# Noise Removal with Regular Expression in Knowledge Graphs

Pei-Yu Hou Rada Chirkova David Wright

North Carolina State University

## 1 Introduction

With more and more knowledge graphs appear in the real-world data presentation, the complexity of the knowledge graph is increasing, such as heterogeneous graphs in the biomedical area [5, 8]. Analytics on heterogeneous knowledge graphs is challenging because the types of nodes and edges are different. To be efficient, methods of extracting linear paths from knowledge graphs might fail to capture the rich semantics and topology of knowledge graphs. To be effective, methods of using the entire knowledge graph might introduce too much noise irrelevant to the topics.

One of the downstream analytical tasks connecting graphs and data analysis is graph embedding, which converts the graph presentation into the numerical presentation. Among various embedding methods, random-walk based methods are known to be an efficient way to execute such transformation [1, 10]. However, they are originally developed for homogeneous graphs [4, 9].

Without considering any limitations, the random walk will walk through any nodes including those that are irrelevant. It causes inaccurate information extraction and therefore incorrect inference from downstream data analysis. A solution to limit the node types (meta-path) was proposed to avoid such incorrect walks in citation networks [2]. Nevertheless, they assume edge type is unique between two types of nodes. Knowledge graphs in the biomedical area contain more complicated relations between entities.

**Toy Example.** To illustrate, Figure 1 showcases a toy example involving a biomedical knowledge graph. Suppose now we need to execute a graph embedding task for the downstream data analysis. The graph embedding pipeline takes the graph as input shown in Figure 1.

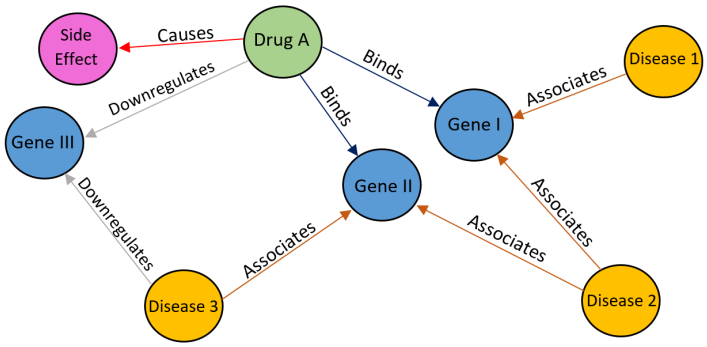



Figure 1: A biomedical knowledge graph

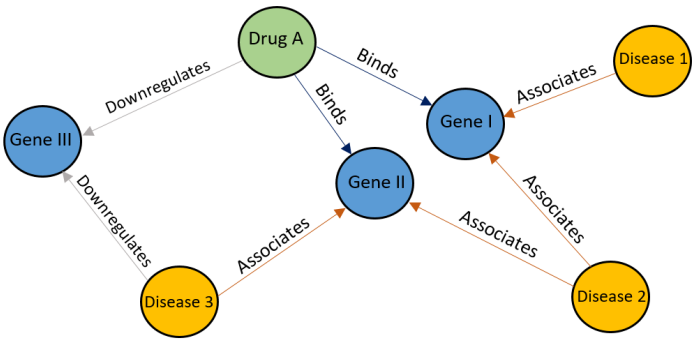
Unlike homogeneous graphs, a heterogeneous graph contains multiple types of nodes and edges that represent more complicated relations. For example, see Figure 1, the user wants to observe and analyze a specific relation (e.g. `Drug` through `BINDS`, `Gene` through `ASSOCIATES`, and then `Disease`).

Suppose our current starting node is `"Drug A"` (which is `Drug`), a random walk can go to any node from `"Drug A"` as long as it connects to it. In other words, the space for the basic random walk is the same as the input graph itself. Hence, the random walk could be `Drug A-[Causes]-Side Effect`, which is clearly not correct.

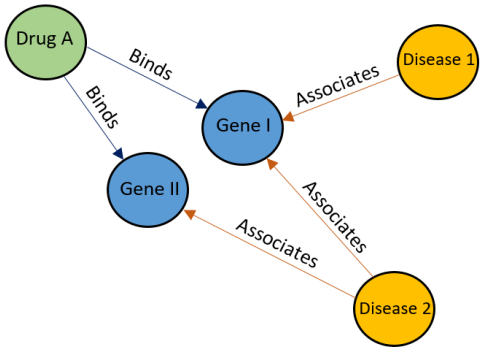
If we consider using a meta-path (e.g. `Drug-Gene-Disease`) to limit the node types, the assigned meta-path will only give the specified types of nodes the possibility to walk through before the random walk is performed. Therefore, a walk from `Drug A` can go through anything, and `Gene` through anything, and then `Disease`, irrelevant nodes have been removed. Figure 2a illustrates the resulting modified random walk space.

