VisCFSM: Visual, Constraint-Based, Frequent Subgraph Mining

Jennifer L. Leopold
Missouri University of Science & Technology
Department of Computer Science
Rolla, Mo USA
leopoldj@mst.edu

Chaman L. Sabharwal
Missouri University of Science & Technology
Department of Computer Science
Rolla, MO USA
chaman@mst.edu

Nathan W. Eloe
Northwest Missouri State University
School of Computer Science and Information Systems
Maryville, MO USA
nathane@nwmissouri.edu

Abstract—**Graphs long have been valued as a pictorial way of representing relationships between entities. Contemporary applications use graphs to model social networks, protein interactions, chemical structures, and a variety of other systems. In many cases, it is useful to detect patterns within graphs. For example, one could be interested in identifying frequently occurring subgraphs, which is known as the frequent subgraph mining problem. A complete solution to this problem can result in numerous subgraphs and can be time-consuming to compute. An approximate solution is faster, but is subject to static heuristics that are beyond the control of the user. Herein we present VisCFSM, a visual, constraint-based, frequent subgraph mining system which allows the user to dynamically specify a variety of constraints on the subgraphs to be found while the mining algorithm is running. The constraint specification interactions are performed through a visual user interface, thereby facilitating a form of visual algorithm steering. This approach can be integrated with any frequent subgraph mining algorithm. Most importantly, this approach has the potential for the user to better, and more quickly, find the information that is of most interest to him/her in a graph.**

Keywords-graph; data mining; visual algorithm steering

I. Introduction

Graphs long have been valued as a pictorial way of representing complex relationships between entities. Commercial, research, and government organizations use graphs to model social networks, protein interactions, chemical structures, and a variety of other systems. A common application of graph data mining is to identify the most recurrent relationships or patterns amongst the data in a graph, which typically requires finding frequently occurring subgraphs.

For some applications, the input will be a collection of relatively small graphs, and the search for frequent subgraphs is performed over each individual graph in the collection before those results are combined. This is known as a *graph-transaction setting*. In contrast, the input may be a single graph; this is referred to as a *single graph setting*. Our work refers to the latter environment. We also restrict our work to *static graphs*, and do not address *dynamic graphs* or *streaming graphs*, which are discussed in [1].

DOI reference number: 10.18293/DMS2016-026

Formally, we define the Frequent Subgraph Mining (FSM) problem as in the paper by Abedijaberi [2] using Definitions 1-4 given below.

Definition 1. *A labelled graph $G = (V, E, L_V, L_E)$ consists of a set of vertices V, a set of undirected or directed edges E, and two labeling functions L_V and L_E that association labels with vertices and edges, respectively.*

It should be noted that the labels of any two vertices (or any two edges) may not be unique. However, each vertex (and each edge) will have a unique *id*.

Definition 2. *A graph $S = (V_S, E_S, L_{VS}, L_{ES})$ is a subgraph of $G = (V, E, L_V, L_E)$ iff $V_S \subseteq V$, $E_S \subseteq E$, $L_{VS}(v) = L_V(v)$ and $L_{ES}(e) = L_E(e)$ for all $v \in V_S$ and $e \in E_S$.*

Definition 3. *A subgraph isomorphism of S to G is a one-to-one function f: $V_S \rightarrow V$ where $L_{VS}(v) = L_V(f(v))$ for all vertices in $v \in V_S$, and for all edges $(u,v) \in E_S$, $((f(u),f(v)) \in E$ and $L_{ES}(u,v) = L_E((f(u), f(v))$.*

Definition 4. *Let I_S be the set of isomorphisms of a subgraph S in graph G. Given a minimum support threshold τ, the frequent subgraph mining problem (FSM) is to find all subgraphs S in G such that $|I_S| \geq \tau$.*

The advantage of limiting frequent subgraphs to only those with disjoint edges is computational tractability [3]. But this comes at the expense of disregarding potentially useful information. Hence, in our work we allow isomorphic subgraphs to share edges.

