Study and possible contribution to Graph Matching from handful of Seeds

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

1 Coclusion on related work

1.1 Limitations and survey of related works

NEW, percolation based algorithms performance overview

The limitations become more clear after analyzing algorithm performance on real graph than on synthetic ones, thus having more performance analysis on real graph is a good approach for performance evaluation.

These algorithms have some weak points, the most obvious one was found by Facebook graph matching **??** the recall of ExpandWhenStuck can not reach 100% at any given number initial seeds, there was revealed an impossibility to match a leave with one edge because of unaccepted two marks from at least two different edges Figure 1. It could be avoided by reusing matched seed as candidate seed. For example every vertex that has a unique neighbor leaf would have special marking that it could be reused while being in matched set.

The worst case is when between two clusters is just one vertex that connects them just by two edges Figure 2. In case of initial seeds in one cluster the second cluster would not be achieved and matched because of unmatched bridge vertex Figure 2. In this case ExpandOnce has benefit with respect to ExpandWhenStuck and PGM, ExpandOnce firstly generates noisy seed by noisy seeds not through matched pairs as in ExpandWhenStuck. The resulting break-throughly received noisy seeds could percolate both clusters.

The given Theorem 1 (Robustness of NoisySeeds) in [5] is applicable, but it is very rare when we could guaranty good expansion of NoisySeed algorithm because we always unsatisfy condition given in this theorem as the graphs from conducted experiments do not satisfy the condition of the theorem. That's why the new algorithm for solving this task should use percolation theory with not random strategy for matching candidate seed, as it was done with success in ExpandWhenStuck in order to get a little bit more guarantee for good percolation.

The next weak point is a case of statistically same probability of a vertex for being chosen as a candidate seed, examples are presented in Figure 3 that could propagate huge exception in percolation of a graph.

ExpandWhenStuck has good **Time Execution** with respect to other two algorithms because of trade-off with space, ExpandWhenStuck generates a lot of candidates in every iteration. In case of big dense and deg_{avg} of a graph the space complexity will become huge as it was with Chung-Lu random graph.

2 Matching by percolation process

CHANGED, description of basic behavior of percolation based algorithms

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.

2.1 Graph percolation process

COPY, bootstrap percolation process over a graph

The Bootstrap percolation is the process of node activation on a random graph $G(n,p)$ [4]. 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 [5]. 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 $G(n,p)$.

2.2 Baseline theorem of percolation graph

COPY, listing of theorem for robustness to wrong seeds

As it was proved in [10], 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 [4], [10] 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** [5] 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 [5] 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 2.1: With high probability:

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

3 Contribution

NEW, contribution explanation

In this section we introduce new algorithm *ExpandTopStack* algorithm 1 based on the robustness ideas developed in the *ExpandWhenStack* [5]. Among candidate pairs generated in *ExpandWhenStack*, a small fraction is correct and most of them are wrong, (i) correct pairs help the algorithm to continue the percolation process and match remaining unmatched pairs, and (ii) wrong pairs would have a negligible effect, our contribution is to hide this negligible effect by super vertex alias over unpredictable pairs and increase performance by top-k selection for matching process nonetheless we found cases when percolation is impossible while having percolation threshold equal to 2, namely saying the cases of leafs and clusters connected through a bridge are considered as unmatched in *ExpandWhenStuck* while overall its performance is perfect.

3.1 Bad cases for percolation based algorithms

NEW, list of weak points of related works

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 1. As a result *ExpandWhenStuck* could have bad performance in cases of bridge occurrence in a graph matching as presented in a figure Figure 2. 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.

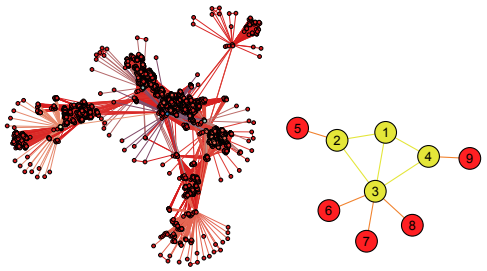

Figure 1: Face-book circles graph ranked by degree. Contains a lot of sparse nodes, that includes some leafs that are not matched.