While using node limiting meta-path to filter the graph provides a smaller space for executing a random walk, we still have the problem of non-unique edge types that allow undesirable paths. One example is the path `Drug A-[DOWNREGULATES]-Gene III-[DOWNREGULATES]-Disease 3` .

The actual relations that the user wants to observe are `Drug` through `BINDS`, `Gene` through `ASSOCIATE`, and then `Disease`. Edge types are necessary to specify as well. The filtered graph should be like in Figure 2b, so that each path from `Drug A` would not go through unwanted nodes or unwanted edges.




(a) The filtered graph when we consider using the meta-path.




(b) The exact graph that user desires.

.

Figure 2: The motivating example

To capture accurate information and filter irrelevant noise, a user-oriented subgraph construction approach is proposed in this study. Regular expression has been widely used in the applications of finding text patterns in documents as well as in graph databases [6, 13]. In this report, we leverage user-specified regular expression to find semantic subgraphs in heterogeneous knowledge graphs and thus achieve preliminary noise removal for downstream analytical tasks.

The rest of the report is organized as follows. In Section 2, we formulate the problem and introduce relevant definitions. In Section 3, we illustrate our approach in detail. In Section 4, we use a real-world dataset to demostrate the subgraph construction with regular expression patterns and evaluate the noise removal. In Section 5, we provide an extended application with our approach. We conclude this report with some remarks in Section 6.

## 2 Problem Formulation

We formalize the semantic subgraph construction problem in this section.

**Definition 1. Heterogeneous Graph.** A heterogeneous graph is defined as a (directed/undirected) graph $G = (V, E)$ with an object mapping function $\phi : V \to \mathcal{A}$ and a link type mapping function $\psi : E \to \mathcal{R}$. $\mathcal{A}$ and $\mathcal{R}$ are the sets of predefined object types and link types, where $|\mathcal{A}| + |\mathcal{R}| > 2$. The case when $|\mathcal{A}| = 1$ and $|\mathcal{R}| = 1$ is defined as a homogeneous graph.

**Definition 2. Knowledge Graph.**A knowledge graph is defined as a directed graph $G = (V, E, T_V, T_E)$, where each node $v_i \in V$ is assigned a node type and each edge $e_j \in E$ an edge type from $T_V$ and $T_E$ respectively.

In a knowledge graph, nodes are entities and edges are relations so that they construct subject-property-object triple facts. Each path of the form (head entity, relation, tail entity) (denoted as < h, r, t >) indicates a relationship of r from entity h to entity t, where h, t $\in V$ are entities and r $\in E$ is the relation. The form < h, r, t > is called a knowledge graph triplet. Because entities and relations in a knowledge graph are usually of different types, a knowledge graph can be viewed as an instance of the heterogeneous graph. For example, suppose we have a small knowledge graph in Figure 3, where the capital letters (e.g. X, Y, Z, etc.) represent the node types $T_V$, and the small letters (e.g. a, b, c, etc.) represent the edge type $T_E$. An instance of a knowledge graph triplet is <Z8, a, Y1>.

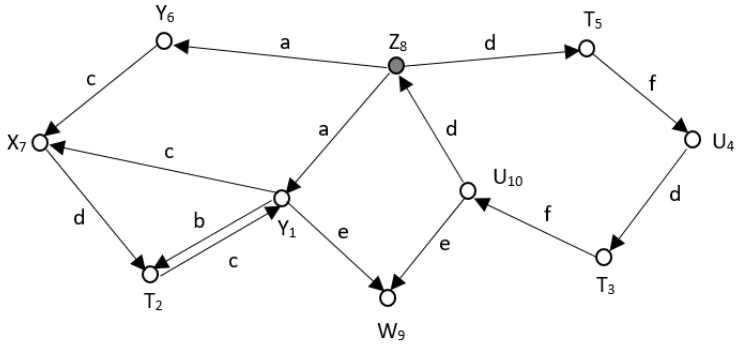



Figure 3: The example knowledge graph

In real-world data, a knowledge graph could contain a lot of unwanted information, especially when the graph is completed from crowdsourcing [12].