FSM algorithms that find complete solutions may, depending upon the specified threshold value and the size of the graph, result in numerous subgraphs and take a considerable amount of time to compute. Algorithms that find approximate solutions are faster, but apply static heuristics that are beyond the control of the user (unless s/he modifies the software).

Herein we present VisCFSM, a visual, constraint-based, frequent subgraph mining system which allows the user to dynamically specify a variety of constraints on the subgraph mining algorithm while it is running. The constraint specification interactions are performed through a visual user interface, thereby facilitating a form of visual algorithm steering.

The prototype implementation uses the FSG [4] frequent subgraph algorithm; however, the approach we employ can be integrated with any FSM algorithm. Most importantly, this approach has the potential for the user to better, and more quickly, find the information that is of most interest to him/her in a graph.

The organization of this paper is as follows. Section II provides a brief overview of related work in graph data mining. Motivation for the need for dynamic, visual steering of FSM using constraints is presented in Section III. In Section IV, we discuss the VisCFSM infrastructure in terms of the FSM, the constraint satisfaction system, and the graphical user interface. An example of running VisCFSM is presented in Section V. Finally, we discuss our plans for future work in Section VI and conclusions in Section VII.

II. RELATED WORK

A. *Graph Data Mining Algorithms*

Graph Data Mining (GDM) algorithms are divided into three main categories: Graph Theory Based, Inductive Logic Programming, and Greedy Search [5]. Our work focuses on the Graph Theory Based category, which consists of two main groups: Apriori-based and pattern growth-based approaches. Algorithms in the first group generate candidate subgraphs by joining two frequent subgraphs of the same size to generate larger subgraphs. Pattern growth algorithms generate candidates by adding a new edge to each smaller frequent subgraph.

FSM algorithms typically face two computational challenges: (i) candidate subgraph generation, and (ii) identification of candidate subgraphs that meet the minimum support threshold. In the worst case, all subgraphs in the graph must be examined, which is exponential in complexity, and subgraph isomorphisms must be computed, which is an NP-complete problem. FSM algorithms may attempt to improve runtime performance by reducing the size of the search space, avoiding duplicate comparisons, and/or minimizing the amount of memory required for compiling intermediate results. Another solution to reduce the runtime is to provide an approximate, rather than a complete, solution to the FSM problem.

B. *Heuristics for Approximate Solutions*

Heuristic FSM algorithms such as SUBDUE [6], GREW [7] and GRAMI [8] discover only a subset of all frequent subgraphs of a graph. These algorithms do not return any infrequent patterns (i.e., the results do not have false positives), but may miss some frequent ones (i.e., the results effectively may have false negatives). The type of heuristics that are employed are quite diverse, and also vary considerably in their degree of complexity. Some examples are listed below:

- SUBDUE [6] starts with frequent subgraphs consisting of a single vertex, and then expands those in a breadth-first manner by adding a new edge. The order of processing is known as a “beam search”, and only a predetermined number of paths (i.e., the beam width) are kept as candidates at each iteration. Hence some valid frequent subgraphs will be missed.
- Like SUBDUE, GREW [7] employs a beam search to prune large portions of the search space. It also iteratively joins frequently occurring pairs of nodes into a single supernode, and determines disjoint embeddings of connected subgraphs using a maximal independent set algorithm. GREW employs an additional heuristic that deliberately underestimates the frequency of each discovered subgraph in an attempt to reduce the search space. While experiments showed that GREW significantly outperformed SUBDUE with respect to runtime, those experiments showed that this came at the expense of finding fewer frequent subgraphs.
- Pattern growth algorithms generate candidate subgraphs by adding a new edge to smaller frequent subgraphs. GRAMI [8] only adds frequently occurring edges to smaller frequent subgraphs when generating candidate subgraphs. This will miss finding some valid frequent subgraphs, but reduces the total number of iterations over edges that must be considered.
- AGRAMI [8] is an extension of GRAMI that employs additional heuristics in an effort to scale to larger graphs. For example, it enforces a timeout when testing whether a subgraph occurs at least as many times as the minimum support threshold; if the solution cannot be computed within a particular amount of time, that subgraph is assumed to be infrequent.

