Social network de-anonymization by the Percolation Graph Matching

Ildar Nurgaliev
E-mail: *i.nurgaliev@innopolis.ru*

Abstract

Graph Matching (GM) is a fundamental part of social network analysis and it also finds applications in other domains such as in ppi-network analysis in genetic biology. However, not only the complexity of the GM itself but also the growing size of graphs thanks to its ubiquity further improves the complexity of the problem. As a result, among the many GM solutions very few, recently, are proposed to solve GM on large graphs. We focus on the new class of GM algorithms that are based on the theory of percolation. This class of algorithms shows both theoretical and practical feasibility of processing gigantic graphs. Therefore, we conduct an experimental and a formal analysis of the percolation based GM algorithms. We find that the existing solutions result to false positive matches under certain conditions that undermines the matching precision. Finally, we conduct practical experiment for the network reconciliation of VK and Instagram users in Kazan city.

1 Introduction

Graph matching (GM) is a computationally extensive task in the field of graph mining. However given the growing size of graphs, due to their ubiquity, further increase the complexity of GM. In this paper we shortly discuss on the state-of-the-art GM algorithms that are specifically designed for processing large graphs. The complexity leads from the fact that Graph Matching is a generalization of the classical graph isomorphism problem that is considered as NP-intermediate. Huge graph matching has applications in such tasks as de-anonymization social network that most of IT companies do now for improving their social network by de-anonymization other's one [9], [25]. As the most practical usage of huge graph matching is revealing functional equivalences of proteins in different species [35].

We focused on the new class of GM algorithms that are based on the theory of percolation. This class of algorithms shows both theoretical and practical feasibility of processing gigantic graphs. Therefore, we conduct an experimental and a formal analysis of the percolation based GM algorithms. As a result we want to use the algorithm in combination with named vertices in order to increase its performance for social network reconciliation task.

In the project we firstly consider a problem as follows: by using only given two graphs structures a graph-matching algorithm finds a map between the vertex sets. For this task there exist a various kind of algorithms that uses evaluation function of additional information from vertices or just its structure. As an initial point for the inexact graph matching work there were taken for an implementation 3 algorithms that in common has heuristic approach based on percolation theory of graph matching: Percolation Graph Matching (*PGM*), *ExpandOnce* [43] with *NoisySeed* and the last one and more efficient ExpandWhenStuck [21]. Afterwards we conducted experiments under 4 kinds of random graphs and else 1 experiment on real graph in order to find relation between degree distribution and efficiency of these algorithms. The main algorithm that is observed is *ExpandWhenStuck* - the distinguishing feature of this algorithm is that it requires a dramatically smaller number of seeds (sometimes just one) and it is more stable than the other heuristic algorithms analyzed. Under these experiments there were revealed weak points and clear points that are need to be elaborated for increasing quality performance of inexact Graph Matching algorithms based on percolation theory. As the main part of the project we try to reveal from 2 real social networks some set of correct vertex matches that represent a user in both network.

2 Related work

Revealing the structural similarity of graphs is commonly referred to as graph matching [24]. A large variety of methods addressing specific problems of structural matching have been proposed [12]. Graph matching algorithms could roughly be separated into systems matching structure in an exact manner and error-tolerant way. Most known approaches are listed below:

- **Edit distance**: this approach is based on intuitive dissimilarity measures on graphs by suitable edit costs. Exact approach [27], [38], inexact [7], inexact with an upper bound of the exact edit distance [31].
- **Computing maximum common subgraph**: based on maximum clique problem in an associated graph. Exact approach [8], [14], [23], [40], inexact [6].
- **Learning probabilistic edit costs**: the idea is to model the distribution of edit operations in the label space [2], [18], [37].
- **Granular computing paradigm**: based on general graph classification system [16].
- **Density based approach**: identifies the most central patterns from the dense regions in both matching graphs [15].

- **Exploit graph kernel**: extract attributes from kernels that will be used for similarity measure between nodes of non-attributed graphs. Next, such attributes are embedded in a graph-matching cost function, through a probabilistic framework [28].

The most scalable graph–matching approaches use ideas from percolation theory, where a matched node pair "infects" neighboring pairs as additional potential matches. This class of matching algorithm requires an initial seed set of known matches to start the percolation. The first algorithm that was implemented is Percolation Graph Matching (*PGM*). It is not such as powerful otherwise it generates only correct matching seeds. In this implementation *PGM* algorithm was slightly modified: the threshold r was reduced from 4 to 2. That leads to significantly more matches (wrong matches also accepted). In *PGM* algorithms, initial seeds play an important role because it does not generate them as the other two algorithms. Under conducted experiment it is clear that *PGM* has very good performance on scale free networks in case of reduced threshold, the analysis and the proof is explained in [26].

The next algorithm is *ExpandOnce* matching algorithm, which performs one round of expansion of the initial seed set. This helps the percolation process overcome the bottleneck due to a small seed set size. Afterwards it executes *NoisySeed* algorithm that does percolation in a graph. The main part that we should know about this algorithm is that it takes candidate seed for percolation randomly without any strategy and smart heuristic. That's why this algorithm could be considered as not efficient one.