In this study, compared to the original complete graph, we explore a pre-filtered subgraph as a potential solution to the noise.

**Definition 3. Subgraph.** A subgraph of $G$ is defined as $G' = (V', E')$, where $V' \subseteq V$, and $E' \subseteq E$.

**Definition 4. Semantic Subgraph.** A semantic subgraph is defined as $Q = \{(V', E', T'_V, T'_E)|$ fits certain conditions that user provides in $G$, such as an the user-specified semantic path $\mathcal{P}\}$, where $V' \subseteq V$, $E' \subseteq E$, $T'_V \subseteq T_V$, $T'_E \subseteq T_E$.

In this paper we solve the following problem:

**Problem. Semantic Subgraph Construction.** Given a knowledge graph $G$, the task is to find a semantic subgraph $G_{v_0,\mathcal{P}}$ for an anchor node $v_0$ under the user-specified semantic path $\mathcal{P}$. To solve this problem, we require the following information as input:

- a heterogeneous knowledge graph $G = (V, E, T_V, T_E)$
- a starting node $v_0$
- a user-specified path $\mathcal{P}$

The output of the problem is a subgraph $G_{v_0,\mathcal{P}} = \{(V', E', T'_V, T'_E)|$at least a path $v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} ... \xrightarrow{e_k} v_k$ exists in $G$ match $\mathcal{P}\}$, where $V' \subseteq V$, and $E' \subseteq E$, $T'_V \subseteq T_V$, $T'_E \subseteq T_E$, and $v_i$ and $e_i, \forall i$ represent nodes and edges it matched semantically the types in path $\mathcal{P}$.

We assume that the semantic subgraph is a small subset of the input graph, with the additional information representing noise to the user. We assert that by using user-specified path pattern $\mathcal{P}$ filtering, we can eliminate most of the noise in the input graph, increasing the efficiency of downstream analytic tasks.

## 3 Semantic Subgraph Construction

In this section, we illustrate our approach of semantic subgraph construction process. We introduce each phase in sequence and then demonstrate some examples of the subgraph construction.

In our subgraph construction problem, the objective is equivalent to match the user-specified path pattern and the languages in the input knowledge graph. In this study, we use regular expression to represent user-specified path pattern. We know that: (1) Rregular expression pattern can

be expanded to multiple possible strings; (2) There may be a lot of matched paths in the input knowledge graph.

To simply our thought, we can view the two types of texts as *user's text* and *graph's text*. We first understand what the user wants and find if those are in the graph. Therefore, below we implement regular expression expansion for understanding what the user wants from *user's text*, graph triplet-string generation, and matched paths searching for finding if those are in the graph in *graph's text*.

### 3.1 Regular Expression Expansion

Our approach to filtering noise from knowledge graphs uses regular expression. In this section, we define the language and regular expression.

Let $\Sigma$ be an alphabet. A word $w$ over an alphabet $\Sigma$ is a finite ordered sequence of symbols in $\Sigma$, e.g., $w = \{s_{i0}, s_{i1}, ..., s_{in} \mid s_i \in \Sigma \text{ for } i \leq |\Sigma| \text{ and } n \geq 0\}$. A language $L$ is an arbitrary set of words over $\Sigma$. Some simple but powerful operations can work with languages, such as union, intersection, difference, concatenation, etc.

**Definition 5. Concatenation of Languages.** Given languages $L_1$ and $L_2$, we define their concatenation to be the language $L_1 \circ L_2 = \{xy | x \in L_1, y \in L_2\}$

Therefore, given languages $L$,

$$L^n = \begin{cases} \{\epsilon\} & \text{if } n = 0 \\ L^{n-1} \circ L & \text{otherwise} \end{cases} \tag{1}$$

i.e., $L_i$ is $L \circ L \circ \cdots \circ L$ (concatenation of $i$ copies of $L$), for $i > 0$.

**Definition 6. Kleene Closure of Languages.** The Kleene Closure of $L$ is set of strings formed by taking any number of strings from $L$, possibly with repetitions and concatenating all of them.

$$L^* = \bigcup_{i \geq 0} L^i \tag{2}$$

A regular expression is a formula for representing a language in terms of "elementary" languages combined using the three operations: union, concatenation and Kleene closure.