In the same paper that presents GRAMI and AGRAMI [8], the authors briefly discuss CGRAMI, a version of GRAMI that seeks to find more general patterns in graphs than just frequent subgraphs. This work is noteworthy to mention herein because it claims to support the following user-defined constraints:

- Number of vertices (or edges) in a pattern cannot exceed a specified value
- Vertex degree in a pattern cannot exceed a certain value
- A pattern must include/exclude only vertices with certain labels
- A pattern must include only certain edges
- A pattern cannot include certain edges
- A pattern cannot include a specified subgraph
- A specified vertex label cannot appear more than N times in a pattern

To specify desired constraints in CGRAMI, the user must comment out certain lines of code (and uncomment other lines) for the constraints, set the values for parameters, and then recompile the program. The program has a command-line interface; there is no graphical user interface.

As stated previously, what all heuristic FSM (and constraint-based GDM) algorithms have in common is the inability for the user to dynamically customize the heuristics, or any form of constraints, while the algorithm is running. This is the novel contribution of the work presented herein.

III. Motivation

In part, motivation for this work came from a graduate course on Advanced Data Mining taught by author Leopold in 2015 at Missouri University of Science and Technology. That year the focus of the course was graph data mining. Students read several research papers on GDM algorithms and applications. Some of the students in the class implemented a few of the algorithms in Python, but were frustrated that they had to wait a considerable amount of time for the computation on some relatively small graphs (e.g., a graph of 50 vertices with average vertex degree 3.5 took over 8 hours to compute all frequent subgraphs of minimum support 2). When they had their programs output intermediate results as the subgraphs were being found, sometimes the students would terminate the program, and restart it with a different threshold value to further discriminate the result set and make the program finish more quickly.

At the end of the course, the students were asked what kinds of constraints they would have found useful to "steer" a FSM algorithm dynamically, even if it meant that the resulting set of subgraphs would not be complete. Here we use the term *steering* as discussed in [9]: the ability to have a continuous visualization of the (output) data as a program executes, coupled with the ability for the programmer to interactively modify any aspect of the program and see the effects without restarting the computation.

With a social network (specifically, a terrorist network) as an application domain, the students identified the constraints and use case examples listed below. In this social network, it is assumed that a vertex in the graph is labelled with a person's name, which are not necessarily unique. Additional information about a person and his/her relationship to other people may be represented in the graph as vertex or edge data.

- **Include/exclude subgraphs containing a certain set of vertices.** Ex.: Suppose that we're using a social media network to identify terrorist threats. The number of frequent subgraph results may be quite high at first due to very small terrorist groups. So we then want to narrow our search, and only continue to look for subgraphs that include a specific group of people that we know conduct terrorist activities.
- **Include/exclude only frequent subgraphs that appear more/less than subgraph (or vertex) X does.** Ex.: We see a specific name in the preliminary results of our search that we already know is a leader and a threat. But his name isn't the only one we see, and we want to know who in the group is more important, of high rank, or higher rank than this person. So we then narrow our search to find subgraphs that appear more often than those containing this person. Or maybe we are looking for someone we can capture and get information from, in which case we look for someone important who appears less often.
- **Include/exclude only frequent subgraphs that are disconnected/connected to subgraph (or vertex) X.** Ex.: We begin a search on terrorist cells. However, based on seeing a particular group appearing frequently in the results, we want to narrow our search to those connected to that group. Similarly, if we are trying to identify new terrorist cells or rival cells, we may want to look only at those groups that are disconnected from a certain group.
- **Include/exclude only frequent subgraphs where the average vertex degree is greater than some number.** Ex.: We're looking for potential terrorist cells, and not interested in groups with only a couple of connections; such groups are unlikely to be funded or be a real threat. We may not see this until after we have seen the initial (small-sized) frequent subgraphs.
- **Include/exclude subgraphs containing a certain number of edges.** We may not be interested in seeing small terrorist groups, but rather want to see a certain amount of interconnectivity; these might prove to be the more dangerous terrorist groups.
- **Change minimum support.** Ex.: We may start our search very wide open, but, after seeing some preliminary results that are too numerous and/or contain trivial information (e.g., everyone is a potential terrorist), decide that we want to raise the threshold.

