Visualized Subgraph Search[*]
VAST 2009 Flitter Mini Challenge Award: Good Analytical debrief

Dóra Erdős Zsolt Fekete András Lukács
Computer and Automation Research Institute (MTA SZTAKI), Hungarian Academy of Sciences
Data Mining and Web Search Group
{edori, zsfekete, lukacs@ilab.sztaki.hu}

ABSTRACT

We present a visually supported search and browsing system for network-type data, especially a novel module for subgraph search with a GUI to define subgraphs for queries. We describe how this prototype was applied for the Vast Challenge 2009, Flitter Mini Challenge.

Keywords: Visual Analytics, Heterogeneous Graph Visualization

Index Terms: E.1 [Data]: Data Structures—Graphs and networks; K.6.1 [Visual Analytics]: Heterogeneous Graph Visualization—

1 INTRODUCTION

In the Vast 2009 Flitter Mini Challenge data set, the connections between the Flitter network users were given along with additional geographical location data. The task was to identify a criminal network defined by a certain structure of the users. Our main idea was to represent the Flitter users' network as a graph with vertices corresponding to users and edges to their connections. In this representation all information given in the textual description of the criminal network could be fully converted by the analyst into constraints on the number of connections and geographic location of members in specific roles. When regarding the Flitter network as a graph, this description defines two subgraph structures for the two scenarios that have to be identified.

Our in-house search and visualization tool was designed for search and browsing in large scale network-type data. For the VAST 2009 Challenge we developed a new module for general subgraph search extending the previous single node or edge based queries. The new subgraph based search seems fulfill the requirements arisen in the Flitter Mini Challange and in more general in the targeted application area. The technical novelty of our method lies in the user interface for subgraph structure description.

Juba et. al. ([1])designed a system with a similar approach for searching for spanning trees in a graph.

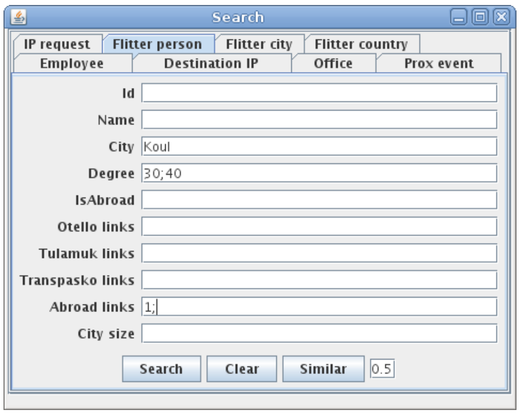
In the next section we present the architecture of our system. In Section 2.2 we describe the query language and in Section 2.3 the subgraph search UI. Finally in Section 3 we show how we used the subgraph search on the Flitter data.

2 OUR SEARCH AND VISUALIZATION SYSTEM

2.1 Architecture of the Search System

The central notion of the system is an *entity* with several attributes. In the Flitter dataset the attributes include Id, $Name$, $City$, $Degree$, $IsAbroad$, $OtelloLinks$, $TulamukLinks$, $TranspaskoLinks$, $AbroadLinks$ and $CitySize$. The type of an attribute can be integer, double or string. A sample entity can be $e = \{Id : 1,\ Name : @irvin,\ City : Koul,\ Degree : 300,\ IsAbroad : 0,\ OtelloLinks : 10,\ TulamukLinks : 9,\ TranspaskoLinks : 5,\ AbroadLinks : 24,\ CitySize : 4\}$. We note that our system can handle multiple type of entities.

[*]Supported by the EU FP7 project LiWA—Living Web Archives and by grant OTKA NK 72845.



Id	
Name	
City	Koul
Degree	30;40
IsAbroad	
Otello links	
Tulamuk links	
Transpasko links	
Abroad links	1;
City size	

Figure 1: A query window set for an entity match query

The system has a server part, which has been implemented in C++, and a Java client, which provides the graphical user interface. We mention that the data loaded into the system can be arbitrarily large, we use an embedded database manager for storing data. Though in case of the Flitter dataset the total amount of data was stored in the memory cache.