The last one and more efficient algorithm is *ExpandWhenStuck* that is based on the robustness ideas developed [19]: means that it dynamically generates huge number of seeds for percolation with no big affection of wrong matching. Whenever there are no further pairs with score at least two, we add all the unused and unmatched neighboring pairs of all the matched pairs to the candidate pairs. In comparison with *ExpandOnce*, this algorithm has better performance for both real and random graphs, and its computational complexity is lower. Else on significant difference with respect to *ExpandOnce* is that *ExpandWhenStuck* uses smart choice of candidate seed for percolation: namely saying it takes a seed with max score and min difference in vertex degree of that seed.

2.1 Repeating experiments on prior work

For experiments there were generated 4 random graphs (Erdős–Rényi, Watts-Strogatz, Barabási–Albert and Chung-Lu) and was chosen one real graph (Face-book graph). In order to observe behavior of the algorithms on random graphs, power-law graph and high clustered graph. The algorithms were written in C++ language with usage just *stl* library. All the experiments were conducted on the server with 125 GB RAM and processor Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz. For experiment we take one graph and do matching with itself with given some initial correct matches whose count depends on the algorithm.

3 Algorithms

The algorithms are based on percolation graph theory that considers **structural similarity** to be the **most** important feature in the graph–matching process.

3.1 Notation

The Graph G is represented as V - vertex set and E - edge set. Given two graphs: $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ the problem is formally defined as: some pair $[i, j] \in V_1 \times V_2$ corresponds to some **unique** entity (ex: *person*). Find corresponding vertex pairs $[i, j] \subseteq V_1 \times V_2$ based on the topologies of the 2 networks. Having just a structure of graphs (two graph are unlabeled) and both $V_{1,2}$ and $E_{1,2}$ could be not exactly the same. Algorithm works just with **pairs** where $[i', j'] \in V_1 \times V_2$ is a neighbor of $[i, j]$ if $(i, i') \in E_1 \wedge (j, j') \in E_2$. The matching algorithms in relate work section refer to a pair $[i, j]$ spreading out marks by adding one mark to each neighboring pair of $[i, j]$ where the score of a pair is defined as the number of marks it obtained during percolation process. *Activated pair* is already matched pair. The output of algorithms presented in this section is a set of identity pairs S. Let $\wedge(S)$ denote the number of correct pairs in the set S, where the number of wrong pairs is represented by $\Psi(S)$. Also, $V_1(S)$ is the set of vertices from graph G_1 in output, formally $V_1(S) = i, \exists j : [i, j] \in S$ and the same with $V_2(S)$.

The main algorithms notions that are based on graph percolation:

- Matched node pair "infects" neighboring pairs as additional potential matches (candidate pair);
- require an initial seed set of **known matches** to start the percolation.

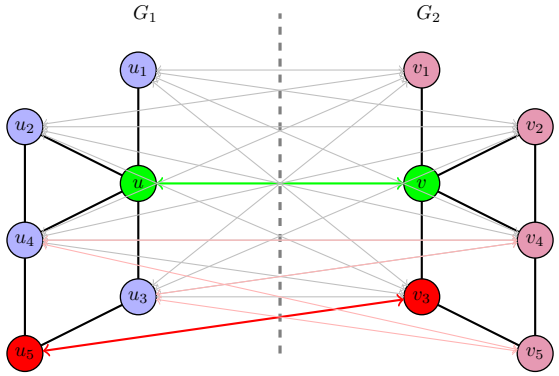

Figure 1: Nodes u_1, u_2, u_3 and u_4 are unmatched neighbors of matched node u . Nodes u_4 and u_3 are unmatched neighbors of incorrectly matched u_5. The same is with G_2. After new seeds generated from the neighbors nodes and their marks spreading we got $[u_4, v_4]$ and $[u_4, v_3]$ candidates for matching, as a result $[u_4, v_4]$ will be correctly matched having more marks and $|d_{1,u} - d_{2,v}| = 0$.

3.2 ExpandWhenStuck algorithm

A main feature of *ExpandWhenStuck* is to expand the seed set by many noisy candidate pairs whenever there are no other

unused matched pairs. More precisely, whenever there are no further pairs with score at least two, the candidate set is extended by unused and unmatched neighboring pairs of all the matched pairs that make wild percolation process. Among these candidate pairs, where a small fraction is correct and most of them are wrong, (i) correct pairs help us to continue the percolation process and match remaining unmatched pairs, and (ii) wrong pairs would have a negligible effect [21] as it is shown on Figure 1. Additional and the most effective modification made is top candidate structure that is organized in queue: at every iteration there is just one candidate is chosen with the highest score among such pairs afterwards the candidate is added to the matched set; also, each node is matched at most once. We then proceed with spreading out the marks from this new matching. Moreover mostly there are several top pairs in candidate queue. Among all such candidate pairs $[i, j]$, the pair that minimizes the difference in the degrees of nodes $|d_{1,i} - d_{2,j}|$.