It should be noted that these constraints are not intended to be mutually exclusive, but rather conjunctive; we should be able to specify any combination of constraints.

In the next section, we discuss the VisCFSM infrastructure in terms of the FSM, the constraint satisfaction system, and the graphical user interface. The system was designed to address many of the above listed constraints.

IV. VisCFSM

The infrastructure of VisCFSM consists of a front end and a back end. The front end is comprised of the graphical user interface which displays the frequent subgraphs as they are computed, and allows the user to visually steer the FSM by specifying constraints on frequent subgraph selection as the algorithm is progressing. The back end consists of the FSM and the constraint satisfaction system. In this section we briefly discuss each part of the infrastructure.

A. The FSM

As mentioned in Section II, we have chosen to focus on the Graph Theory Based category of graph data mining algorithms, which consists of Apriori-based and pattern growth-based approaches. For the prototype implementation of VisCFSM we chose a pattern growth algorithm, FSG [4]. The algorithm starts by finding all frequent subgraphs consisting of one edge. It then makes repeated iterations, generating candidates by adding a new edge to each of the largest frequent subgraphs found so far. This particular algorithm was selected primarily for its simplicity; it is certainly not one of the most efficient FSM algorithms that exists, but we believed that the logic upon which it is based could easily be understood by most users. The choice of FSM algorithms to be used in VisCFSM is not important; the constraint satisfaction system and visualization control system that we employ actually can be integrated with any FSM algorithm.

B. *The Constraint Satisfaction System*

Inspired by the use cases presented in Section III, several structural and semantic constraints have been implemented for VisCFSM. These are listed below:

- Include/exclude frequent subgraphs that contain certain vertices or edges
- Include/exclude frequent subgraphs that include a particular subgraph
- Include subgraphs that are connected/disconnected to a particular vertex or edge
- Include only frequent subgraphs that have at least one vertex that has degree greater than a specified number
- Include only frequent subgraphs where the average vertex degree greater than a specified number
- Include/exclude frequent subgraphs containing a certain number of edges
- Exclude frequent subgraphs where a certain vertex label appears greater than a specified number of times
- Change minimum support

The user interface allows the user to specify the constraints that should be applied to the set of frequent subgraphs found so far, and whether to continue applying these constraints in the next iteration of the algorithm in an effort to find new frequent subgraphs (e.g., in the case of the FSG algorithm, the next iteration adds an edge to each of the largest-sized frequent subgraphs found so far in order to form new candidate frequent subgraphs).

C. *The Graphical User Interface*

The VisCFSM FSM and constraint satisfaction system were implemented in SWI-Prolog. The choice of a logic programming language seemed most suitable for modeling a constraint satisfaction problem. However, SWI-Prolog has no graphical capabilities. Hence, the VisCFSM graphic user interface was developed in Python.

The graphic user interface (GUI) consists of the following controls: (i) a file chooser to allow the user to select a Prolog file that contains the specification of a graph, (ii) a text input field to specify the name of the graph (i.e., a Prolog file may contain multiple graph specifications, each defined as a relation), (iii) a text input field to specify the minimum support threshold for considering a subgraph to be frequent, (iv) a constraint editor, (v) a control button to start the FSG by finding the smallest-sized FSGs, and (vi) a control button to add an edge to each of the largest FSGs found thus far. A graph specification consists of a Prolog list containing the list of vertices (in the format [ID, label]) and a list of edges, where each edge is represented as a list of two vertices. Fig. 1 shows the GUI after an undirected graph named *sampleGraph* has been loaded from a Prolog file named *graph.pl*. In this figure, no frequent subgraphs have been found yet.

