Interactive Searching and Visualization of Patterns in Attributed Graphs.

Pierre-Yves Koenig, Faraz Zaidi, D. Archambault

Interactive Searching and Visualization of Patterns in Attributed Graphs

Pierre-Yves Koenig
CNRS UMR 5800 LaBRI &
INRIA Bordeaux-Sud Ouest
pierre-yves.koenig@labri.fr

Faraz Zaidi
CNRS UMR 5800 LaBRI &
INRIA Bordeaux-Sud Ouest
faraz.zaidi@labri.fr

Daniel Archambault
INRIA Bordeaux-Sud Ouest
& University College Dublin
daniel.archambault@inria.fr

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.

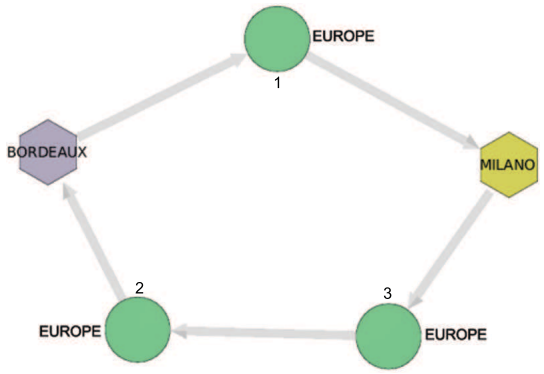

Figure 1: Pattern for European Tour of four cities from Bordeaux and visiting Milano. Nodes are numerically labeled for reference.

The problem can also be modeled as a subgraph isomorphism [4] problem. The goal is to search for a graph G_1 in a larger graph G_2 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 problems(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

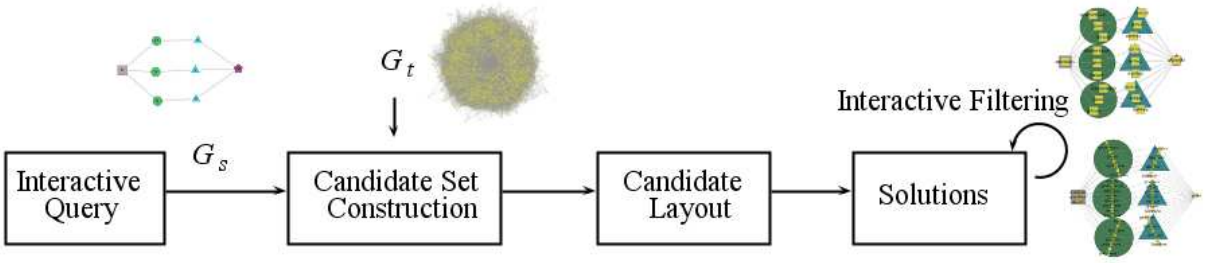

Figure 2: Block diagram of the proposed system. The interactive query allows the user of the system to draw the source graph. Candidate set construction finds potential candidates in the target graph. Candidate layout draws the collection of solutions in a way that accentuates structures similar to the 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.

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_s may be present in G_t, 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_s: *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_s. 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.

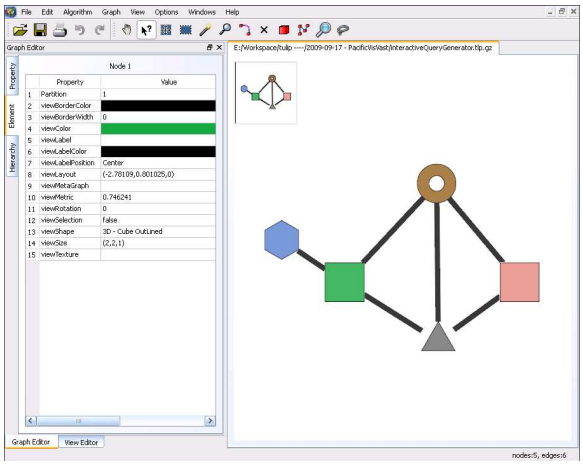

Figure 3: The interactive query generator. The interactive query generator is a graph drawing tool, actually the Tulip software interface [1], that allows the user to draw the pattern. The tool allows the user to specify node shapes, colors, and positions manually. These nodes are the nodes of G_s, and the visual attributes assigned here will be used in the presentation of the solutions as described in section 4.3

