Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к выпускной квалификационной работе

«Поиск подграфов социального графа, включающих заданное множество пользователей»

Автор: Филиппов Дмитрий Сергеевич ________________
Направление подготовки (специальность): 01.03.02 Прикладная математика и информатика
Квалификация: Бакалавр
Руководитель: Фильченков А.А., канд. физ.-мат. наук ________________

К защите допустить
Зав. кафедрой Васильев В.Н., докт. техн. наук, проф. ________________

«___» _________ 20___ г.

Санкт-Петербург, 2019 г.

Студент Филиппов Д.С. **Группа** M4138 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования

Направленность (профиль), специализация Математические модели и алгоритмы в разработке программного обеспечения

Квалификационная работа выполнена с оценкой ____

Дата защиты «___» __________ 20___ г.

Секретарь ГЭК *Павлова О.Н.* Принято: «___» _________ 20___ г.

Листов хранения ____

Демонстрационных материалов/Чертежей хранения _____________________________________

CONTENTS

INTRODUCTION

Researching and analysis of social networks has become very popular in the century of the social networks. One of the examples of the social networks analysis is community search problem. Most of the related works explore communities of the whole network whereas the search of community containing only selected vertices of the network also presents a big interest. This task is also researched quite widely, but adding some requirements that make the problem more related to the real life makes it more interesting and harder. We're going to investigate the problem of finding community containing not necessary all the selected vertices, but only most of them, i.e. making the noise in the selected vertices possible.

The relevance of the initial problem, where all selected vertices should be present in the resulting subgraph may be found in many spheres:

а) Police — knowing several suspects, you need to find the whole groupment or gang;
б) Social networks — after adding one or more friends in the social network, you may find suggestions with people that are densely connected with recently added people useful;
в) Medicine — by several infected people you need to find other possibly infected using the social graph of their familiarity;
г) Party organizing — if you need to call several people on some event, you may find people who are densely connected with them also helpful.

Our problem (which doesn't require the answer to contain all selected vertices, only most of them) is also relevant in the same spheres and besides as you can see, our problem is more close to the real life:

а) Police — find the rest of the groupment, taking into consideration that some of the suspects may be taken only for one deal and doesn't relate to the whole groupment;
б) Social networks — friends recommendation after recent friending, taking into consideration that recently added friends may be from different social communities and may know each other only by you;
в) Medicine — find the most likely infected people, taking into consideration that some of the results may be false-positive;
г) Party organizing — inviting more relevant people, taking into consideration that some already invited people may don't know each other.

Almost all existing articles about community search problem are solving the first task, i.e. the community search with all selected vertices in it. We're going to introduce new algorithm which takes query noise into consideration and solves the community search problem better especially on noisy queries.

In chapter 1 main definitions and terms are presented. Also this chapter contains review of the related work and existing solutions. At the end of this chapter we provide updated requirements for our algorithm to make sure it works good.

In chapter 2 our new algorithm is presented. The algorithm is splitted into several phases for better understanding. You can also find some examples of how this algorithm works with some illustrations.

In chapter 3 the experiments are presented — we describe what data was taken for the experiments, how the test cases and queries were built and how our algorithm works on these experiments comparing to the main baselines.

CHAPTER 1. REVIEW OF TERMS AND EXISTING SOLUTIONS

For now the automation of different problems like community search in social networks is very relevant and progress from day to day. There are different methods for finding communities in the whole network [1–4] and also for finding a dense community containing all selected vertices in the network [5–8]. There are even algorithms for splitting selected vertices into several communities [9] (DOT2DOT algorithm). However, almost all of these algorithms don't support noise in the query and find non-optimal subgraphs on such queries. Precisely this problem we are going to solve in this work, suggesting the new algorithm that will effectively find the needed subgraph even if there is some noise in the query.

1.1. Terms and definitions

1.1.1. Graph terms

In this chapter we will write down all non-common terms and definitions that may be helpful for the further reading.

Under $N_G(v)$ we will understand the set of the neighbors of vertex v in graph G, i.e. the set of the vertices that are directly connected to v by edge: $N_G(v) = \{u|(v,u) \in E(G)\}$. If graph G can be obviously recognized from the context, we can write just $N(v)$.

