.

It can happen that $|v_{ave}|$ is zero. Figure 6 shows a typical case: $v_1$ is directly opposite to $v_2$. We resolve this problem by simultaneously computing $v_{ave}$: the average vector, but with a 180 degree rotation for any vector with a negative x-component. Let us denote this series of vectors $v_1', v_2', \ldots, v_n'$. Therefore, we are guaranteed that $|v_{ave}|$ is non-zero as all x-components are positive. Also, the orientation of $v_1'$ is equal as good as $v_1$ in this case, because it does not matter from which side of the row edges attach themselves. If $v_{ave}'$ is, on average, more orthogonal to the vectors $v_1 \ldots v_n$, it is chosen instead of $v_{ave}$ for $\ell$ which occurs when:

$$\sum_{i=1}^{n} |v_i \cdot v_{ave}'| < \sum_{i=1}^{n} |v_i \cdot v_{ave}|$$

In this equation, all vectors of $v_1 \ldots v_n$ along with the vectors $v_{ave}$ and $v_{ave}'$ have been normalized.

When drawing the source graph, the user of the system can specify the color and shape of the nodes. Once the layout has been determined, polygons are added to the background of the layout for each candidate set. The shape and color of these polygons match those chosen by the user as seen in Figure 4. The layout algorithm described above specifies the size and positions of these nodes.

5 CASE STUDIES

We present three case studies on different data sets. The layout algorithm, presented in section 4.3, places candidate nodes along diameters of circles positioned as close as possible to the input layout specified by the user. Also, the shape of the nodes is changed so that it has the same color and shape of the node in $G_s$, allowing candidate sets to be identified. Only the edges that satisfy all edge constraints are drawn, and these edges span the space between metanodes. Figure 13 presents information about the data used and the execution of our algorithm. All of the case studies were executed on a 2.16GHz dual core Pentium IV with 2.0GB of memory, running Fedora Core 8 with a 2.6.26.8-57 kernel.

5.1 Airline

The air traffic network is a directed and highly multi-edge graph consisting of individual flights between cities in Europe. As there exists one edge per flight between two cities, many multiple edges can exist between two nodes. We have many attributes on the nodes and edges of this graph, including the departure and arrival time of the flight and the servicing airline company.

On this data set, we explore a number of possible tours around Europe. For example, tourists who want to visit important European cities in a limited time. Thus, they may be looking for a loop that connects several touristic cities, subject to some constraints. Maybe they need to visit a relative in Milano on a particular day. They may like to take a certain airline to maximize frequent flyer points. Additionally, they have constraints on departure and arrival times from their base city.

Our source graph, in this example, is a directed cycle of five nodes as shown in Figure 7. The data set does not have any dates so we assume that all flights are available on all days. The trip begins at Bordeaux where the first flight must take off after 9:35 in the
morning. The destination of this flight, the green node at the top of the diagram, is unconstrained. On the next day, from the green node, we fly to Milano, the yellow hexagon in the diagram. We visit our relative for two days, and our third flight leaves after 21:00 to an unconstrained destination. Our fourth flight is to an unconstrained destination in Europe. Finally, the last flight is constrained to land at Bordeaux before 20:31, so that we are able to get back home before midnight. The first and last flight of our trip is constrained to be with a particular carrier, to earn some frequent flyer miles.

In Figure 8, we show the result of Airline. As we constrain the first node of the cycle to leave from Bordeaux, we only have a single candidate for this node. This situation is exactly the same for Milano. The first leg of our trip can take us to a variety of cities, as shown in the upper, green node, including Porto, Dublin, and Barcelona. From each of these cities, we have a variety of flights to Milano. Destinations for the third city on our tour include Lisboa, Firenze, and Amsterdam. Then we have our fourth city followed by a flight home. These unconstrained airports contain ten and fifteen cities respectively. In many cases, the graph is constrained enough so that we can follow the individual cycles. Although edge occlusion does occur between the fourth and fifth cities of our trip, most of the cycles are relatively easily read through visual inspection of the diagram.

In the 2009 VAST Challenge [9], contestants were asked to find a pattern in a large graph, subject to a variety of constraints. The nodes of this graph are users of Flitter, a fictitious social networking program, and edges exist between nodes if they communicate. The graph is simple and undirected. Once the contest was over, the ground truth was known, but the initial task was to find this solution based on the indices provided by the contest organizers.

In the given scenario, an employee leaked sensitive information from an embassy. In order to protect the identity of the employee’s fearless leader, the information was redirected twice. The first level of indirection is through three handlers while the second level is through one or three middlemen. Figure 9 shows the pattern of communication with three middlemen. The light brown box in the diagram is the employee. The employee communicates with three handlers which are the green circles. Each handler has a middleman, the light blue triangles. Finally, all three middleman connect to a fearless leader, the purple pentagon. Employees are constrained to have a degree between thirty-five and forty-five in $G_t$. Handlers have a degree between twenty-seven and forty-three. The middleman has a degree of between zero and seven. Finally, the fearless leader has a degree greater one hundred. Geospatial constraints dictate that the fearless leader has to reside in the city Koul and the employee in Prouvnov.

The system returns nine subgraphs in total. Four of these subgraphs contain valid solutions and the remaining five are empty subgraphs because filtering eliminated all solutions. Figure 10 shows the solution many teams found in the contest. The algorithm started
In this paper, we have presented a system to find and understand patterns in graphs. One of the strengths of the system is that it presents solutions to an inexact matching of a pattern where edges can be optional or required. The mapping of attributes to layout and color help reduce the visual complexity of the presented solutions. Finally, we provide an interactive system that allows users to explore sets of matches to a particular pattern and modify them manually. As a result, even when the source and target pairing approaches unconstrained subgraph isomorphism, our system is polynomial, because we do not iterate over all patterns, but present them in a visualization.
In future work, we would like to improve the performance of the filtering algorithms as we believe that the complexity can be reduced significantly. Also, it would be interesting to look at ways of reducing visual clutter of the edges present in our solution as in our constrained layout case, it may be possible to be more efficient. Several algorithms exist to reduce edge crossings, a few directly applicable approaches are discussed in Bauer and Brandes [2], and would remain an important topic for future work. Finally, formal user experimentation and testing, especially in terms of interactivity, would be needed to validate the approach.

REFERENCES