4.2.1 Constructing Node Candidate Sets

In the first step of the algorithm, for each node in G_s, we construct a set of candidate nodes from G_t that fulfill all the attribute constraints of that particular node in G_s. 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 in the given example, and this region constraint is the only attribute specified by the user for the nodes. Initializing the candidate sets requires $O(|A||N_s||N_t|)$ time in the worst case, as we need to check to see if every element of N_t 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_s, considering only the nodes and edges attributed as *required*. As G_S is constructed using only required edges, there must exist at least a spanning tree of G_s composed entirely 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_t to find all possible instances of RT. From each element of S, we look at the adjacent neighboring vertices in G_s. 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_s.

As a result of this step, we obtain a subgraph of G_t as shown in Figure 8. In this subgraph, for each node in G_s, we have identified the possible candidate nodes from G_t. 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_t is the most expensive step as every node of $|N_t|$ can be a part of $|N_s|$ candidate sets. Thus, this step can take $O(|S|(|N_s||N_t|+|A||E_t|))$ 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_S. The algorithm begins by computing, for each node in G_s, 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_s. 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 $G_{\max} = (N_{\max}, E_{\max})$ 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 $O(|N_s||N_{\max}|)$ times. Thus, in the worst case, this step can take $O(|S||N_s|^2|N_{\max}|^2 + |S||N_s|^2|N_{\max}||A||E_{\max}|)$ time. It is possible that $|N_t| = |N_{\max}|$, but this would imply that nearly all nodes in G_t are candidates in a subgraph of S, or that hardly any filtering occurred. Frequently, we have observed that $|N_t| >> |N_{\max}|$. 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

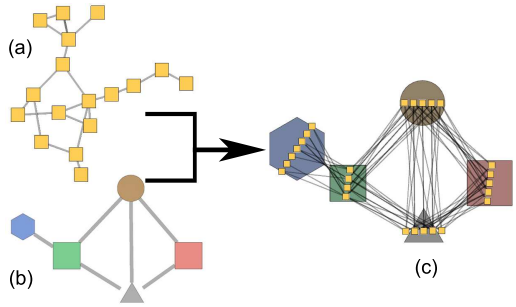

Figure 4: Mapping the graph to the user-created layout and encoding of G_s to the candidates found by the system. **(a)** An example graph where nodes have been associated to several candidate sets. **(b)** The candidate graph drawn by the user. **(c)** Candidate graph mapped to the layout of G_s. The shape and color of the nodes of G_s are mapped to polygons situated behind the sets of nodes for easy identification

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 ℓ, as shown in Figure 5 and defined by the following equation:

$$\ell = p + tv_{ave}^{\perp} \tag{1}$$

where p is the position of the metanode and t is a scalar value. The vector $v_{ave}^{\perp}$ 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 $v'_1 \ldots v'_n$. Therefore, we are guaranteed that $|v'_{ave}|$ is non-zero as all x-components are positive. Also, the orientation of v'_1 is equally 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 ℓ which occurs when:

$$\sum_{i=1}^{n} |v_i \cdot v'_{ave}| < \sum_{i=1}^{n} |v_i \cdot v_{ave}| \tag{2}$$

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

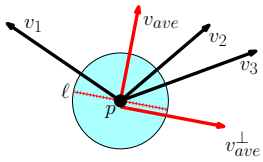

Figure 5: Diagram showing how the diameter, where the nodes of a particular candidate set are placed, is chosen. All vectors to adjacent nodes are averaged to produce an average vector v_{ave}. In this case, those vectors are v_1, v_2, and v_3. The perpendicular to this vector is $v_{ave}^{\perp}$. The point p, the position of the metanode, and $v_{ave}^{\perp}$ are used to define the line ℓ which is the diameter on which the nodes are placed. The line ℓ is to be as perpendicular as possible to all incident edges from other metanodes in the drawing.