$G[V]$ is called **originated subgraph** of the graph G by the set of vertices V if $G[V] := (V, E[G,V])$, where $E[G,V]$ — the subset of the set of edges of G, both ends of which are contained in V, i.e. $E[G,V] = E(G) \cap (V \times V)$.

k-truss of the graph G is the subgraph $G' \subseteq G$ containing the maximal possible number of vertices, such that for each edge (v,u) the number of vertices w, such that edges (w,v) and (w,u) exists in G' is at least k. In other words, $k-truss$ is the maximal by size subgraph G', for each of which edge (v,u), the $|N_{G'}(v) \cap N_{G'}(u)| >= k$ is true.

k-core of the graph G is called the maximal by the number of vertices subgraph $G' \subseteq G$, so that the degree of each of its vertices is at least k. For the fixed k, by C_k we will denote *k-core*, namely the set of the connected components of which it consists. So, $C_k = \{H_i\}$, where H_i is the i-th connected component, where the degree of each vertex is at least k. The number k we will call the **order** of *k-core*.

By $\boldsymbol{\mu(G)}$we will denote the minimal degree of the vertices G, i.e. $\mu(G) = \min_{v \in V(G)} deg(v)$.

Core decomposition is the set of *k-core* for all possible k: $C = \{C_k\}_{k=1}^{k=k^*}$. We also need to clarify that from the definition of the *k-core* you can see that $C_1 \supseteq C_2 \supseteq C_3 \ldots \supseteq C_{k^*}$ (where k^* is the maximal possible k in core decomposition).

Core index for the vertex v is called the minimal by the size *k-core* which includes v, i.e. *k-core* with the maximal k: $c(v) = \max(k \in [0..k^*] | v \in C_k)$.

γ**-quasi-clique** of the graph G is called any such subgraph $G' \subseteq G$ that it is «dense enough», i.e. $\frac{2 \cdot |E(G')|}{|V(G')| \cdot (|V(G')|-1)} \geqslant \gamma$.

1.1.2. Social networks

The community or **The community in social network** is called the set of vertices of the social network G, where all vertices are united by some property or attribute. For example, «the community of rock lovers» or «the community of Apple shareholders».

The social clique or **clique** we will call the set of people in social network, where everyone ”knows” (i.e. is connected by edge) each other, in other words when between any pair of distinct people there is an edge in social network.

Social pseudoclique or **pseudoclique** we will call the set of people, where it is not required that each pair of distinct people is connected by edge, but this set is still densely connected. The estimation, how dense the pseudoclique is connected depends on the type of the pseudoclique and will be discussed later in the work, but in all definitions the biggest role plays the number of edges in subgraph in comparison with the number of pairs of vertices ($\frac{2 \cdot |E(G)|}{|V(G)| \cdot (|V(G)|-1)}$).

Free-rider effect is called the effect, appearing during the obtaining the answer for the given problem (finding the dense community in social network), when the final or intermediate answer contains unnecessary subgraphs — subgraphs which can be deleted without violation the optimality of the answer. Thereby, we can make the answer smaller which is one of ours main goals.

1.1.3. Useful abbreviations

RW — Random Walks. The main idea of this method is based on moving from one vertex to the neighbor one with probability proportional to the edge weight.

RWR — Random Walks with Restarts. The idea is similar to the RW, but in this case the probability to move to the initial vertex (from which we’ve started) from the current one exists as well.

Smart-ST — Smart Spanning Trees. The heuristic for Steiner Tree problem, which is used in Gionis et al. article [10].

CSP — Community Search Problem. This is the problem for finding the community in the social network which contains all the selected vertices.

NCSP — Noising Community Search Problem. This is the problem for finding the community in the social network which contains most of the selected vertices, but not necessary all (not including the noise).

1.2. Overview

The problem that we're analyzing in the work is formulated as follows: given a graph G and a set of selected vertices $Q \subset V(G)$, the goal is to solve community search problem — to find the community which contains most of the vertices from Q, but not necessary all of them. Sometimes we will call vertices from Q «query vertices», «query» or «vertices from query».