The constraint editor allows the user to set up rules to filter the frequent subgraphs that will be reported. Examples of the constraint editor are shown in Fig. 2 and Fig. 3. Constraints are represented in Disjunctive Normal Form (DNF); that is, as a series of AND clauses OR'd together. The editor includes a drop-down menu of the possible constraints, a text input field for specifying the arguments to a constraint, and a display of the DNF clauses that have been specified so far. Help text is also provided to guide the user in specifying the arguments correctly.

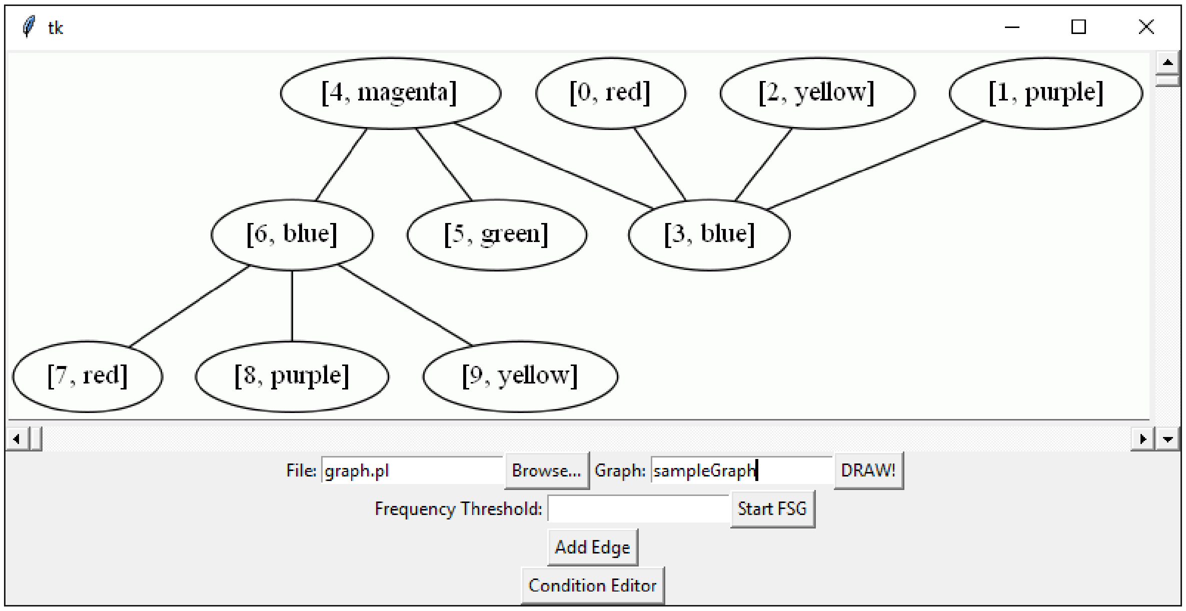

Figure 1. VisCFSM GUI after a sample graph has been loaded

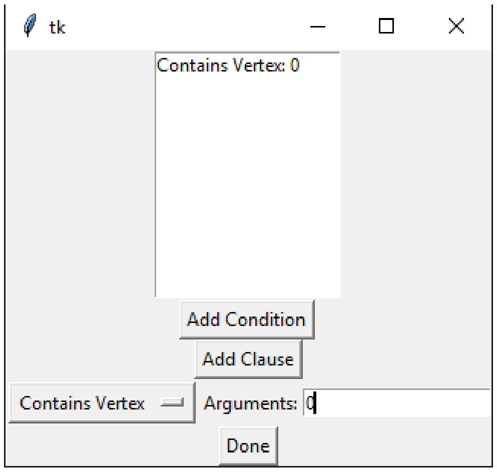

Figure 2. Constraint editor showing one clause

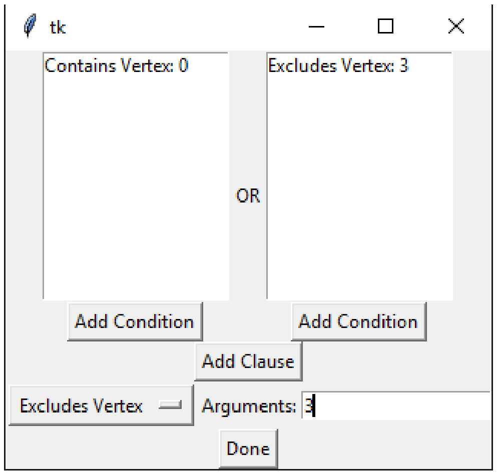

Figure 3. Constraint editor showing two clauses

The main display area in the GUI initially shows the graph that the user has specified from the selected Prolog file. Once the FSG algorithm is invoked, that area of the GUI is used to display the frequent subgraphs found in the most recent iteration of the algorithm. Recall that FSG starts by finding all frequent single-edge subgraphs, then makes repeated iterations, adding a new edge to each of the largest frequent subgraphs found so far. In future refinements of the GUI, the user will be given the ability to scroll back to previously displayed sets of (smaller-sized) frequent subgraphs, and also will be given the option to undo/redo the application of constraints and edge additions. With the current implementation, at any time, the user may restart the FSG generation algorithm from the beginning by clicking on the **Start FSG** button.

V. An Example in VisCFSM

To demonstrate the concepts behind VisCFSM, here we walk through two simple examples. We start by assuming that we have reached the state shown in Fig. 1, having specifying the file *graph.pl* and selecting *sampleGraph* as the desired graph. Clicking on the **DRAW!** button renders the graph without finding any frequent subgraphs.