Furthermore a percolation based algorithms have some cases of unsupported decision of candidate matching that's example is presented in Figure 3. 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

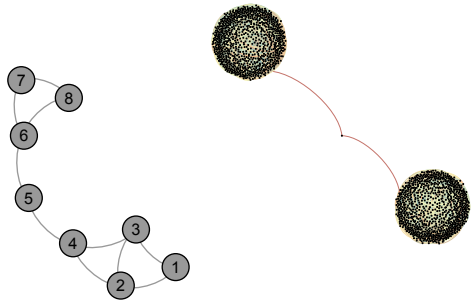

Figure 2: A cluster possibly will not be reached, that makes ExpandWhenStack score twice bad.

these unsure matching and present all the pairs associated as one automorphism set or super-vertex.

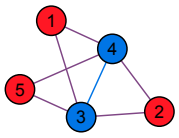

Figure 3: Unpredictable case: 1 and 5 vertex spread their marks, thus [3,3'] [3,4'] [4,3'] [4,4'] have the same score and the same probability t be selected because the degree of vertex 3 is the same to vertex 4. Under conducted experiment accuracy = 3/5.

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 significantly 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 $V = 10^6$.

3.2 ExpandTopStack modification

NEW, ExpandTopStack description

The main aim of ExpandTopStack is to be able to process graph matching over huge graphs with a good performance. The decision was made over percolation approach. As a main basis for developing novel algorithm we took the best percolation based graph matching algorithm *ExpandWhenStuck*. In analysis of related work conducted over various graph models *ExpandWhenStuck* showed the best performance. The main feature of *ExpandWhenStuck* is the seed set expanded by many noisy candidate pairs whenever there are no other unused matched pairs. In step-by-step explanation, there are no further pairs with score at least two, all the matched pairs add all the unused and unmatched neighboring pairs to the candidate pairs afterwards new marks would be spread out by these neighbors. Among these candidate pairs, there are a small fraction is correct and most of them are wrong, correct pairs help the algorithm to continue the percolation process and match remaining unmatched pairs in right way, and wrong pairs would have a negligible effect in most cases of graph distribution given. The ExpandTopStack also implements the next ideas that solves the critics.

1 ExpandTopStack

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

$\mathcal{Z} \leftarrow \emptyset$; ▷ set of used pairs
$\mathcal{A} \leftarrow \mathcal{A}_\prime$;
while $|\mathcal{A}| > 0$ **do**
  $\mathcal{M} \leftarrow \mathcal{A}$; ▷ set of matched pairs
  while $|\mathcal{A}| > 0$ **do**
    for all pairs $[i, j] \in \mathcal{A}$ **do** ▷ spread seeds
      add [i,j] to $\mathcal{Z}$ **and** spread marks;
    while exists an unmatched pair with score ≥ 2 **do**
      ▷ init empty map
      $\mathcal{T} \leftarrow Map(V_1, associatedpairs[i, j])$;
      ▷ max score and min deg diff
      $\mathcal{T} \leftarrow$ TOP-k $[i, j] \notin \mathcal{M}$;
      if among $\mathcal{T}$ $\exists [i, j], [i, j'] :$
        $: samescore \wedge |d_{i,j} - d_{i,j'}| = 0$ **then**
        $[v, u] \leftarrow CompressUnpred(G_1, G_2, [i, j])$;
        add $[v, u]$ to $\mathcal{Z}$ **and** spread marks;
        Continue; ▷ Update Top-k result
      $\mathcal{M} \leftarrow values(\mathcal{T})$;
      for all pairs $[i, j] \in values(\mathcal{T})$ **do**
        if $[i, j] \notin \mathcal{Z}$ **then**
          add $[i, j]$ to $\mathcal{Z}$ **and** spread marks;
    $\mathcal{A} \leftarrow$ all neighboring [i,j] of matched pairs $\mathcal{M}$ s.t. $[i, j] \notin \mathcal{Z} \ \wedge \ i \notin V_1(\mathcal{M}) \ \wedge \ j \notin V_2(\mathcal{M})$;
  $\mathcal{A} \leftarrow$ filter $\mathcal{S}$ s.t. remain $score([i, j]) = 1 \wedge \wedge \ i \notin V_1(\mathcal{M}) \ \vee \ j \notin V_2(\mathcal{M})$;
return $\mathcal{M}$