1.2.1. Initial solutions

а) The community search problem by the given selected vertices is researched during a long time. Even in 2004 Faloutsos et al. [11] suggested the algorithms for finding the dense community by 2 selected vertices in network ($|Q| = 2$). The algorithm shows that metrics like «the shortest path» and «max flow» between two given vertices are not optimal. Instead of them, the initial graph is considered as the electric network and «the current delivered between vertices» metric is used — setting voltage $+1$ on the first vertex-query and 0 on the second one, we find subgraph which delivers the maximal current between vertices from query. The provided metrics works only for $|Q| = 2$, but this algorithm was the foundation in research of the community search problem. After that many authors were working on optimization for this article and were quite successful.

б) Authors of the second article that was taken for consideration **Faloutsos06** suggest the metric function based on *random walks with restarts* (RWR) used on the weighted graph. They consider $r(i, j)$ — the probability that starting in vertex i-th query vertex q_i we will end in vertex j using RWR, where on each step we move to the neighbor by edge vertex with probability proportional the edge weight. Also we introduce $r(Q, j)$ which is equal to the sum of $r(i, j)$ for each query vertices: $r(Q, j) = \sum_{i=1}^{i=|Q|} r(i, j)$. The dense metric is considered as $g(H) = \sum_{j \in H} r(Q, j)$. This method has shown quite good results compared to the previous article, because expanded the number of query vertices from 2 to any number from 2 to $|V(G)|$. Also, it introduced new ability of finding the subgraph containing not all vertices, but at least k of them (k is a parameter which is given as input). This operation was called $K\_softAND$ and was successfully implemented in the article. Further algorithms were expanding this idea, were applying other metrics and improved the results of this algorithm, but the problem of finding the community containing not necessary all query vertices, but only a part of them (our problem), almost wasn't optimized.

в) Authors of the third article [6] suggest to use *Wiener index* as the metric for subgraph density. This metric is equal to the pairwise sum of the shortest distances between vertices from the query. Authors are trying to solve the issue of obtaining too large graph as the result of processing the query if query vertices are placed in several communities and are weakly connected between each other. To solve this issue, authors suggest to add some «important vertices» to the query which will connect communities, even if not very dense. Results has shown that this method works several times better than previous methods [5, 12] and almost the same as the methods that based on *Steiner tree problem*. Unfortunately, the article doesn't consider more late methods based on *Steiner tree problem* which significantly improved the old results, which make this method less priority comparing to them.

1.2.2. Finding optimal pseudocliques

The most part of all algorithms for solving the described problem are the algorithms based on the finding optimal pseudocliques with some additional heuristics. There are a lot of different pseudocliques the were considered in different articles: for example, *k-core* [8], *k-truss* [7], γ*-quasi-clique* [13] or just algorithms that maximizes the edge density in the resulting subgraph [14] which is almost a definition of a pseudoclique. For each of these pseudocliques the algorithms are evolving and becoming better, optimizing the previous results using new heuristics. Comparing the results of the algorithms that use different pseudocliques is quite hard and unlikely will give visible results because of the difference of the metrics that are being optimized — the result strongly depend on the initial graph and the queries on it. In some cases one pseudoclique will obtain results better than others, but in other cases it will work worse, so actually it's worth to compare some common performance metrics, but unfortunately it doesn't give us the whole understanding of the optimality or non-optimality of the algorithms.