1 ExpandWhenStuck

Input: $G_1(V_1, E_1), G_2(V_2, E_2)$, $\mathcal{A}_0$ ▷ seed set of correct pairs
Output: $\mathcal{M}$ ▷ The set of matched pairs

$\mathcal{A} \leftarrow \mathcal{A}_0$; ▷ the initial set of seed pairs, $\mathcal{M} \leftarrow \mathcal{A}_0$
$\mathcal{Z} \leftarrow \emptyset$; ▷ the set of used pairs
while $|\mathcal{A}| > 0$ **do**
  for all pairs $[i, j] \in \mathcal{A}$ **do**
    add the pair $[i, j]$ to $\mathcal{Z}$;
    add one mark to all of $[i, j]$ś neighboring pairs;
  while exists an unmatched pair with $score \geq 2$ **do**
    among the pairs with the highest score
      select the unmatched pair $[i, j]$
        with the **minimum** $|d_{1,i} - d_{2,j}|$;
    add [i,j] to the set $\mathcal{M}$;
    if $[i, j] \notin \mathcal{Z}$ **then**
      add one mark to all of $[i, j]$ neighboring pairs;
      add the pair [i,j] to $\mathcal{Z}$;
  $\mathcal{A} \leftarrow$ all neighboring pairs $[i, j]$ of matched pairs $\mathcal{M}$ s.t. $[i, j] \notin \mathcal{Z} \ \wedge \ i \notin V_1(\mathcal{M}) \ \wedge \ j \notin V_2(\mathcal{M})$;
return $\mathcal{M}$

3.3 Bad cases for percolation based algorithms

Under conducted experiment over Facebook graph **??**, we mentioned that the graph has a lot of standalone vertices. Vertices with degree equal to one (leafs) or two (bridges), that leads to stable not perfect result of *ExpandWhenStuck* Figure 2. As a result *ExpandWhenStuck* could have bad performance in cases of bridge occurrence in a graph matching as presented in a figure Figure 3. Imagine two clusters in a graph connected with a bridge (just one vertex as bridge). ExpandWhenStuck can't deal with it thus having twice bad result. The problem is located in established percolation threshold $r = 2$ that does not consider these situations.

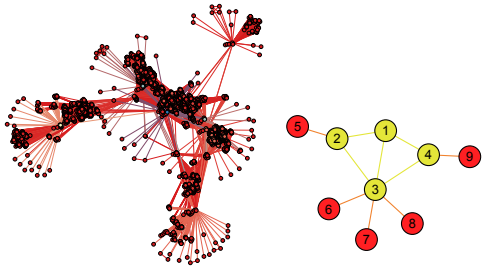

Figure 2: Face-book circles graph ranked by degree. Contains a lot of sparse nodes, that includes some leafs that are not matched.

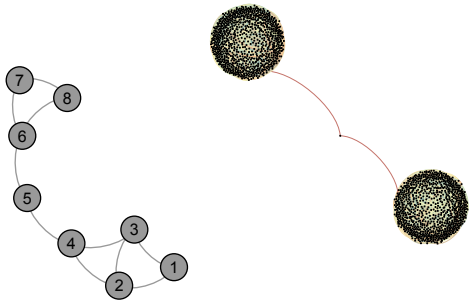

Figure 3: A cluster possibly will not be reached, that makes ExpandWhenStack score twice bad.

Furthermore a percolation based algorithms have some cases of unsupported decision of candidate matching that's example is presented in Figure 4. After marks spread operation we could have situation when in TOP pairs selection with the highest score and the same degree difference between vertices in a pairs we get a set of candidates among which impossible choose just one matching because of overlapping candidates as $[i, j], [i, j'], [i, j'']$. We consider this situation as unreasonable matching. In this case the decision of *ExpandWhenStuck* will be done definitely at random manner. Having just the structure of a graph we definitely cannot avoid these situations thus it would be convenient if an algorithm could aware a user about these unsure matching and present all the pairs associated as one automorphism set or super-vertex.

The last weak point to mention is the performance of *ExpandWenStuck* in cases of large egocentric networks. Over experiments conducted on Chung-Lu model **??** rather for not huge graph of 100000 vertices the time consumption was very high. The distinctive point of Chung-Lu model generator is huge ego centric concentration that in fact generates huge number of edges from just some thousands number of vertices in *ExpandWhenStuck* implementation. In fact we established just $V = 100\ 000$ and got $E = 3\ 585\ 863$ with $deg_{avg} = 72$. As a result *ExpandWhenStuck* performance sig-

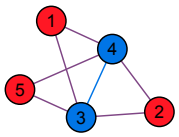

Figure 4: Unpredictable case: 1 and 5 vertex spread their marks, thus [3,3'] [3,4'] [4,3'] [4,4'] have the same score and the same probability t be selected because the degree of vertex 3 is the same to vertex 4. Under conducted experiment accuracy = 3/5.

nificantly degraded while candidate matching process because the searching space has direct relation on deg_{avg} and on count of E: the search space is enriched with candidates by all the combinations of vertex's neighbors, thus we never could conduct experiment over Chung-Lu model with $V = 10^6$.

4 Matching by percolation process

The most scalable graph–matching approaches use ideas from percolation theory, where a matched node pair "infects" neighboring pairs as additional potential matches. There is a various implementations of this approaches with and without restrictions. The main algorithms notions are based on graph percolation: (i) matched node pair "infects" neighboring pairs as additional potential matches; (ii) require an initial seed set of **known matches** to start the percolation.