3.3 Dealing with leafs

NEW, leafs and bridges overcoming

As one and the easiest approach for dealing with leafs and bridges we could re-execute *ExpandWhenStuck* algorithm some times having initial seeds for the next iteration from used and unmatched candidates taken from the previous execution

algorithm 1. *ExpandWhenStuck* algorithm is designed as very aggressive in spreading out marks that involve all the vertices that are some hop away and reachable from at least two indent edges, thus in one iteration of *ExpandWhenStuck* we would have fully percolated one cluster without leafs and a bridge vertices unmatched but in candidate set having $score = 1$. After filtering *used seed* from matched ones $\mathcal{Z} \setminus \mathcal{M}$ we get unmatched seed with one score but it does not limit us in decision to match it (line 21). If there is some overlapping leafs incident to one vertex we represent them as one super vertex in other case it is surely matched leaf, the same with bridge (line **??**). As a result the algorithm will not stop up to the last possible match occurs that makes algorithm *ExpandTopStack* more crazy.

3.4 Dealing with error propagation by super vertex encoding

NEW, super vertex introduction

The error propagation is the most dangerous problem with which we could face while using percolation propagation mechanism implemented in *ExpandWhenStack*. As one of possible approach we could use randomization while selecting candidates with the same scores but actually it will not solve the problem. As a solution we propose an approach of dynamic super-graph creation algorithm 2, we will represent these unpredictable set of pairs that forms automorphism group of candidate pairs as one **super vertex** in both matching graphs by the fact that they are the same from the perspective of graph structure and percolation statistics Figure 4 (line 12).

A common theme in all of numerous large-scale systems and applications is the need to analyze large graphs with millions and even billions of nodes and edges. For this reason the application of vertex summarization techniques are used together with different applications over a graph:

2 CompressUnpred

Input: $G_1(V_1, E_1), G_2(V_2, E_2)$, $[v, u]$ ▷ seed of super-vertex
Output: $\mathcal{S}$ ▷ Super vertex

1: $\mathcal{H} \leftarrow \{[i,j] : i = v \ \wedge \ score([i,j]) = score([v,u]) \wedge \wedge \ |d_{i,j} - d_{v,u}| = 0\}$;
2: $\mathcal{R} \leftarrow \emptyset$; ▷ other pairs
3: **for** all $l \in V_2(\mathcal{H})$ **do**
4: $\quad \mathcal{R} \leftarrow \{[i,j] : j = l \ \wedge \ score([i,j]) = score([v,u]) \wedge \wedge \ |d_{i,j} - d_{v,u}| = 0\}$;
5: $\mathcal{H} \leftarrow \mathcal{R}$ ▷ make one group of probable overlapping ▷ find maximum full overlapping of pairs
6: $\mathcal{S} \leftarrow$ MAXAUTHOMORPHISM$(\mathcal{H})$
7: V_1 and $V_2 \leftarrow$ alias vertex as super vertex;
8: E_1 and $E_2 \leftarrow$ unique edges incident to super vertex;