Let's consider several newest algorithms for the most popular pseudocliques:

а) X. Huang et al. [7] choose *k-truss* pseudoclique. However, just finding the optimal *k-truss* (i.e. *k-truss* with the maximal k containing all query vertices in it) is not an optimal solution, and also it is an already solved problem (even with polynomial solution). That's why the authors of the article suggest to find *k-truss* with maximal k and minimal subgraph diameter, which, as they show in their article, is a NP-hard problem. However, this idea pretends to show good results, so authors made a research trying to understand how close the answer found by polynomial time may be to the theoretically optimal answer. It turned out that this problem couldn't be solved with accuracy better than in $(2 - \varepsilon)$ times worse for each $\varepsilon > 0$ (under the accuracy we understand the length of the diameter in the final answer). However, authors suggested the heuristic algorithm which in the worst case makes exactly 2 times error, which shows that their algorithm is optimal for the provided problem. The algorithm is based on building the supposed maximal *k-truss* with the followed iterative deleting vertices which doesn't make the answer worse and make the diameter smaller. The results obtained in this article are really good comparing to the previous articles [12, 14], however, even despite of proved optimality for the provided algorithm, it is not optimal for the initial problem (finding the dense subgraph by the given query vertices), because authors found the optimal solution only for the problem provided *by themselves*.

б) N. Barbieri et al. [8] is using *k-core* pseudoclique. However, even here simple finding the optimal *k-core* (i.e. *k-core* with maximal k which contains all query vertices in it) is not an optimal solution, and also this problem is already solved by polynomial time [12]. That's why the authors of the article apply some heuristics targeted for minimizing the size of the resulting subgraph without loosing its optimality. These heuristics allow to reduce the problem to finding the answer in component $H^* \subset G$, and besides it is guaranteed that all possible optimal answers for the initial problem are lying in H^*. After that authors bring some heuristics for minimizing the obtained subgraph H^*. The main statement described by the authors is not new, but it looks quite interesting for the further researches because it adds more information to the initial problem without loosing any solutions. The results of this article shows that provided method really works better and faster than previous ones [12, 15]. Based on all the above information, it was decided to take this article as the baseline and try to improve it, especially to expand it to our problem (finding the community containing not necessary all selected vertices).

1.2.3. Other methods

As we already saw earlier, optimizing functions may be quite different. In the previous part we were considering pseudocliques, and here we're going to consider several other popular optimizing functions.

а) L. Akoglu et al. [9] slightly change the initial problem — they try to find subgraph that unites not all the vertices in the query, but their groups. Actually the idea is based on splitting the query into groups and building the answer for each of the groups separately, so that in each groups the vertices are densely connects and are united by some common property. But between each other the groups may be connected not very densely. This corresponds to the splitting the query into several communities. The results of the article has shown that this method solves the problem provided by authors quite good, but however as we said before, this problem differs from ours and it's hard to compare it's results with ours. However, we still are going to compare this solution with ours, because this solution supports finding community not for all selected vertices.

б) A. Gionis et al. [10] in their article consider *linear local discrepancy* metric which is equal to the weighted difference of the number of query vertices in the resulting subgraph and the number of remaining vertices. More formally, $g(C) = \alpha p_C - n_C$, where $p_C = |Q \cap V(C)|$ and $n_C = |V(C) \setminus Q|$. The algorithm that maximized this function based on the *Steiner tree problem* and *Smart-ST*s. The distinguishing feature of this algorithm is the ability of solving the problem using *local access model*, i.e. the model where we don't know the whole graph (or it is too big to save it in RAM), and API allows us only to make queries for accessing all vertex neighbors — $get - neighbors$ method. This model allows to solve the problem optimally even on very big social networks, such as *Twitter* or *Facebook*. Based on the definition, the resulting subgraph may not contain all vertices from query, which coincides with our research. Unfortunately, the provided metric doesn't fit our problem very good — it doesn't take into consideration edge density of the resulting subgraph, it looks only the ratio of the vertices. Also it's possible that the answer will contain too few vertices from the initial query — it also doesn't work for us.

1.3. Final requirements for our work

Let's sum up everything described above. Most of the current solutions solves CSP quite optimal — each of the solutions uses it's own metric and obtains quite good results. However, as we can see, solutions for NCSP (which includes noise into consideration) are quite rare, despite it is more useful problem in real life. We've noted consideration of NCSP problem in articles C. Faloutsos & H. Tong **[3]** and A. Gionis et al. [10], however the last problem is not based on solving NCSP (but solves it at the same time). So, the goal of our article will be to build the algorithm that focuses on NCSP solving and obtains better results than the current ones. Here are some requirements for our algorithm:

- The algorithm should obtain better results than the current ones [5, 8, 10];
- The algorithm should be quite optimal, ideally not loosing the competition with other algorithms in terms of working time;
- It would be an advantage to support backwards compatibility — if the user wants to find subgraph that contains *all* query vertices, it should be possible to be done.