Our algorithm is based on the next ideas of *ExpandWhenStuck* as: (i) reduced phase transition by taking in consider the degree of vertices in a matching pair, (ii) *ExpandWhenStuck* correctly matches almost all the nodes which are in the intersection of the two graphs, and (iii) the main feature and contribution of *ExpandWhenStuck* is the proved idea about negligible affection of many wrong pairs in initial correct seed set $\mathcal{A}_0$ and while generating new candidates thus having enormous set of incorrect pairs accelerating percolation process up to the end. As a result the smart implementation of percolation based graph matching algorithm is robust to wrong pairs, it succeeds with high probability and the best performance with respect to other approaches possible for graph matching problem solution over huge graphs.

For the baseline of our ideas described before, we will firstly explain bootstrap percolation process for graph matching with reduced threshold afterwards we will show main theorems and the citations of robustness confidence on $G(n,p)$ random graph model, that are easily and well elaborated with different degree distribution graphs by the *ExpandWhenStuck* heuristic.

4.1 Graph percolation process

The Bootstrap percolation is the process of node activation on a random graph $G(n,p)$ [19]. Initially we are given a set $A(0)$ ($|A(0)| = a_0$) of active nodes and a threshold $r \geq 2$. This is the lowest threshold possible in graph percolation process and in the same time it is efficient in $G(n,p)$ model as it was proved in [21]. At each time step the infections from one vertex only. A node is activated at time step τ if at least r of its neighbors were activated and used in the previous τ time steps. Let $A(\tau)$ and $Z(\tau)$ denote the set of active and used nodes at time step τ. Initially $Z(0) = \emptyset$. At each time step $\tau \geq 1$, we choose a node u_τ from $A(\tau-1) \setminus Z(\tau-1)$ and give each one of its neighbors a mark. We call u_τ a used node and update $Z(\tau) = Z(\tau-1) \cup u_\tau$. As an example choose $u_1 \in \mathcal{A}(0)$ and give each of its neighbors a *mark*; we then say that u_1 is *used*, and let $\mathcal{Z}(1) := u_1$ be the set of used vertices at time 1. We continue recursively: at time t, choose a vertex $u_t \in \mathcal{A}(t-1) \setminus \mathcal{Z}(t-1)$. We give each neighbor of u_t a new mark. The bootstrap percolation process stops when $A(\tau) \setminus Z(\tau) = \emptyset$. The main point in bootstrap percolation is the phase transition threshold that is found for random graph $G(n,p)$.

4.2 Baseline theorem of percolation graph

As it was proved in [43], for having just correct matches in resulting output of percolation process over two exactly the same graphs ($t = 1, s = 1$) it is sufficient to have phase transition threshold $r = 4$ and initial seed set $\mathcal{A}_0$ should be at least $a_{1,1,4}$. Parameter $a_{t,s,r}$ defines phase transition for the percolation process [19], [43] in other words it defines number of initial seed for passing phase transition for full percolation.

- $a_{t,s,r} = (1 - \frac{1}{r}) b_{t,s,r}$ (minimal size of initial seed)
  - where $b_{t,s,r} = \left[\frac{(r-1)!}{nt^2(ps^2)^r}\right]^{\frac{1}{r-1}}$
- for $\Psi(A_0) = 0$: the size of seed set should be at least $a_{1,s,4}(r = 4)$ (r - threshold).

For this special case ($r = 4$), in **Theorem of robustness of NoisySeeds** [21] guarantees that a seed set of size $a_{1,s,2}(r = 2)$ is enough for matching almost all the nodes correctly with a vanishing fraction of errors having that the ratio $\frac{a_{1,s,4}}{a_{1,s,2}} \to \infty$.

The proof is structured on findings of lower and upper bounds of error fraction while percolation processes. As a result in [21] they came out with the next lemma about upper bound of resulting mistakes count that is the same in initial seed generation thus mentioning that the percolation works on $G(n,p)$ almost clearly:

Lemma 4.1: With high probability:

$$\Psi(\mathcal{M}_\mathcal{T}) = o(a_{1,1,4}) \text{ and } \Psi(\mathcal{M}_{\mathcal{T}_{init}}) = o(a_{1,1,4})$$

5 Social network reconciliation

The final experiment is the real application of graph matching over social networks in order to help advertisement companies or sociologist study. As the main algorithm for social graphs matching we have chosen *ExpandWhenStuck* that is having the best performance over Percolation based solutions

for anonymized graphs. We should notice that in many real life graphs, the edges and nodes also contain various attributes, that we could use as additional features with respect to structure of a graph, that depend of an application of graph matching techniques on a particular field. That in our case, nodes in social graphs have the first and second name. Thus we could use this for reducing the search space while candidate pairs generation process. For the resulting experiment we decided to do graph matching over named graph (first name, last name) such information is useful in order to reduce the bottlenecks and it has free access.

Experiment steps.