For this particular graph, a minimum support threshold of 2 provides interesting results. Fig. 4 shows all frequent subgraphs with a minimum support of 2. These are listed individually with their unique vertex IDs rather than, for example, simply reporting that a subgraph with edge *(blue, red)* occurs at least 2 times; there is, however, an option in the GUI to report the results only by unique label combinations in the subgraphs.

Suppose that we want only subgraphs that contain a specific vertex, say those with the ID 0. The user would first click on the **Condition Editor** button, which will open the condition editor window. In the condition editor, the user would then select "Contains Vertex" from the dropdown menu, and specify the ID of the vertex to include; see Fig. 2. Finally, the user would click on the **Add Condition** button. The constraint editor can be closed by clicking on **Done** or closing the window.

Upon completing those steps, if the user now clicks the **Start FSG** button, the FSG algorithm will be restarted with the specified constraint applied; these results are shown in Fig. 5. At this point, the user may increment the size of these subgraphs (by clicking on the **Add Edge** button), or go back to the constraint editor, add additional constraints to the current AND clause, and recompute the set of frequent single-edge subgraphs.

To add an alternative set of constraints (i.e., add another clause that will be OR'd in the DNF representation), the user can open the constraint editor and click on the **Add Clause** button. A new clause column will be displayed, and the user now can add constraints to either of the clauses. Fig. 3 shows the addition

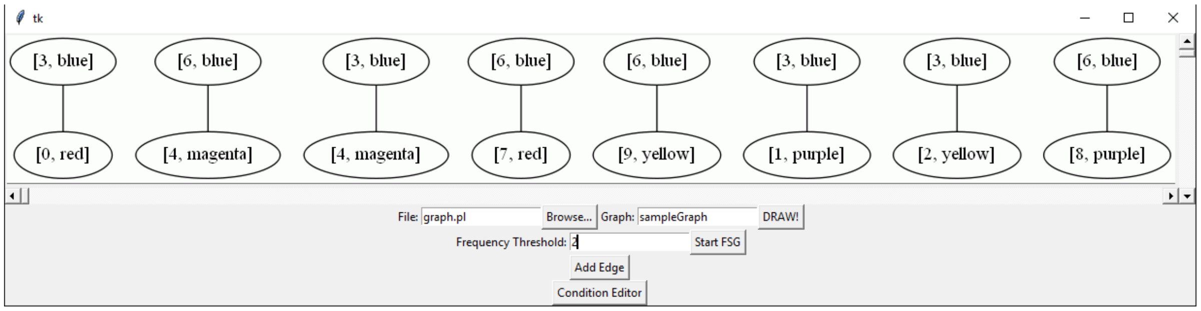

Figure 4. Subgraphs from the graph shown in Fig. 1 that occur with minimum support 2