**Definition 7. Regular Expression.** $R$ is a *regular expression* if $R$ is

1. $a$ for some $a$ in the alphabet $\Sigma$,
2. $\epsilon$
3. $\emptyset$
4. $(R_1 \cup R_2)$, where $R_1$ and $R_2$ are regular expressions
5. $(R_1 \circ R_2)$, where $R_1$ and $R_2$ are regular expressions
6. $(R_1^*)$, where $R_1$ is a regular expression

In items 1 and 2, the regular expressions $a$ and $\epsilon$ represent the languages $\{a\}$ and $\{\epsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages $R_1$ and $R_2$, or the Kleene closure of the language $R_1$, respectively.

**Definition 8. Regular Language.** A language $L \subseteq \Sigma^*$ is a *regular language* iff there is a regular expression $R$ such that $L(R) = L$.

Let $\Sigma_{KG} = \{V \cup E\}$ be the alphabet on a knowledge graph. Strings in the language of a general knowledge graph must be in the form of a triplet (node-edge-node) pattern, which we can define as a regular expression: $L_{KG} = \{(v_i e_j v_k)* \mid \text{where } e_j \text{ is a directed edge from } v_i \text{ to } v_k\}$. However, what we are interested in are regular expressions specified by the types of nodes and edges, which are semantic attributes of the node and edge objects. Below we define the language of a knowledge graph:

**Definition 9. Language of a Knowledge Graph.** Let $r_s$ be a simple semantic regular expression. The language of $r_s$:
$L(r_s) = \{t_i t_j t_k \mid t_i = \text{type}(v_i),\ t_j = \text{type}(e_j), \text{ and } t_k = \text{type}(v_k)\}$

In this phrase, the generic generate function can be used for expanding all possible texts from the user-specified regular expression pattern [11]. For example, if the user enter `Z(aY|dT)` as the regular expression pattern input, the output will be a list including `ZaY, ZdT`. After knowing all the texts that user-specified (*user's text*), we need to know what the language is in the graph (*graph's text*) in our next step.

### 3.2 Triplet-strings Generation

In this study, we use the term *triplet-strings* to refer to the concatenated strings of head-node type, edge type, and tail-node type. To see what we have in the *graph's text*, triplet-strings generation is necessary for each hop.

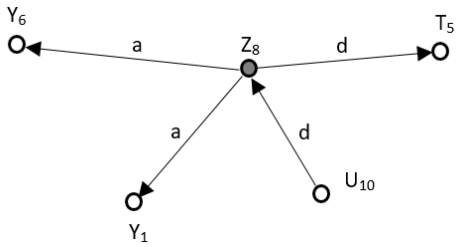



Figure 4: The current path

For example, in Figure 3 above, given $Z_8$ as the starting node, the triplet-strings from $Z_8$ to its neighbors include `Z-[a]->Y`, `Z-[a]->Y`, `Z-[d]->T`, and `Z<-[d]-U`, as shown in Figure 4.

**Algorithm 1:** tripletString($G, v_0$)

**Input:** The graph $G$, node $v_0$
**Output:** A dictionary of triplet-string from input node to all neighbor nodes, $P$

```
begin
    initialize P;
    T_0 ← type label of node v_0;
    for v ∈ v_0's neighbors do
        initialize string list s;
        for all edges from v_0 to v do
            T_E ← edge type label;
            T_1 ← node v's type label;
            if v is a successor then
                Concatenate T_0, T_E, '>', T_1 and Append to s;
            if v is a predecessor then
                Concatenate T_0, T_E, '<', T_1 and Append to s;
        P[v] ← s;
    return P.
```

Algorithm 1 demonstrates the triplet-strings generating process with a graph $G$ and a starting node $v_0$ as input. To simplify, we adopt an easier

format of the triplet-string output, such as using `Za>Y` to represent `Z-[a]->Y`. Consequently, the output of triplet-strings will be `Za>Y`, `Za>Y`, `Zd>T`, and `Z<dU` in our example.

### 3.3 Matched Paths Searching

Regular expression is often used in pattern matching, such as text pattern searching in a text document. That implies the pattern should be shorter than the document (source). However, in our problem, the source is the triplet-strings from the graph, which will be shorter than the user-specified regular expression pattern.

To tackle this problem, in our matched paths searching process, we adopt a dynamic way to match. We scan the input string, adjust the length by each hop, and conduct the string matching until the end of the input string is scanned.

Algorithm 2 demonstrates the matched paths searching process with a graph $G$, a starting node $v_0$, and a string $s$ as input. Because we have already expanded the regular expression pattern into one or more strings in our first phase, here we only need to match one string with multiple triplet-strings from the graph.