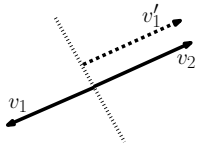

Figure 6: Incidents where $|v_{ave}|$ can be zero. In this case, many of the unit vectors are directly opposing vectors. We compute the mirror image of all vectors with a negative x-component as we do here for v_1. The average of these vectors is non-zero since all x-components are positive. Also, the orientation is equally as good because it does not matter if edges from adjacent metanodes attach to nodes on the diameter from the left or right side of the line ℓ.

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 on some flights 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

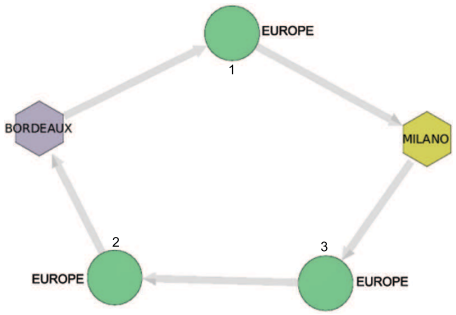

Figure 7: G_s for `Airline` where the user is looking for touristic cities in Europe and wants to pass through *Milano* to visit a relative.

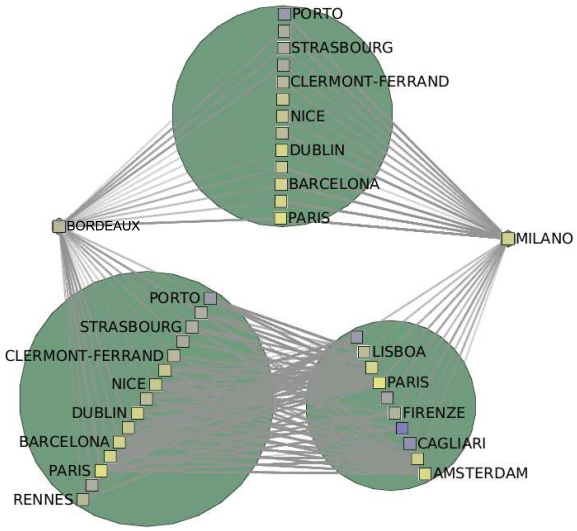

Figure 8: Results of the possible touristic destinations in Europe starting from, and returning to Bordeaux.

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.

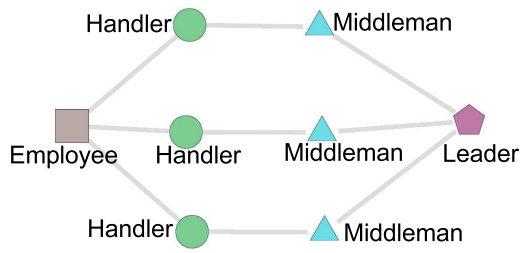

Figure 9: G_s for the Vast Challenge where the objective is find a similar pattern in the data set.

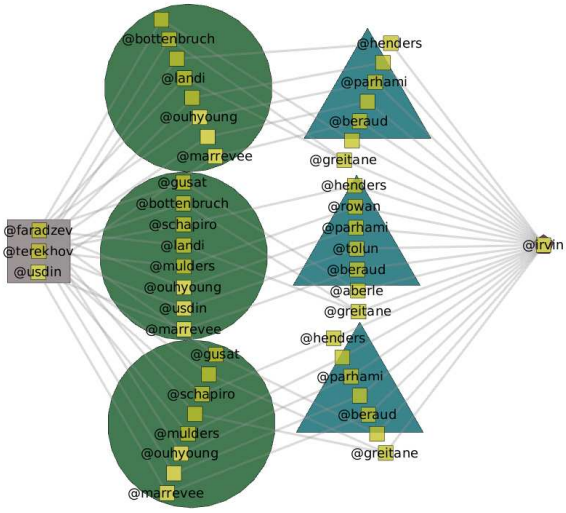

Figure 10: One of the 9 possible subgraphs found with various possible solutions to the pattern searched for the Vast Challenge.