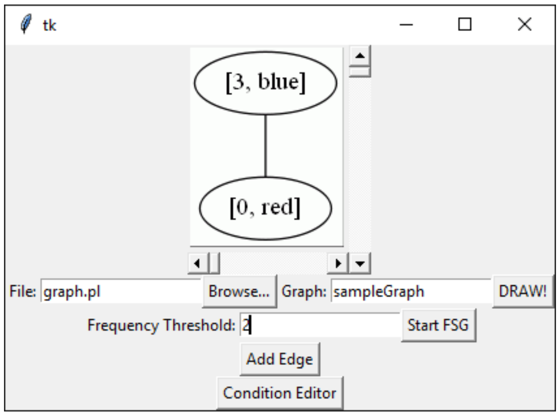

Figure 5. Results of applying the constraint specified in Fig. 2 when finding single-edge frequent subgraphs for the graph shown in Fig. 1

of another constraint in our example; namely we also want frequent subgraphs that exclude the vertex with ID 3. The resulting set of frequent single-edge subgraphs (obtained by clicking the **Start FSG** button) is shown in Fig. 6. The result set now includes single-edge subgraphs that occur at least 2 times, and include vertices with ID 0 or exclude vertices with ID 3.

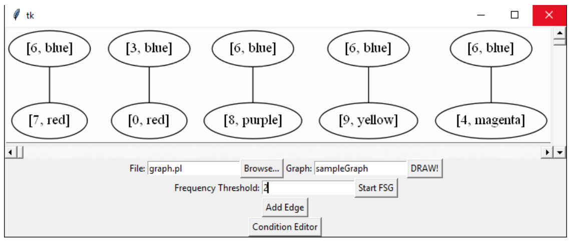

Figure 6. Results of applying the constraints specified in Fig. 3 when finding single-edge frequent subgraphs for the graph shown in Fig. 1

VI. Future Work

Future work on VisCFSM will be focused on three areas. First, we intend to perform informal studies of the usefulness and usability of the system when author Leopold again teaches the graduate course on Advanced Data Mining at Missouri University of Science and Technology in 2017. This should yield ideas for improving existing features in the user interface and adding new constraints.

Secondly, we plan to explore the incorporation of specifications of constraints by natural language and/or gestures/drawing in the user interface. The use of multimodal user interfaces for spatially-oriented applications have been of interest for years, particularly for GIS applications; see the discussion in [10], for example. Such interfaces have become a particularly active research topic in the past few years with the prevalence of mobile GIS applications (e.g., [11], [12]). Many of the same concepts for querying general spatial settings (e.g., objects such as buildings, connections between objects such as roads, directionality, etc.) should also be applicable to specifying patterns in a graph.

The third focus for future work will be improving the runtime efficiency of VisCFSM. Most algorithms used for graph data mining are designed for implementations in procedural languages. It is possible that some of our FSM operations (written in Prolog) could be sped up by expressing the steps of those algorithms (e.g., testing for graph isomorphism) in a more logical form rather than mimicking procedural solutions.

Additionally, many parts of the algorithm are embarrassingly parallel in nature; operations that are performed on every subgraph in the list of frequent subgraphs are independent from those being applied to other subgraphs, and as such can be done in a parallel manner. SWI-Prolog offers the concurrent/3 and concurrent_maplist/N predicates that automatically distribute the execution of goals across multiple threads. As threading introduces significant overhead to the process, determining the appropriate size of problems to apply concurrent operations will require empirical experimentation so as to not degrade the performance of the system.