One of the main functionalities of the user interface is search: the user can define a *query* by (partly) filling a query form. We say that an entity *matches* a query if the appropriate attributes of the entity are in the range required by the query. For example the entities matching query $q = \{City : Koul,\ Degree : [30, 40],\ AbroadLinks : [1, \infty)\}$ are the Flitter persons that live in Koul, have degree between 30 and 40 and have at least one abroad link.

Additional functionalities of the interface include automatic layout, zoom, delete, save graph and graph export into an image file. In order to browse, the user may double click on a node to display the neighbors of this node. Some natural graph algorithms are also included such as finding shortest paths or specified node pairs.

2.2 Query language

Subgraph search is composed of advanced search operators described below. An advanced search $A = (S, C_1, \ldots, C_k)$ contains an *base search* S and *constraints* $C_1, \ldots, C_k$ (where k can be 0).

Every base search S defines a result set, which is a set of entities. A base search S can be one of the following.

- $EntitySearch(q)$, where q is a query. The result set of this is the set of entities matching query q.
- $NeighborSearch(e_1, \ldots, e_n)$, where e_i is an entity. The result set is the set of entities that are connected to every entity e_i by an edge of the graph.

Every constraint is a boolean valued function on entities. A constraint C_i can be the following:

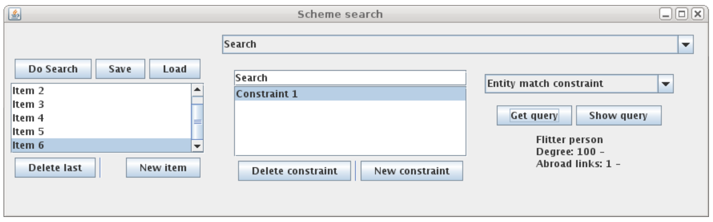


Figure 2: A window for assembling a subgraph structure. It shows how the EntityConstraint is set.

- $EntityConstraint(q)$ is true for entity e, if e matches query q
- $NeighborConstraint(e_1,\ldots,e_n)$ is true for e, if e is a common neighbor of $e_1,\ldots,e_n$
- $NonNeighborConstraint(e_1,\ldots,e_n)$ is true for e, if e is not linked to any of entities $e_1,\ldots,e_n$.
- A $CompareConstraint$ is defined by an entity e_0 an attribute $attr$ of entity e_0 and a symbol $\alpha \in \{<,>,\leq,\geq,=,\neq\}$. The value of $CompareConstraint(e_0, attr, \alpha)$ on entity e is the logical value of the following expression: $e_0.attr\ \alpha\ e.attr$.

The result set of an advanced search $A = (S, C_1, \ldots, C_k)$ is the set of entities that are included in the result set of the base search S and satisfy all the constraints.

For example let $S = NeighborSearch(e,f)$, $C_1 = NonNeighborConstraint(g)$, $C_2 = EntityConstraint(q)$, where $q = \{City : Koul, Degree : 20\}$ and let advanced search $A = (S, C_1, C_2)$. Then the result set of A is the set of entities that are connected to e and f, not connected to g, have degree 20 and are living in $Koul$.

2.3 Search for Subgraphs

The search of a subgraph is defined by a list of advanced search expressions $A_1,\ldots,A_n$, so that in A_i $(i \geq 2)$ the e_i entities appearing in $NeighborSearch$, $EntitySearch$, $NeighborConstraint$, $NonNeighborConstraint$ and $CompareConstraint$ can be a symbol (variable) $Item_j$ where $j < k$. We say that an n-tuple of entities $e_1,\ldots,e_n$ matches the search list if e_k is in the result set of A_k where the symbols $Item_j (j < k)$ were substituted by e_j.