a) **Graph compression**: Transfer a graph thought a network is becoming an overhead operation with growing number of edges and vertices [1]. As a solution we could minimize the representation by MDL (Minimal description length) from Information theory [7], having summary and correction a graph could be stored in even lower memory consumption way.
b) **Protein-interaction net**: Graph partitioning is a promising technique for predicting gene disease associations because it can uncover functional modules in PPI networks. The nodes in this summary correspond to modules in the input network. The summary graph can be further compressed by discarding the list of corrections and applying GS again, resulting in larger modules [6]. This process can be repeated i times, yielding a 'GSi' method. Yielding to semi-supervised clustering method that uses annotations in the training set when creating modules.
c) **Network visualization**: The problem of huge graphs is obvious while we try to visualize it and analyses its structure. In attack graph we describe a state-transition representation that is huge, in comparison the generalized exploit-dependency representation representation could be deduced, that reduces attack graph complexity from exponential to quadratic, which makes attack graph generation computationally feasible for realistic networks [8].
d) **Compression of the WEB structure**: the algorithms are based on reducing the compression problem to the problem of finding a minimum spanning tree in a directed graph thus resulting to super-graph structure [2]. Else two-level representation of Web graphs is proposed in [9], called an S-Node representation, where a Web graph is represented in terms of a set of smaller directed graphs.
e) **Clusters and outliers detection**: AutoPart [3] uses the MDL (Minimal description length) principle to compute disjoint node groups such that the number of bits required to encode the graph's adjacency matrix is minimized.

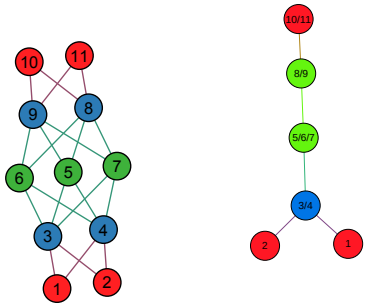

Figure 4: All the vertex has the same probability to be selected, because they have the same degree and the same marks in case of 1 and 2 initial seed thus we could represent them as one super vertex.

As we mentioned before, there is possible unpredictable cases while making matching decision from the top of candidate queue having the same score and difference in the degrees of nodes $|d_{1,i} - d_{2,j}|$ in a candidate pairs. Thus we consider them as formation of automorphism group that could be represented as one super-vertex in both matching graphs.

3.5 Increase performance by batch matching: TOP$-k$ stack

NEW, increase performance by batched selection

As it was mentioned in critics section *ExpandWhenStuck* performance significantly degraded while candidate matching process over huge egocentric network because the searching space has direct relation on deg_{avg} and on count of edges: the search space enriches with candidates by all the pair combinations of a vertex's neighbors. That's why we never could conduct experiment over Chung-Lu model with $V = 10^6$ that in fact generates enormous number of edges. As the next contribution to percolation based algorithms we propose batch matching from candidate set instead of one vertex matching at a time. The batch selection is performed in the same way as TOP$-k$ selection in every iteration (line 10). As a result that approach would be more efficient from the perspective of the performance because this approach has pruning affection to candidate seed generation. In *ExpandWhenStuck* algorithm takes one match and spread marks thus generating new candidates with overlapping pairs that could be already presented in TOP pairs. Having TOP$-k$ selection of candidates for matching we reduce number of candidates generated considering such pair $[i, j]$ as $i \notin V_1(G) \wedge j \notin V_2(G)$. Moreover TOP$-k$ selection permits as a convenient way for super vertex detection, having a big number of a given parameter k the one iteration of ExpandTopStack costs k iterations in *ExpandWhenStuck* thus we start analyze overlapped pairs in less iterations. Thus we consider super vertex detection dependent on TOP$-k$ parameter.

TOP$-k$ technique is first and clear way to increase a performance of a large-scale systems in OLAP system or in an algorithm with huge number of iterations. For this reason the application of TOP$-k$ selection techniques are used together with different applications over a graph.

4 Experimental Study

NEW, experiments over ExpandTopStack

The experiments are performed on synthetic graph that is generated in the next way, (i) generate 3 different Erdős–Rényi random graph with 300000 nodes; (ii) connect them with bridges as in Figure 5. The next experiment was conducted on Facebook graph presented in related work experiments Figure 1.