Another option for exploring concurrency in VisCFSM includes using a Prolog interpreter embedded in the language in which the GUI is implemented. Javascript, for example, has a library that is a Prolog interpreter implemented in Javascript. This would allow the GUI to use the threading and GUI features of the host language (in Javascript's case, Electron or NW.js could provide the GUI features) while preserving the Prolog computational backend. This has the disadvantage of requiring the GUI to manage communication between threads, but the higher-level threading abilities may make this a preferable option to sending queries to a Prolog process.

If reconciling the data representations of Prolog and a host language prove to introduce more complexity than is needed, a domain-specific logic language that integrates tightly with other languages (such as miniKanren [13]) would provide a more native approach to interacting with the computational backend. This would provide access to high-level threading and GUI capabilities of more general-purpose programming languages while leveraging the constraint programming capabilities of a logic programming language.

VII. Summary and Conclusions

Herein we have presented the infrastructure for a graph mining system that provides the ability for the user to dynamically customize a variety of semantic and structural constraints while the algorithm is working to complete its overall task. Effectively, this system supports visual algorithm steering, providing the ability for the user to continuously visualize the results of the graph mining program as it executes, interactively modify aspects of the program, and see the effects without restarting the computation from the very beginning. Such capabilities are extremely valuable when dealing with graph mining, wherein the data representation is intrinsically visual, and patterns of interest may not become obvious until preliminary results are seen. Because frequent subgraph mining is a computationally intensive problem, the ability to dynamically adjust constraints on the computation can allow the user to more quickly find the information that is of most interest to him/her in a graph.

REFERENCES

[1] A. Bifet, G. Holmes, B. Bfahringer, and R. Gavald'a, "Mining frequent closed graphs on evolving data streams", Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, 2011, pp. 591-599.

[2] A. Abedijaberi and J.L. Leopold, "FSMS: a frequen subgraph mining algorithm using mapping sets," Proceedings of the 12th International Conference on Machine Learning and Data Mining, New York, NY, 2016.

[3] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph," Data Mining and Knowledge Discovery, 11.3, 2005, pp. 243-271.

[4] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," Proceedings of the 2001 IEEE International Conference on Data Mining, IEEE Computer Society, 2001.

[5] M. Gholami and A. Salajegheh, "A survey on algorithms of mining frequent subgraphs," International Journal of Engineering Inventions, 1.5, 2012, pp. 60-63.

[6] D.J. Cook and L.B. Holder, "Substructure discovery using minimum description length and background knowledge, " Journal of Artificial Intelligence Research, 1(1), 1994, pp. 231-255.

[7] M. Kuramochi and G. Karypis, "An efficient algorithm for discovering frequent subgraphs," IEEE Transactions on Knowledge and Data Engineering, 16(9), September 2004, pp. 1038-1051.

[8] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GRAMI: frequent subgraph and pattern mining in a single large graph," Proceedings of VLDB Endowment, 2014, pp. 517-528.

[9] J.W. Atwood, Jr., M.M. Burnett, R.A. Walpole, E.M. Wilcox, and S. Yang, "Steering programs via time travel", 1996 IEEE Symposium on Visual Languages, Boulder, CO, 1996, pp. 1-8.

[10] I. Schlaisich and M.J. Egenhofer, "Multimodal spatial querying: what people sketch and talk about", 1st International Conference on Universal Access in Human-Computer Interaction, 2001, pp. 732-736.

[11] F. Cutugno, V.A. Leano, R. Rinaldi, and G. Mignini, "Multimodal framework for mobile application", Proceedings of the International Working Conference on Advanced Visual Interfaces, New York, NY, 2012, pp. 197-203.

[12] J. Doyle, M. Bertolotto, and D. Wilson, "Evaluating the benefits of multimodal interface design for CoMPASS – a mobile GIS", GeomInformatica, Volume 14, Issue 2, April 2010, pp. 135-162.

[13] C.E. Alvis, J.J. Wilcock, K.M. Carter, W.E. Byrd, and D.P. Friedman, "cKanren miniKanren with constraints", 2011.