1) Crawl Kazan city peoples from Vkontakte social network;
2) Crawl Kazan city peoples from Instagram social image share network;
3) modify the *ExpandWhenStuck* algorithm for the named graph and candidate set generation with respect to named features of vertices;
4) conduct experiment over a little graph of 20000 vertices;
5) conduct over the whole retrieved graphs and reveal the efficiency;

In order to have automated test of algorithms's performance over the real graph collected from Instagram and VK in previous section, we should conduct test just over labeled subgraphs with instagram usernames. After retrieving users from VK social graph we got 30000 labeled users, such subgraph consist of 30 components. The Instgram was sliced by the usernames found in VK. As a result we could test the resulting matched pairs by their user names.

6 Vkontakte network data collection

Vkontakte is the largest European on-line social media and social networking service. It is available in several languages and is especially popular among Russian-speaking users. For the experiment purpose we plan to de-anonymize Kazan city peoples from both networks: VK and Instagram. The API of VK is well structured and actually have not big limitations. In order to increase speed of the crawler we used such feature as VK-script which is executed on server side of VK, thus we could replace our 21 request for collecting information to one request as the VK-script could execute 21 request in maximum on server side, moreover we made 3 user tokens for one the application (Crawler) by tree different users and conducted parallel crawling by*tmux* process daemon creation and python script. Moreover the data was collected to PostgreSQL database tables as follows in Table I that also facilitated the multi-threaded version of crawler by transactions.

personal	**slinks**	**friend_links**	**uids**
id	uid	lid	id
name	fb	rid	is_proceed
sname	telephone	f:valid_edge($lid < rid$)	
bdate	Instagram	f:unique_edge(lid, rid)	
sex			

Table I: Database schema for data crawled from VK.

The crawling process was aimed on crawling just users who has relation to Kazan city. We started from 3 social groups listed below as it has most of the Kazan peoples. After that we proceed every user and their friends in one round.

- https://vk.com/int_kazan 107840 users
- https://vk.com/kazan.club 85479 users
- https://vk.com/kzngot 51993 users

As a result we got 393 386 user profiles and 204 489 813 friend links in total. After filtering users without Instagram public user name in 'slinks' table we got just 24800 users with open Instagram profile and 463 468 links. This was done for the automated test explained in **??**.

Final Vk graph has 40 components and most of the nodes are in one component (99.68 %), while the other components have less than 0.01 %.

- average degree = 38
- Connected components = 40
- Graph density = 0.002
- $|V| = 24800$
- $|E| = 463468$

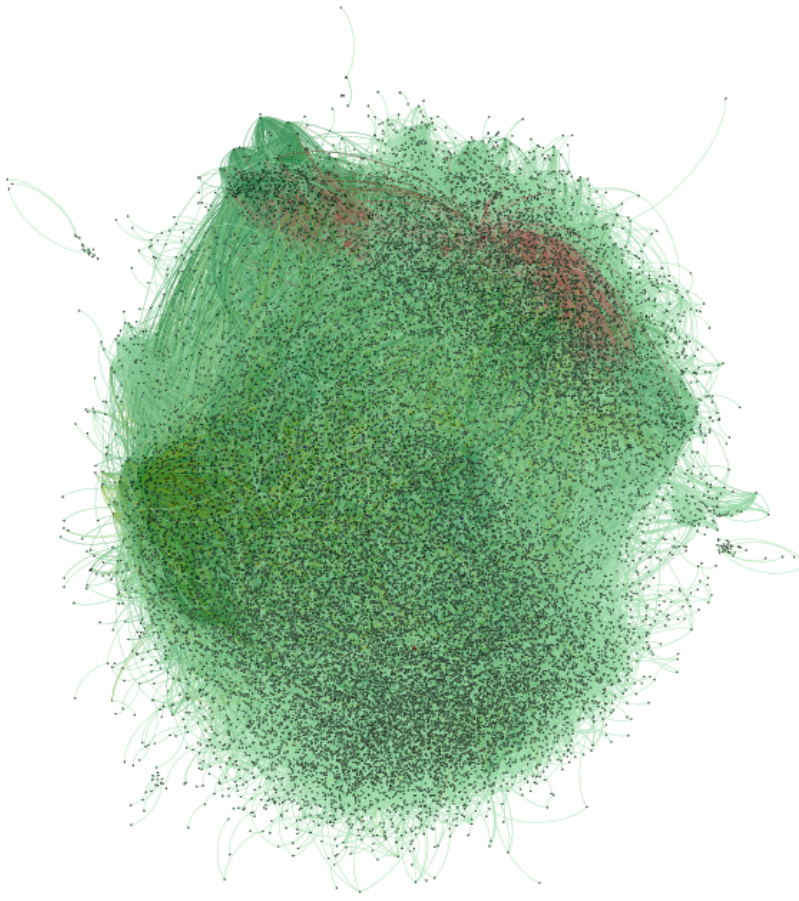

Figure 5: VK social network limited to users with public instagram user-name

The VK API is very open, that we could retrieve some additional information of a user (with public profile) telephone number, Facebook and Instagram page that we collected in

database Table I. The interesting part is that we could use such open information as a seed vertices for initial seed set for Instagram or Facebook network matching with Vk network. Moreover the telephone number could be used for future to extend user information by crawling Avito https://www.avito.ru/ advertisement that in every advert has an associated telephone number.