We assign *hop* to count the number of hop in the path searching, $x, y$ to indicate the position of input string $s$, $prevNodes$ and $nextNodes$ to control the path connection, $matchedStr$ and $currentStr$ to identify the language string that we have matched (confirmed) and the current string that we want to compare (uncertain).

With controlling the variables above, we, in fact, use a dynamic way to find the matched paths in the graph. In our next section, more examples will be provided.

**Algorithm 2:** findMatchedPaths($G, v, s$)

```
Input: The graph G, node v, string s
Output: A list of all matched path, M
begin
    hop ← 1; x, y ← 0; prevNodes ← ∅; nextNodes ← v;
    matchedStr ← type label of v; currentStr ← ∅;
    initialize M;
    while y < length(s) do
        y ← x + 1; regStr ← substring(s, 0, y);
        for n ∈ nextNodes do
            initialize tempN, tempStr;
            for k, tripletStr ∈ tripletString(G, v) do
                initialize path;
                Append n to path;
                currentStr ← Concatenate matchedStr and
                 tripletStr;
                if currentStr matches regStr then
                    Append k to tempN; append tripletStr to
                     tempStr; append k to path;
                if length(path) > 1 then
                    if hop == 1 then
                        Append path to M;
                    for every path i ∈ M do
                        if the end node == n then
                            Append k to M[i];
        matchedStr ← Concatenate matchedStr and tempStr;
        initialize cList;
        for i = 0; i < length(M) do
            if hop < length(M[i]) then
                Append M[i] to cList;
        prevNodes ← nextNodes; nextNodes ← tempN;
        M ← cList; hop ← hop + 1; x ← y − 1;
    return M;
```

### 3.4 Main Process

Incorporated with previous algorithms, the semantic subgraph construction process can be accomplished by the main process as shown in Algorithm 3.

The user takes the graph $G$, the starting node $v$, and the regular expression pattern $\mathcal{P}$ as the input. Regular expression expansion will generate all possible strings from user-specified pattern. We assign $N$ as am empty set to collect nodes in every path which is identified as a matched path from matched paths searching. Once the program collects all the nodes with the information from $G$, the output will be the subgraph $G_{v,\mathcal{P}}$.

**Algorithm 3:** SubgraphConstruct($G$, $v$, $\mathcal{P}$)

**Input:** The knowledge graph $G = (V, E, T_V, T_E)$, starting node $v$, regex pattern $\mathcal{P}$
**Output:** Subgraph $G_{v,\mathcal{P}}$
**begin**
  $regexSet \leftarrow$ using generic generate function to expand $\mathcal{P}$;
  initialize $N$;
  `/* Let` $N$ `be the union set of matched nodes */`
  **for** *each string* $s \in regexSet$ **do**
    $paths \leftarrow$ findMatchedPaths$(G, v, s)$;
    `/* Let` *path* `be the matched paths */`
    **for** *each path* $p \in paths$ **do**
      **for** *each node* $n \in p$ **do**
        $N \leftarrow N \cup n$;
  $G_{v,\mathcal{P}} \leftarrow$ create the subgraph object of $G$ with matched nodes set $N$;
  **return** $G_{v,\mathcal{P}}$.

For instance, if we use the knowledge graph in Figure 3 as our input graph $G$, and set $Z_8$ as the starting anchor node $v_0$. Users can specify a regular expression path $\mathcal{P}$ such as:

1. $\mathcal{P}_1 = \{Z \overrightarrow{d} Y \overrightarrow{c} X \overrightarrow{d} T\}$

   Staring from $Z_8$ in Figure 3, since $\mathcal{P}_1$ matches the paths '$Z_8 - a \rightarrow Y_6 - c \rightarrow X_7 - d \rightarrow T_2$' and '$Z_8 - a \rightarrow Y_1 - c \rightarrow X_7 - d \rightarrow T_2$', the relevant nodes that user wants should be $Z_8, Y_6, Y_1, X_7, T_2$. To union those nodes and construct a subgraph from our proposed procedure, the resulting semantic subgraph is shown in Figure 5a.