CHAPTER 2. ALGORITHM

First of all, we need to answer two questions:

а) How the subgraph density should be measured?

б) What is «the most of the selected vertices»?

To compare two subgraphs obtained by different algorithms we will use edge density and the size of the resulting subgraph: $density(G) = \frac{2 \cdot |E(G)|}{|V(G)| \cdot (|V(G)|-1)}$, $size(G) = |V(G)|$. Really, if the number of edges in obtained subgraph is quite big, we can assume that it is dense. However at the same time it should have the smallest possible size. What is «the most of the selected vertices» will be discussed a bit later.

2.1. Idea of the algorithm

Because there are quite few algorithms that solves exactly our problem, we have two ways: to come up with absolutely new idea and algorithm or to take an existing idea and try to improve it or to make it work on our problem instead of common *CSP* with all query vertices needed to be in resulting subgraph. Starting from now we will use *CSP* abbreviation for community search problem, where all query vertices should be presented in the resulting subgraph and *NCSP* for noisy version of the problem, what we need only most of the query vertices in the answer.

Absolutely new algorithm may look like this: we can bruteforce the set of vertices which will be noise, then find dense subgraph that contains all query vertices except bruteforced ones using one of the described in first chapter solutions and we're done. This solution will obviously return the most optimal answer, but it works too slow, the complexity of this solution is not polynomial. So, we won't use this method and will try to come up with something else.

The easier way is to take some existing algorithm for *CSP* and transform it for *NCSP*. Really, it is proved that such algorithm works good and is one of the best for now, so we have a lot of chances to beat all current *NCSP* solutions in that case. Here we go: let's take algorithm Barbieri et al. [8] for finding k-core with maximal k and minimal size and try to optimize it and change so that is will solve *NCSP*. One more note that makes us more assured about this idea is two articles — Bogdanov et al. [16] and Cui et al. [15] which showed that maximization of the minimal degree in subgraph is very effective for finding the optimal community, so our idea absolutely may take place.

The idea of Barbieri et al. algorithm [8] is in finding in graph G *k-core* with the maximal k and among all such solutions, we want to find the one with the minimal number of vertices in it. In their work, authors show that it is a NP-hard problem, so we need to use some heuristics to solve it. The idea of algorithm is based on one interesting fact, which we're going to use in our work as well: consider the core decomposition of graph G: $C = \{C_k\}_{k=1}^{k=k^*}$. Now let's find such maximal k' that all vertices from query are lying in the same component of $H^* \in C_{k'}$. The fact is that all solutions for *CSP* are lying in this component H^*. More formally: $k' = \max\{k|\exists H_i$ — connected component $C_{k'}$, such that $\forall v \in Q : v \in H_i\}$. The found *k-core* we will denote as C_Q^* and the component which contains all query vertices — H^*. We need to note two things here: first — this statement helps us to decrease the size of the initial graph without loosing solutions and second — this statement is true even for *NCSP*, because if H^* contains all optimal subgraphs with all query vertices, it contains all optimal subgraphs containing only a subset of query vertices as well. It is possible that the resulting subgraph will be a *k-core* with bigger k than H^* have, but nothing prevent us from starting our algorithm from H^*.

Let's explain the final idea of our algorithm: by given social network G and the selected vertices Q, we will find *k-core* C_Q^* and its connected component H^* that contains all optimal solutions for *NCSP*. H^* is actually one of the candidates for the answer, because it contains all vertices from Q, but it's too large and we want to make it smaller (also, this subgraph contains all query vertices, i.e. the noise as well). So, after we found H^*, we are going to apply some heuristics that will allow to decrease the size of H^* without loosing the condition on minimal vertex degree (i.e. *k-core* invariant) and therefore not making the answer worse.