7 Instagram network data collection

For the experiment we firstly assumed to crawl Facebook users but with the forced migration of Facebook API to Graph API v2.0, the friends' data API was finally closed by provider. As a result it reduced app virality to protect users privacy. That's why we could not use Facebook for the experiment.

The next widely used social network in Kazan city is the Instagram, but the API was limited also in 2016 year after Facebook has bought this service. Thus we started to design a solution in order to avoid these bottlenecks of API. The solution for the Instagram was to implement simulation of Android Instagram application as a standalone crawler app that would maintain cookies, session, encryption of https communication by the Instagram application, a open key of the application from Android market. As result we made an imitation through Python scripting the Android application and also we used 6 pseudo accounts to crawl data from Instagram. Every user after 100 request was blocked for a minute thus the other pseudo users could continue crawling. The solution is good and quite fast moreover the Android app simulation get the data in JSON format that is very convenient data format to get. Moreover the data was collected to PostgreSQL database tables as follows in Table II that also facilitated the multi-threaded version of crawler by transactions.

personal	**fwlink**	**uids**
id	lid	pk
fname	rid	is_proceed
uname	f:valid_edge($lid < rid$)	is_private
	f:unique_edge(lid, rid)	

Table II: Database schema for data crawled from Instagram.

As initial users for crawler expansion process we took vk users with open Instagram user-name on their profile pages (24800) section 6 and also we crawled their friends. The Instagram have two abstractions for the connection links as a) "Follower" and b) "Following", we decided to retrieve just "Following" links representing them as undirected link in the retrieved social graph and expecting that most of them would be the same as in vk friend links.

As a result we got 4 089 683 user profiles in total. After filtering users with vk users with associated instagram names we got just 20794 users and 240414 links. This was done for the automated test explained in **??**.

Instagram graph has 36 components and most of the nodes are in one component (99.65 %), while the others have less than 0.01 %.

- average degree = 23
- Connected components = 36
- Graph density = 0.001
- $|V| = 20794$
- $|E| = 240414$

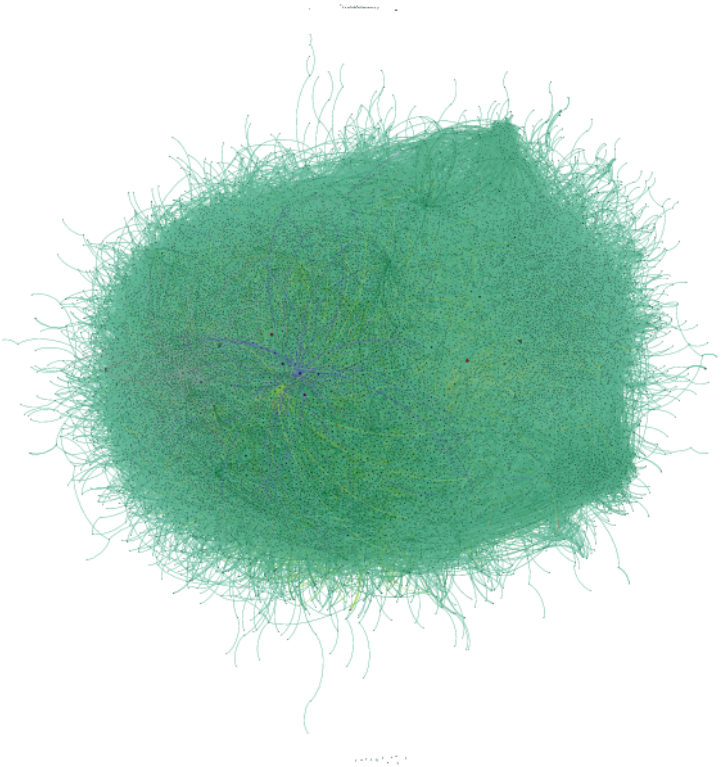

Figure 6: Instagram social network limited to associated VK profiles

8 Experimental results

The experiment was conducted over two associated social graphs (VK, Instagram) Figure 5 and Figure 6. These graph has homophily property that is every user have relation to Kazan city, that's why both of these graphs have one big component with 99% of nodes in it. The degree distribution is represented in Figure 7, both of them are Scale-free as it was expected.

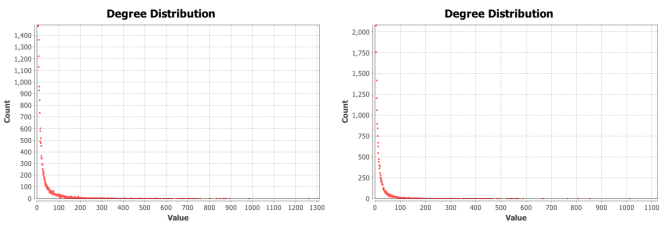

Figure 7: Degree distribution, on the left side VK, one the right side Instagram

We use *ExpandWhenStuck* with anonymized graph expecting that it would match some of the nodes just only over

the assumption of the graph structure. The graph analysis in previous two sections show that the structure of these graphs are different in every metric, that's why we show there the result of updated algorithm that would use the "full name" property of every neighbor vertex in order to assume potential matching that is expected to be very useful for social network reconciliation task.