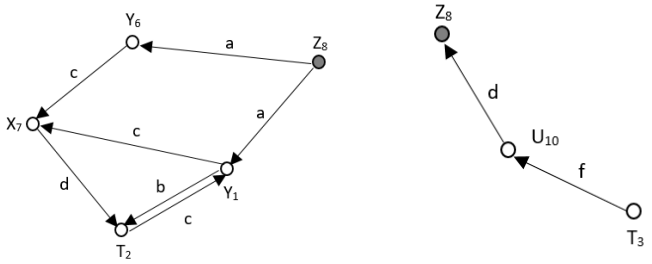
2. $\mathcal{P}_2 = \{Z \overleftarrow{d}\, U \overleftarrow{f}\, T\}$

   Staring from $Z_8$ in Figure 3, since $\mathcal{P}_2$ matches the path '$Z_8 \leftarrow d - U_{10} \leftarrow f - T_3$', the relevant nodes that user wants should be $Z_8, U_{10}, T_3$. To union those nodes and construct a subgraph from our proposed procedure, the resulting semantic subgraph is shown in Figure 5b.

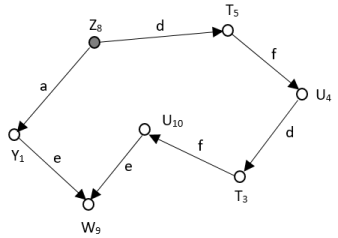
3. $\mathcal{P}_3 = \{Z(\overrightarrow{a}Y|(\overrightarrow{d}\,T\,\overrightarrow{f}\,U)+)\overrightarrow{e}W\}$

   In regular expressions, the pipe (|) is a special character that means find either the part of the pattern on the left or the right side of the pipe. A regular expression followed by a plus sign (+) matches one or more occurrences of the one-character or pattern in parentheses.

   Staring from $Z_8$ in Figure 3, since $\mathcal{P}_3$ matches the paths '$Z_8 - a \rightarrow Y_1 - e \rightarrow W_9$' and '$Z_8 - d \rightarrow T_5 - f \rightarrow U_4 - d \rightarrow T_3 - f \rightarrow U_{10} - e \rightarrow W_9$', the relevant nodes that user wants should be $Z_8, Y_1, W_9, T_5, U_4, T_3, U_{10}$. To union those nodes and construct a subgraph from our proposed procedure, the resulting semantic subgraph is shown in Figure 5c.




(a) The subgraph of using $\mathcal{P}_1$ (b) The subgraph of using $\mathcal{P}_2$




(c) The subgraph of using $\mathcal{P}_3$

Figure 5: The results of using user-defined RE paths from Figure 3

# 4 Results

In this section, we present our approach with real-world dataset and evaluate the noise removal by using graph entropy.

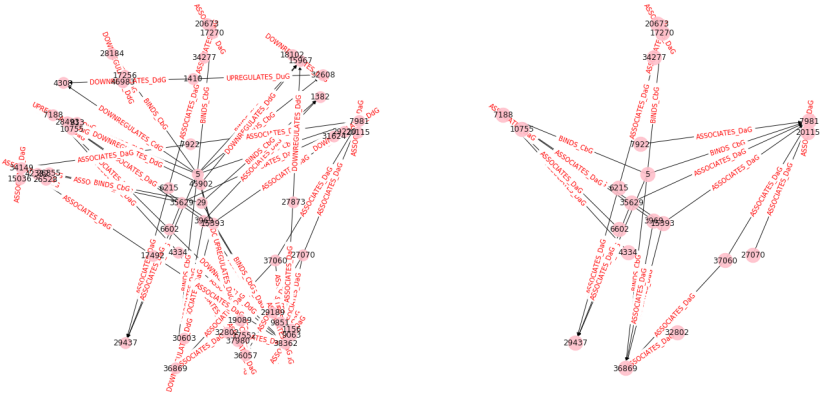
## 4.1 Real-world Dataset

We demonstrate the subgraph construction with regular expression using a biomedical network - Hetionet [5]. A subset of it is extracted and shown in Figure 6a.

**Example 1.** If we take the following inputs, after the subgraph construction process, the output semantic subgraph would be shown in Figure 6b.

Inputs:

- the knowledge graph in Figure 6a.
- the starting node,
  `ID = 5 (NAME = Flavoxate, TYPE = Compound)`.
- the regular expression,
  `CompoundBINDS_CbG>GeneASSOCIATES_DaG<Disease`.