2.2. Step-by-step algorithm

In this section we will describe the whole algorithm. Actually, we can divide the algorithm into two phases:

а) Find C_Q^* and H^*;

б) Decrease size of H^* without loosing the *k-core* property.

Actually, in future we will split the second phase into several more, but let's talk about it later. Now we will try to sort out with each phase separately.

2.2.1. Phase 1. Finding C_Q^* and H^*

C_Q^* and H^* finding phase will be taken from Barbieri et al. article [8] as well as their idea of minimization of *k-core*. It's easy to see that actually core decomposition doesn't depend on the query, so it's not necessary to build it each time. But unfortunately saving it in RAM before all the queries is not a good idea as well, because it's size may be too big. That's why we're going to do a pre-calculation which will build a core decomposition once, compress it and save on disk. It will be a so-called index which will allow us to find C_Q^* and H^* much faster for each query.

In index of our core decomposition $C = \{C_k\}_{k=1}^{k=k^*}$ we will store the following information:

а) Core indices for all vertices $c(v) = \max\{k \in [0..k^*] | v \in C_k\}$;

б) For each *k-core* C_k we will store the set of its components $C_k = \{H_i\}$.

The following picture illustrates the above statements 1:

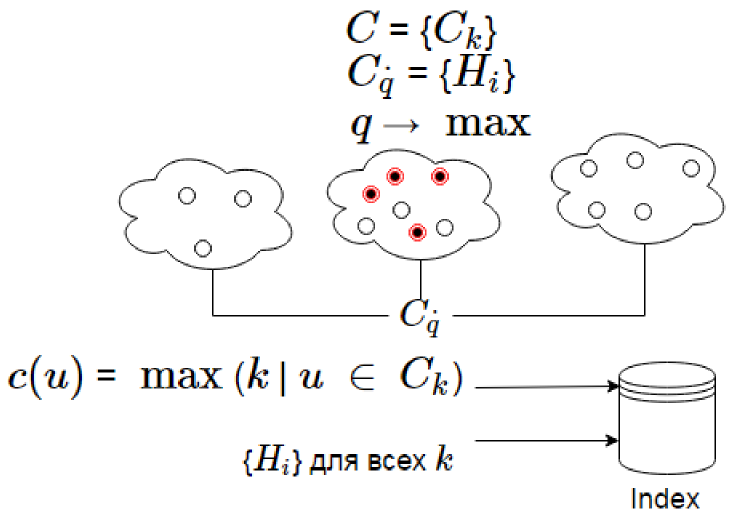


Figure 1 – Index construction

We also need to take a small note that some neighboring *k-core*s are equal: $C_k = C_{k+1}$. There is no need to store them twice, so we will store all duplicating neighboring *k-core*s only once. For now it doesn't look like a very good optimization, but actually on the real datasets the number of different *k-core*s may be even several orders less, so this optimization makes sense. We will also use variable h in future — it means the number of different *k-core*s in our core decomposition, i.e. the number of *k-core*s which we're going to store in our index.

We will store code indices and sets of connected components in the appropriate data structures (hash tables). It will allow us to access the following object in $O(1)$ complexity:

— $c(v)$;

— all components of the fixed *k-core*;

— a component of the fixes *k-core* where the given vertex v is located.

But actually we haven't said yet how to build this index. Actually it's quite simple: let's firstly build the set of components for C_{k^*}, after that all other *k-core*s will be built in the following way: if *k-core*s for $k^*, k^* - 1, \ldots, k$ are already built, then to build $(k-1)$-*core* we will add all vertices from $C_{k-1} \setminus C_k$ one by one and edges incident to them (according to the property that $C_{k-1} \supseteq C_k$, it's enough) and then we will update connected components which according to the new vertices can union in new, bigger components.

This phase takes $O(h \cdot |V(G)| + |E(G)|)$ time or simply $O(hn + m)$, where h as you can remember is the number of different *k-core* in G. However this asymptotic won't be taken into the final consideration because for each graph this computation is taken only once (so we can wait if needed) and then the built index is used for finding C^*_Q and H^* for each query.