We now proceed with simulation results of *ExpandWhenStuck* and *ExpandTopStuck* to compare their performance over special cases graphs explained in critics section. The *ExpandTopStuck* having the best features of *EpandWhenStuck* and additional notions implemented bring us an algorithm that outperformed on special cases graphs and has the same performance on related work experiments. In Figure 6 it is clear that *ExpandWhenStuck* is stuck on first cluster while ExpandTopStack passed through bridges between clusters and matched them fully thus having excellent recall measure in that case. The next experiment is conducted over the Facebook

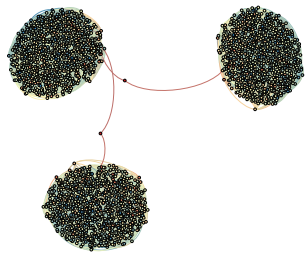

Figure 5: 3 clusters of generated graphs connected with 2 bridges

circles graph. As a result ExpandTopStack has constantly better performance over *ExpandWhenStuck* Figure 7.

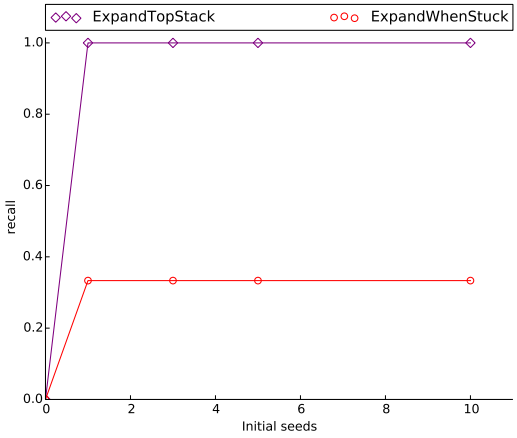

Figure 6: Erdős–Rényi model $n = 900\ 002$ $s = 1$
recall metric: *ExpandWhenStck* is stuck on first cluster while *ExpandTopStack* matched the whole graph correctly

In general ExpandTopStack has the same time complexity as *ExpandWhenStuck* for both real and random graphs. These two algorithms are able to match graphs with only a handful of seeds having the same robust percolation process implementation against a noisy seed set. While there exists some cases in a graph that are hard to predict as a percolation unreasonable choose, in this cases we create super-vertex that also plays the role of pruning of other possible candidates thus making its computational complexity lower in cases of big unreasonable vertex matching number.

5 Conclusions and future work

In this paper, we have presented a highly robust algorithm *ExpandTopStack* based on robustness ideas of *ExpandWhenStuck* and with modifications that avoids its obstacles. Robustness principle permits for the algorithm to be wild in matching nodes while special cases as leafs, bridges between clusters

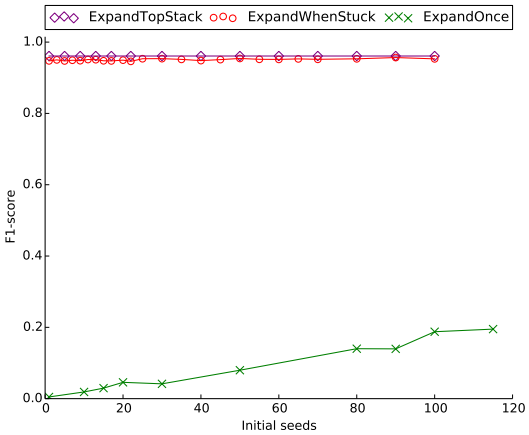

Figure 7: Social circles: Face-book graph $n = 4039$ $s = 1$ F1-score: *ExpandTopStack* matched leafs and covered unpredictable cases with super-vertex

or unpredictable cases from the perspective of a structure could be avoided with repeated use of seeds unused and super vertex covering of unpredictable stack of nodes. As a result we got better performance in special cases experiments and the performance in typical examples as presented in related work.

As for future work, we noticed that in many real life graphs, the edges and nodes also contain various attributes, we could use additional features with respect to structure of a graph, that depend of an application of graph matching techniques on a particular field. For example, nodes in web-graphs have urls, packets in IP traffic have multiple attributes such as port-numbers and type of traffic, etc. Thus we could use this just in super vertex disclosure while the other nodes matching would happen over structure of a graph. Also, as it was mentioned in critics section, we have to create some pruning techniques for candidates generation for increasing performance in memory usage. Afterwards we could consider GPU implementation of percolation graph matching. Proving either the optimality of these algorithms, or conversely, these are definitely other interesting areas of future research.

References

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

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

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

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.