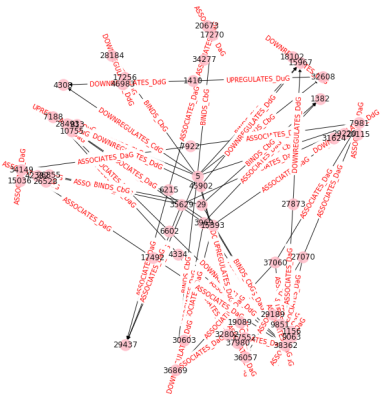
(a) The input graph (b) The output subgraph

Figure 6: The subgraph construction of example 1

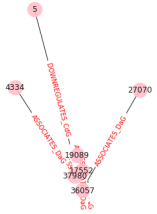
**Example 2.** If we take the following inputs, after the subgraph construction process, the output semantic subgraph would be shown in Figure 7b.

Inputs:

- the knowledge graph in Figure 7a.
- the starting node,
  `ID = 5 (NAME = Flavoxate, TYPE = Compound)`.
- the regular expression,
  `CompoundDOWNREGULATES_CdG>GeneASSOCIATES_DaG<Disease`.



(a) The input graph



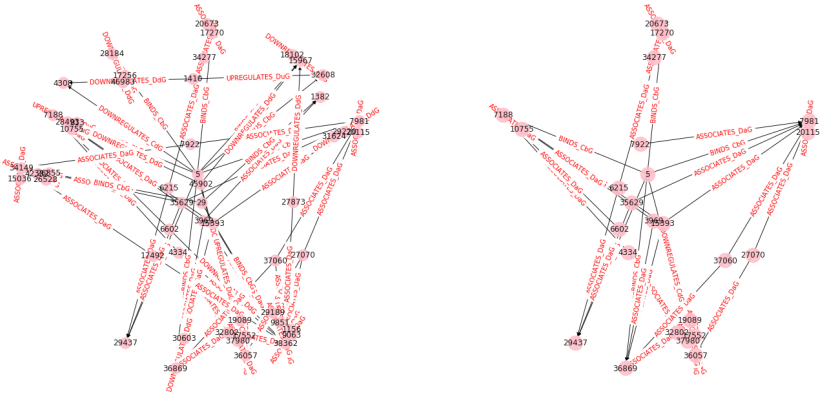
(b) The output subgraph

Figure 7: The subgraph construction of example 2

**Example 3.** If we take the following inputs, the output semantic graph would be shown in Figure 8b.

Inputs:

- the knowledge graph in Figure 8a.
- the starting node,
  `ID = 5 (NAME = Flavoxate, TYPE = Compound)`.
- the regular expression,
  `Compound(BINDS_CbG>|DOWNREGULATES_CdG>)GeneASSOCIATES_DaG<Disease`.



(a) The input graph (b) The output subgraph

Figure 8: The subgraph construction of example 3

## 4.2 Evaluation Metrics

In this study, we focus on the noise removal from a given graph. Intuitively, for example, we know that Figure 1 has more noise than Figure 2a because it contains more unwanted information to the user.

To quantify the noise, we define the noise by the difference between the amount of information from the given graph and the graph that user actually wants. We propose doing it with the entropy. The amount of information from graphs are measured by the graph entropy. In this paper, the graph entropy computation in [3, 7] was adopted. It computes the node entropy in a graph based on its topological diversity information:

$$h(i) = -\sum_{j=1}^{N} r_{ij} \log(r_{ij}) \tag{3}$$

where $r_{ij}$ is the transition probability of node $n_i$ to node $n_j$ in the graph. Note that when computing $h(i)$, we only consider the topological structure of whole graph without differentiating node types. The graph entropy of the given graph is the summation of all nodes' entropy:

**Definition 10. Graph Entropy.**

$$H_G = \sum_{i=1}^{N_G} h(i) \tag{4}$$

where $N_G$ is the total number of nodes in the given graph.

Once we have the graph entropy of the given graph and the filtered graph that user desired, the noise of a given graph to the user can be calculated as the following.

**Definition 11. Noise in the Graph.**

$$N = H_G - H_{user} \tag{5}$$

where $H_{user}$ is the graph entropy of user specified graph, and $H_G$ is the given graph.

In the examples of Section 4.1, the corresponding graph entropy and the amount of noise removal are computed and listed in Table 1.