So, how can we find C^*_Q and H^* by our built index? Actually, it's very simple. Because $C_1 \supseteq C_2 \ldots \supseteq C_{k^*}$, to find *k-core* with maximal order that contains all query vertices from Q in its one component, we can use binary search by the *k-core* order. To check if for some fixed k, C_k contains all vertices of Q in its one component, we will take for each vertex from Q information about the component it is lying in (using the index, we can do it in $O(1)$ time) and then check that all these numbers are equal. So, the check for each binary search step will take $O(|Q|)$ time.

Summing up everything above, we see that the step of finding C^*_Q and H^* takes $O(|Q| \cdot \log(h))$.

2.2.2. Phase 2. Decreasing H^* size

After finding C^*_Q and its connected component H^* that contains all vertices from Q as we mentioned earlier it's easy to see that H^* is the answer for our problem, because all its vertex degrees are at least k and k is the maximal possible (according to H^* construction). However, we still have one more condition to be true — the resulting graph should have the minimal possible size, now it's obviously not true. This is what we're going to fix in this and the next phases. In this phase we are going to remove noise from H^*, i.e. to select its subgraph which contains most of the query vertices (except the noise).

Let's remind that the problem of finding minimal by size *k-core* — NP-complete. Which heuristics can we use? The first idea which comes to mind is to delete weakly connected vertices or subgraph from H^*, therefore making its size smaller. However, because all vertices degrees are at least k, it may be hard to understand which vertices are weakly connected and which are not. Also, it's not obvious how to understand which subgraphs can be deleted without loosing the condition for minimal vertex degree.

We've chosen a reverse approach — adding vertices. We will build the final subgraph H_{min} in the following way: let's take all query vertices Q and delete all other vertices and edges. After that we will add other vertices one by one with some priority, making our subgraph H_{min} larger and more connected. At each moment, when the current subgraph satisfies the answer for our problem (i.e. the component of the current subgraph H_{min} with the maximal number of query vertices in it, contains «the most of the query vertices» and it is also «dense enough»), we will update the final answer. It's obvious that the process is finite — the number of vertices in the current subgraph is growing and we won't obtain the subgraph larger than H^*. However we face three global questions:

а) What is the priority for adding vertices to the current subgraph?
б) When do we need to update the answer (i.e. what is «the most vertices from Q», and what is «component is dense enough»)?
в) When do we need to stop adding vertices?

a. What is the priority for adding new vertex?

There are tons of priority options. However let's think, what metrics are important for us. Firstly, we want vertices that form a community (i.e. the set of query vertices without the noise) to be united into one connected component as soon as possible and began to increase this component density. So, as the first priority we will take the number of components which the newly added vertex unite. I.e., $p_1(v) = |A'| - |A|$, where A is the set of components before adding v and A' is the set after adding it. However after adding the new vertex in many cases the number of components won't change. What do we need to do in this case? Because vertex degrees in the final subgraph are also important for us, let's emphasize attention on it. When adding vertex v, degrees of its neighbors which are already contained in the current subgraph, are increased by 1. We want to maximize this count. However, we also need to take into account the recently added vertex and its degree. So, let's make the following second priority: $p_2(v) = |N_{H_{min}}(v) \cap \{v \in V(H_{min}) | deg(v) < \mu(H^*)\}| - \max(0, \mu(H^*) - |N_{H_{min}}(v)|)$, where $N_{H_{min}}(v)$ is the set of vertices from H_{min} which are connected with v by edge. In other words, we take the number of neighbors of vertex v from H_{min} for which degree is less than needed $\mu(H^*)$ for now with plus sign and the number of edges which is needed for the recently added vertex to have degree $\mu(H^*)$ with minus sign.

b. What is «the most of the vertices from Q», what does «the component is dense enough» mean?

We will say that subgraph $H \subset G$ contains for most of the vertices from Q if the number of query vertices in it is at least $\alpha(|Q|) \cdot |Q|$. $\alpha(|Q|) \in (0, 1]$ or just α is some coefficient which depends on the number of query vertices. But what α should be equal to? From one side we want «the most of the vertices» to be really the most part of them, so we will assume that there is less than $\frac{|Q|}{2}$ noise vertices in the query (i.e. $\alpha \geqslant 0.5$). From the other hand, even $\frac{|Q|}{2}$ noise vertices is too much in most cases.