5.2 Vast

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

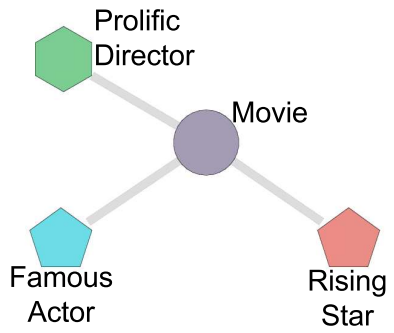

Figure 11: G_s for `Movie` where we want to discover rising stars in cinema.

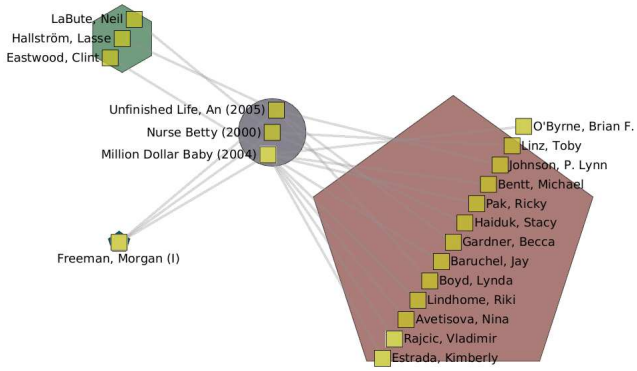

Figure 12: One of the possible 24 subgraphs returned as a result by the search for the Movie pattern where a number of up coming stars can be seen that have performed with famous actors and directors.

its traces into G_t from the fearless leader @irvin. On the other side of the diagram, three possible candidates exist the employee @faradzev, @terekhov, and @usdin. These three candidates could communicate with eight possible handlers in the green circles. As all three handlers have identical constraints, each circle contains the same candidates. Finally, the handlers can communicate with seven different middlemen contained in the light blue triangles. All of these middlemen communicate with @irvin. Using this diagram, we can apply less precise constraints, such as a middleman must be located nearby the city of the fearless leader, to refine the solutions as given in the problem statement of the contest.

5.3 Movie

`Movie` is derived from the 2007 InfoVis Contest data set [13]. The data set consists of a list of movies, containing information such as the actors list, director, title, any Oscars won, and other information. From this data, we constructed an undirected, bipartite graph without multi-edges. Nodes in this graph are actors, directors, or movies. Movies are linked only to actors or directors and vice versa.

In `Movie` database, we test a theory about rising stars. In this theory, we have a famous actor, who has acted in many movies and won at least one Oscar. The actor works with a prolific director on at least one film. In this film, there are many actors who have acted in very few movies, but will get exposure from this work. A famous actor works on at least ten movies and has won at least one Oscar. A prolific director must have directed at least four movies. Movies are completely unconstrained. Rising stars have acted in at least two movies and at most five. In our query graph, Figure 11, the light blue pentagon is the famous actor and the green hexagon is the prolific director. The purple circle is a common, unconstrained movie between the two. Finally, the red pentagon is the rising star.

The query returns twenty-four subgraphs. Two are empty due to constraint filtering. Thus, twenty-two subgraphs contain at least one solution. In Figure 12, we show a solution. Here, *Morgan Freeman* is the famous actor. He worked on many films one of which was *Million Dollar Baby* with *Clint Eastwood* as a director. Many actors and actresses may have gained exposure from working on this movie. This data set suggests that *Brian F. O'Byrne* and *Jay Baruchel* are two such actors. Two other projects with *Morgan Freeman* create other solutions in this data set.

6 Discussion

Our approach seems to be fairly generic and works on a very wide range of data, as seen through our case studies in section 5. The system is able to visualize simple and multi-edge, directed and undirected, as well as attributed and weighted graphs. Also, as seen in Figure 13, the approach executes in less than twenty seconds for our case studies on a standard machine.

The approach proposed in the system seems to have good performance on some simple and interesting patterns. In our examples, the average number of times the constraint application loop was executed, *loop* in Figure 13, was at most 4. This value is quite far from the possible $|N|^2$ iterations that could take place. Additionally, $|S|$ and the average number of nodes and edges in the subgraphs are relatively small as attribute filtering helped significantly. Also, it should be possible to use a database as a back end for our visualization system, which could increase searching speed. However, it is important to note that even if our problem requires a solution to subgraph isomorphism to be solved, we remain polynomial as we do not iterate over all patterns, but present them in a visualization.