Table 1: The Amount of Noise Removal

| **Graph** | **Graph entropy** $H_G, H_{user}$ | **Noise removal** $H_G - H_{user}$ |
|---|---|---|
| Input Graph | 5.682 | 0 |
| Example 1 | 4.072 | 1.610 |
| Example 2 | 2.461 | 3.222 |
| Example 3 | 4.392 | 1.290 |

The graph entropy of the input graph $H_G$ is 5.682. After filtering with our approach, the graph entropy of example 1 in Section 4.1 is 4.072, which reduce 1.61 amount of noise from the input graph. The graph entropy of example 2 is 2.461, which reduce 3.222 amount of noise from the input graph. We can see the significant reduction in Figure 7. Example 3 is the union of example 1 and 2. The graph entropy of example 3 is 4.392, which reduce 1.29 amount of noise from the input graph.

# 5 Extended Application on Graph Embedding

This section presents an extended application of our approach – semantic subgraph construction, incorporated with random walk based graph embedding methods [2, 4, 9].

To apply our method on graph embedding methods, the process would be composed of three modules: 1) Subgraph Construction – it generates

semantic subgraphs for each node based on a user-specified regular expression pattern to assure a meaningful random walk; 2) Random Walk – the subgraphs are further used to execute the random walk in order to produce corresponding sentences for entities; 3) Skip Gram – by inputting the sentences, it generates the latent feature vectors for each node.

Therefore, the incorporation of random walk based graph embedding and our subgraph construction method will be demonstrated as Algorithm 4.

**Algorithm 4:** Graph Embedding with Subgraph Construction

**Input:** The knowledge graph $G = (V, E, T_V, T_E)$, regex pattern $\mathcal{P}$, num. of walks per node $w$, walk length $l$, embedding dimension $d$

**Output:** The node embeddings $X \in \mathbb{R}^{|V| \times d}$

```
begin
    initialize X;
    for v ∈ V do
        G_{v_i,P} = SubgraphConstruct(G, v, P);
        walks = RandomWalk(G_{v,P}, v, w, l);
        X = SkipGram(X, walks, d);
        /* See [2,4,9] for RandomWalk and SkipGram.      */
    return X.
```

## 6 Concluding Remarks

In this report, we propose a user-oriented approach - semantic subgraph construction with regular expression to better capture the rich semantics and topology as well as remove irrelevant information in the heterogeneous knowledge graphs. Regular expression helps users to identify their desire semantic pattern, which represents filtering unwanted both nodes and edges. Several examples have shown the effectiveness of noise removal from the user-specified regular expression pattern.

After obtaining semantic subgraphs, we can therefore perform the downstream tasks, such as random walk procedure in graph embedding methods illustrated in Section 5. It allows us to capture a more meaningful walk before learning its feature matrix in a heterogeneous graph and therefore guarantee a correct inference from downstream data analytics.

## References


[1] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, 2018.

[2] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 135–144, 2017.

[3] Nathan Eagle, Michael Macy, and Rob Claxton. Network diversity and economic development. *Science*, 328(5981):1029–1031, 2010.

[4] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[5] Daniel Scott Himmelstein, Antoine Lizee, Christine Hessler, Leo Brueggeman, Sabrina L Chen, Dexter Hadley, Ari Green, Pouya Khankhanian, and Sergio E Baranzini. Systematic integration of biomedical knowledge prioritizes drugs for repurposing. *Elife*, 6:e26726, 2017.

[6] André Koschmieder and Ulf Leser. Regular path queries on large graphs. In *International Conference on Scientific and Statistical Database Management*, pages 177–194. Springer, 2012.

[7] Shenghao Liu, Bang Wang, and Minghua Xu. Serge: Successive event recommendation based on graph entropy for event-based social networks. *IEEE Access*, 6:3020–3030, 2017.

[8] Kenneth Morton, Patrick Wang, Chris Bizon, Steven Cox, James Balhoff, Yaphet Kebede, Karamarie Fecho, and Alexander Tropsha. Robokop: an abstraction layer and user interface for knowledge graphs to support question answering. *Bioinformatics*, 35(24):5382–5384, 2019.

[9] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[10] Chang Su, Jie Tong, Yongjun Zhu, Peng Cui, and Fei Wang. Network embedding in biomedical data science. *Briefings in bioinformatics*, 21(1):182–197, 2020.

[11] Adam Tauber. Exrex-regular expression string generator.

[12] Guojia Wan, Bo Du, Shirui Pan, and Jia Wu. Adaptive knowledge subgraph ensemble for robust and trustworthy knowledge graph completion. *World Wide Web*, 23(1):471–490, 2020.

[13] Hongzhi Wang, Jiabao Han, Bin Shao, and Jianzhong Li. Regular expression matching on billion-nodes graphs. *arXiv preprint arXiv:1904.11653*, 2019.