For better understanding we illustrate the above definitions by a specific example. In the example below we are looking for a "cherry" in the graph, i.e. we would like to find e_1, e_2, e_3 so that e_1e_2 and e_1e_3 are edges of the graph but e_2e_3 is not an edge. And suppose that we expect the three entities to be vertices with degrees between 30 and 40. So in this case we would define our structure in the following way. Let q_0 be the following query $\{Degree : [30;40]\}$. Now define the following advanced searches A_1, A_2, A_3.

- $A_1 = (S^1)$ (without constraints), where
  - $S^1 = EntitySearch(q_0)$
- $A_2 = (S^2, C_1^2)$, where
  - $S^2 = NeighborSearch(Item_1)$
  - $C_1^2 = EntityConstraint(q_0)$
- $A_3 = (S^3, C_1^3, C_2^3, C_3^3)$, where
  - $S^3 = NeighborSearch(Item_1)$,
  - $C_1^3 = EntityConstraint(q_0)$
  - $C_2^3 = NonNeighborConstraint(Item_2)$
  - $C_3^3 = CompareConstraint(Item_2, id, <)$

The $CompareConstraint$ C_3^3 is needed because without that constraint we would get all results twice. Namely $e_1e_2e_3$ would be a solution and its permutation $e_1e_3e_2$ too.

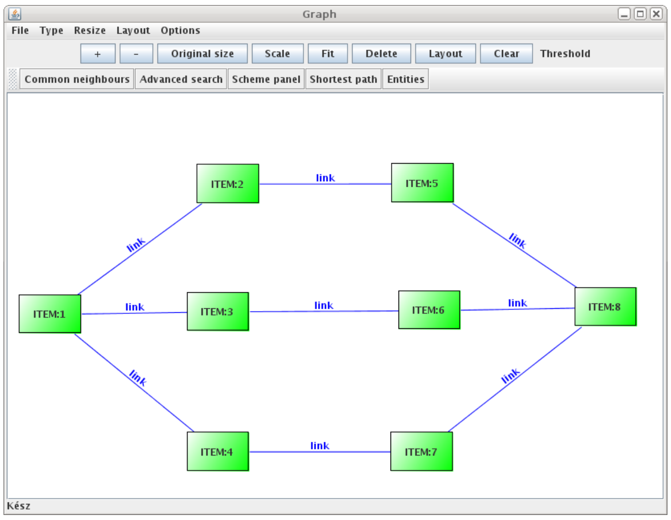


Figure 3: A structural search for scenario B

We implemented a user interface for defining subgraph structures. With this approach one can define an arbitrary subgraph search. Moreover in this way the user does not only specify the expected subgraph, but specifies the algorithm executed by the server, because the sequence $A_1,\ldots,A_n$ naturally defines a backtrack algorithm for finding the desired subgraph. The running time of this algorithm is exponential in the number of vertices of the subgraph.

3 Subgraph Search over the Flitter Data

For both scenarios we set up a structure in our graph visualization tool. In scenario A we were looking for a subgraph with six while in scenario B eight vertices. We set constraints on the degree of the vertices and $NeighborConstraints$ and $NonNeighborConstraints$ where needed. At first we did not put degree constraints on the vertices corresponding to the middlemen and obtained 12 hits for scenario A and over a thousand for scenario B. After imposing the appropriate degree constraints on the middlemen (the degree of the middleman is no more than 6 in scenario A and no more than 4 in B) we obtained two hits for the first scenario and none for the second.

From the two hits we got for scenario A we manually chose the subgraph where the geographic location of members in specific roles fit the description better.

4 Conclusions and Future Work

We solved the Vast Flitter Mini Challenge via a new subgraph search interface in our graph search and visualization tool. Our experience was that we could define subgraph search in a very flexible way, it was easy to modify parameters on the desired subgraph.

Since we use a backtrack algorithm the program could be easily parallelized which would increase computational speed considerably. Currently it is quite complicated to define a scheme. So we plan to improve the user interface of the prototype.

References

[1] D. Juba, T. Chabuk, and C. Hu. Visualizing Search in Networks.