Although not enumerating the patterns in the graph has the advantage of time complexity, if many solutions exist, visual clutter can occur. A good example is present in Figure 8 where two unconstrained sets of cities are linked at the bottom of the diagram.

Currently, the system constraints state that there must exist a spanning tree of required edges. This spanning tree cannot allow for the choice of two or more alternative paths to reach a node. Our search algorithm is currently subject to this constraint, but our visualization algorithm would be able to handle results from this case. Further work would be required to address this limitation.

The system in its current state has some obvious limitations. The most important one is its scalability as the number of solutions increase, visualizing all of them at the same time becomes difficult. As shown in Figure 13, the Movie data set contains a large number of nodes and edges. Even though the source graph has only four nodes and three edges, the resultant graph can be large (Figure 12). As mentioned earlier, there exists an inverse relationship between the amount of information input to the system to the amount of information returned as a result. The more constraints we associate with the source graph, the more focused would be the search results and thus would be easier to visualize.

7 Future Work and Conclusion

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 the solutions to the problem. Rather, a visualization is used because humans can recognize patterns efficiently [30].

Result	G_t			G_s			$\|S\|$	$\|N_{\max}\|$	$\|E_{\max}\|$	$\|N_{ave}\|$	$\|E_{ave}\|$	loops	time
	$\|N_t\|$	$\|E_t\|$	$\|A_t\|$	$\|N_s\|$	$\|E_s\|$	$\|A_s\|$							sec
Airline	250	25,953	23	5	5	4	1	41	1,499	41	1,499	2	20.1
VAST	6, 016	29,888	4	8	9	1	9	57	69	13.8	18.0	4.22	0.82
Movie Discovery	166,928	226,523	16	4	3	4	24	72	72	33.0	32.9	3	5.47

Figure 13: The graphs, queries, and execution time of our algorithm during the case studies. **Result** is the name of the case study. The values of $|N_t|$, $|E_t|$, and $|A_t|$ are the number of nodes, edges, and attributes in the target graph respectively. Similarly, the values $|N_s|$, $|E_s|$, and $|A_s|$ are the corresponding values in the source graph. $|N_{\max}|$ and $|E_{\max}|$ are the maximum number of nodes and edges in any subgraph of the result. $|N_{ave}|$ and $|E_{ave}|$ are the average number of nodes and edges in each subgraph produced by the system. The value of **loops** corresponds to the average number of times the first step of filtering, described in section 4.2.3, is executed. **Time** states the running time of the algorithm in seconds from the time the search is launched to computation of the final drawing.

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

[1] D. Auber. Tulip - a huge graph visualization framework. In P. Mutzel and M. Jnger, editors, *Graph Drawing Software*, Mathematics and Visualization Series. Springer Verlag, 2003.

[2] M. Baur and U. Brandes. Multi-circular layout of micro/macro graphs. In S.-H. Hong, T. Nishizeki, and W. Quan, editors, *Graph Drawing*, volume 4875 of *LNCS*, pages 255–267. Springer, 2007.

[3] F. J. Brandenburg, U. Brandes, P. Eades, and J. Marks. Graph drawing contest report. In *Proc. of Graph Drawing (GD '03)*, volume 2912 of *LNCS*, pages 504–508, 2003.

[4] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *Int. J. Patt. Recog. and A..I*, Volume: 18, Issue: 3:265–298, 2004.

[5] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. of the 3rd Annual ACM Symp. on Th. of Comp.*, pages 151–158, 1971.

[6] T. Dwyer, K. Marriott, and P. J. Stuckey. Fast node overlap removal. In *Proc. Graph Drawing (GD'05)*, volume 3843 of *LNCS*, pages 153–164. Springer-Verlag, 2005.

[7] E. C. Freuder. Backtrack-free and backtrack-bounded search. *Search in Artificial Intelligence*, pages 343–369, 1988.

[8] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. volume Volume 13, Number 1, pages 113–129, 2009.

[9] G. Grinstein, C. Plaisant, J. Scholtz, and M. Whiting. The 2009 VAST challenge. In *IEEE Proc. on Visual Analytics Science and Technology*, 2009.