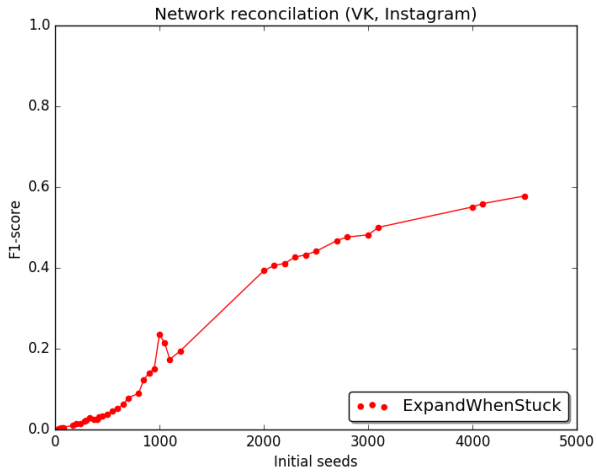

Figure 8: Experiment over anonymized graphs using just graph structure

8.1 Updated algorithm experiment

As we see Figure 8, the results of ExpandWhenStuck over such networks has bad results and even can not fully percolate. Robustness principle permits for the algorithm to be wild in matching nodes while the application of social network reconciliation is a task of subgraph matching rather the graph isomorphism, that's why in order to maintain the robustness of ExpandWhenStuck we filtered the candidate generation process with the full name similarity function and a threshold = 0.61. The simplest way to compare two strings (full names of users) is with a measurement of edit distance. As a result we got better performance in Precision and Recall metrics Figure 9. Practically the updated part will be on the 6th line of the algorithm algorithm 1: where we add else one predicate with name similarity > 0.61.

9 Conclusions and future work

In the report we have presented an application of robustness ideas of *ExpandWhenStuck* and with modifications that allows to use it over named graphs. Robustness principle permits for the algorithm to be wild in matching nodes while the application of social network reconciliation is a task of subgraph matching rather the graph isomorphism, that's why in order to maintain the robustness of ExpandWhenStuck we filtered the candidate generation process with the full name similarity threshold. As a result we got better performance in Precision and Recall metrics.

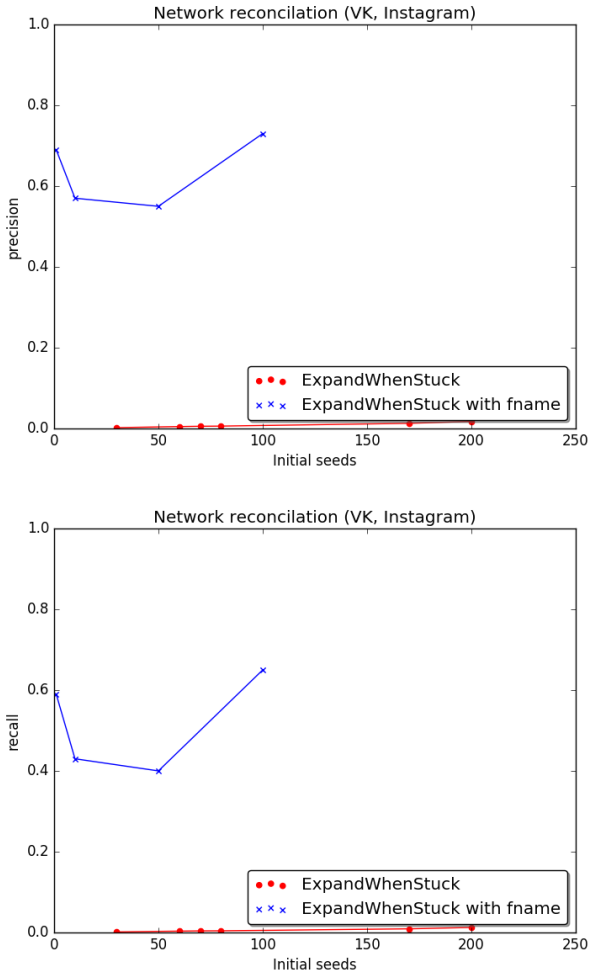

Figure 9: Experiment over named graph that allows us to use first and second name, there we use filtration of candidate pairs by the name similarity function

For future work we could implement more stable version of graph matching with respect to revealed limitations, presented in the report.

References

[1] Compressing the graph structure of the Web. *Proceedings DCC 2001. Data Compression Conference*, pages 1–10, 2001.

[2] N. M. Laird A. P. Dempster and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[3] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Data Compression*, page 203–212, 2001.

[4] Dalal Al-azizy, David Millard, Iraklis Symeonidis, and Nigel Shadbolt. Risks and Security of Internet and Systems. 9572:36–51, 2016.

[5] A R Barabási A-L. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[6] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18:689–694, 1997.

[7] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1:245–253, 1983.

[8] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3):255–259, 1998.

[9] Emilio Leonardi Carla-Fabiana Chiasserini, Michele Garetto. Social network de-anonymization under scale-free user relations. *Journal*, pages 1–14, April 2016.