[10] P. Holleis, T. Zimmermann, and D. Gmach. Drawing graphs within graphs: A contribution to the graph drawing contest 2003. *Journal of Graph Algorithms and Applications*, 9(1):7–18, 2005.

[11] H. Jeong, B. Tomber, R. Albert, Z. Oltvai, and A.-L. Barabási. Large-scale organization of metabolic networks. *Nature*, 407:651–654, 2000.

[12] C. Klukas, D. Koschützki, and F. Schreiber. Graph pattern analysis with PatternGravisto. *J. of Graph Algo. and App.*, 9(1):19–29, 2005.

[13] R. Kosara, T. J. Jankun-Kelly, and E. Chlan, editors. *IEEE InfoVis 2007 Contest: InfoVis goes to the movies*, 2007. www.apl.jhu.edu/Misc/Visualization/index.html (visited 04/03/2010).

[14] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *Artificial Intelligence Magazine*, 13(1):32–44, Spring 1992.

[15] J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Math. Struct. in Comp. Sci.*, 12(4):403–422, 2002.

[16] W.-S. Li, K. S. Candan, K. Hirata, and Y. Hara. IFQ: A visual query interface for object-based image retrieval. In *CHI '97: Extended abstracts on Human Factors in Computing Systems*, pages 32–33, New York, NY, USA, 1997. ACM.

[17] Y. Lin, G. Rawlins, and M. Vanheyningen. Pic1: A visual database interface. volume 8, pages 237–245, 1995.

[18] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[19] A. Madurapperuma, W. Gray, and N. Fiddian. A visual query interface for a customisable schema visualisation system. *Database Engineering and Applications Symposium, International*, 0:23, 1997.

[20] A. P. Madurapperuma, W. A. Gray, and N. J. Fiddian. Customisable visual query interface to a heterogeneous database environment: A meta-programming based approach (abstract). In *BNCOD 15: Proceedings of the 15th British National Conferenc on Databases*, pages 129–130, London, UK, 1997. Springer-Verlag.

[21] M. M. North and S. M. North. An information exploration and visualization approach for direct manipulation of databases. In *VCHCI '93: Proceedings of the Vienna Conference on Human Computer Interaction*, pages 417–418, London, UK, 1993. Springer-Verlag.

[22] J. F. Rodrigues, H. Tong, A. J. M. Traina, C. Faloutsos, and J. Leskovec. GMine: A system for scalable, interactive graph visualization and mining. In *VLDB'2006: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1195–1198. VLDB Endowment, 2006.

[23] C. Rozenblat, G. Melançon, and P.-Y. Koenig. Continental integration in multilevel approach of world air transportation (2000-2004). *Networks and Spatial Economics*, 2008.

[24] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *Selected papers from the 6th International Workshop on Th. and App. of Graph Transformations*, pages 238–251, London, UK, 2000. Springer-Verlag.

[25] L. G. Shapiro and R. M. Haralick. Structural descriptions and inexact matching. *IEEE Trans. on Pattern Analysis and Matching Intelligence*, PAMI-3(5):504–519, 1981.

[26] S. C. Shapiro. *Encyclopedia of Artificial Intelligence*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[27] S. J. Simoff, M. H. Böhlen, and A. Mazeika, editors. *Visual Data Mining - Theory, Techniques and Tools for Visual Analytics*, volume 4404 of *LNCS*. Springer, 2008.

[28] P. Simonetto, P. Y. Koenig, F. Zaidi, D. Archambault, F. Gilbert, T. T. Phan-Quang, M. Mathiaut, A. Lambert, J. Dubois, R. Sicre, M. Brulin, R. Vieux, and G. Melançon. Solving the traffic and flitter challenges with tulip. In *IEEE Proc. on Visual Analytics Science and Technology*, pages 247–248, 2009.

[29] L. Singh, M. Beard, L. Getoor, and M. Blake. Visual mining of multimodal social networks at different abstraction levels. In *IV '07. 11th International Conference*, pages 672–679, 2007.

[30] C. Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann, 2004.

[31] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, Cambridge, 1994.