[10] D Chakrabarti. AutoPart: Parameter-free graph partitioning and outlier detection. *Knowledge Discovery in Databases: Pkdd 2004, Proceedings*, 3202(22):112–124, 2004.

[11] F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of Combinatorics*, 6(2):125–145, 2002.

[12] Lawarence B. Holder Diane J. Cook. *Mining graph data*. WILEY, 2007.

[13] P. Erdős and A. Rényi. On random graphs i. *Publ. Math. Debrecen*, page 6:290–297, 1959.

[14] M.-L. Fernandez and G. Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letters*, 22(6–7):753–758, 2001.

[15] Antonello Rizzi Filippo Maria Bianchi, Lorenzo Livi. Two density-based k-means initialization algorithms for nonmetric data clustering. *Pattern Analysis and Applications*, pages 1–19, January 2015.

[16] Antonello Rizzi Alireza Sadeghian Filippo Maria Bianchi, Lorenzo Livi. A granular computing approach to the design of optimized graph classification systems. *Methodologies And Application*, 18:393–412, February 2014.

[17] M. R. Garey and D. S. Johnson. Computers and intractability: a guide to the theory of np-completeness. 1990.

[18] M. Petrou J. Kittler and M. Nixon, editors. *A probabilistic approach to learning costs for graph edit distance*, number 3, United Kingdom, 2004. Cambridge. 17th International Conference.

[19] Svante Janson, Tomasz Luczak, Tatyana Turova, and Thomas Vallier. Bootstrap percolation on the random graph Gn,P. *Annals of Applied Probability*, pages 1989–2047, 2012.

[20] Ehsan Kazemi and Matthias Grossglauser. On the Structure and Efficient Computation of IsoRank Node Similarities. page 8, 2016.

[21] Ehsan Kazemi, S Hamed Hassani, and Matthias Grossglauser. Growing a Graph Matching from a Handful of Seeds. *Vldb 2015*, (ii):1010–1021.

[22] J. Leskovec. Stanford network analysis project. http://snap.stanford.edu/index.html.

[23] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9:341–354, 1972.

[24] Antonello Rizzi Lorenzo Livi. The graph matching problem. *Pattern Analysis and Applications*, 16:253–283, 2013.

[25] Giulio Mangano and Alberto De Marco. Social Network Deanonymization Under Scale-free User Relations. *ACM, IEEE*, 2016.

[26] Giulio Mangano and Alberto De Marco. Social Network Deanonymization Under Scale-free User Relations. *ACM, IEEE*, 2016.

[27] B. T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomor- phism detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):493–504, 1998.

[28] Francisco Escolano Miguel Angel Lozano. Graph matching and clustering using kernel attributes. *Neurocomputing*, 113:177–194, August 2013.

[29] Saket Navlakha and Carl Kingsford. The power of protein interaction networks for associating genes with diseases. *Bioinformatics*, 26(8):1057–1063, 2010.

[30] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 419, 2008.

[31] M. Neuhaus and H. Bunke, editors. *An error-tolerant approximate matching algorithm for attributed planar graphs and its application to fingerprint classification*, number 3138, Berlin, 2004. Proceedings of 10th International Workshop on Structural and Syntactic Pattern Recognition.

[32] Steven Noel and Sushil Jajodia. Managing attack graph complexity through visual hierarchical aggregation. *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security - VizSEC/DMSEC '04*, page 109, 2004.

[33] P. Pedarsani and M. Grossglauser. On the privacy of anonymized networks. *SIGKDD*, August 2011.

[34] D. M. Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[35] J. Xu R. Singh and B. Berger. Global alignment of multiple protein interaction networks with application to functional orthology detection. *Proceedings of the National Academy of Sciences*, 105:12763–12768, 2008.

[36] Sriram Raghavan Sriram Raghavan and H. Garcia-Molina. Representing Web graphs. *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 405–416, 2003.

[37] R. A. Redner and H. F. Walker. Mixture densities, maximum likelihood and the em algorithm. *Society for Industrial and Applied Mathematics (SIAM)*, page 195–239, 1984. review 26.

[38] A. Sanfeliu and K. S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics (Part B)*, 13(3):353–363, 1983.

[39] Appu Shaji, Aydin Varol, Lorenzo Torresani, and Pascal Fua. Simultaneous point matching and 3D deformable surface reconstruction. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1221–1228, 2010.

[40] M. Kraetzl W. D. Wallis, P. Shoubridge and D. Ray. Graph distances using graph union. *Pattern Recognition Letters*, 22(6):701–704, 2001.

[41] S SH Watts DJ. Collective dynamics of 'small-world' networks. *Nature*, pages 440–442, 1998.

[42] Hongtao Xie, Ke Gao, Yongdong Zhang, Jintao Li, and Huamin Ren. Common visual pattern discovery via graph matching. *Acm Mm*, pages 1385–1388, 2011.

[43] Lyudmila Yartseva and Matthias Grossglauser. On the performance of percolation graph matching. *Proceedings of the first ACM conference on Online social networks - COSN '13*, pages 119–130, 2013.