We are also interested in what «the component is dense enough» means. We will say that the component is dense enough if it contains enough edges (it obviously means that its dense is quite high). To select the bound of the edges count we tried several functions, and the most optimal one turned out to be the following: $|E(H_{min})| \geqslant (V(H_{min}) - |Q|) \cdot \mu(H^*) + \beta(|Q|) \cdot |Q|)$, i.e. for all query vertices we take their degree equal to β (one more parameter) and for other ones degree should be at least $\mu(H^*)$ (actually if there is some noise in the query, degree should be even higher, however, if there is no noise, the previous statement is true).

To choose optimal values for parameters α and β given the fixed $|Q|$ we decided to make some experiments (they we made on the same data as all other experiments described in chapter 3). Results of the experiments are shown in the table below (k is the order of *k-core*, i.e. $k = \mu(H^*)$).

Table 1 – Optimal values for α and β for different $|Q|$

$\lvert Q\rvert$	α	β
2	1	1
3	2 / 3	1
4	1 / 2	1
5	3 / 5	2
6	2 / 3	2
7	4 / 7	3
8	3 / 4	3
> 8	7 / 10	4

c. When the algorithm should be stopped?

It's simple to understand that the algorithm can be stopped when all vertices from H^* are added. However, because the size of H^* may be quite big, it's not the optimal solution — when our current subgraph became connected, it will contain all query vertices and thus all the noise, while our task is to avoid noise in the result subgraph. So let's stop our algorithm when our current subgraph became connected, we won't loose any answers in that case.

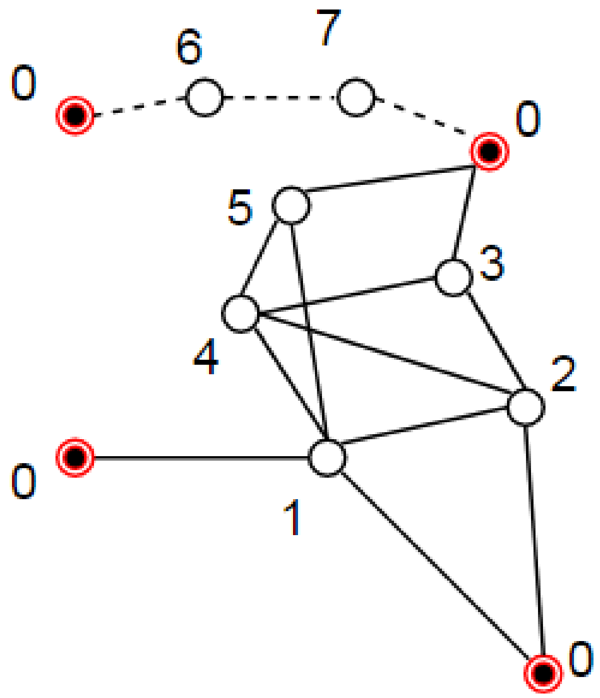


Figure 2 – Example of phase 2 work

Let's assume that H^* looks like at the picture 2. Query vertices are marked red, we're taking them at the first step. After that we will add vertices one by one, as it is shown on the picture: firstly we add vertex 1, because it unites two components, then 2 because it has the maximal second priority, etc. Our algorithm finishes when H_{min} becomes connected, in that case it will happen when all edges will be added. However, the optimal subgraph will be the one without the dotted edges, because it will be more dense. And as we can see the vertex that was not added is an obvious noise, so we shouldn't add it to the result subgraph.

2.2.3. Phase 3. Restoring the condition for vertex degrees

After phase 2 is completed, we have a subgraph H^*_{min}. This subgraph can be returned as the answer, but of course as we just saw on the picture 2, H^*_{min} can be far away from the optimal answer. Why? We were adding vertices one by one with, as it seemed, optimal priority. Actually, because we focus on adding vertices that unite our current components the best and take vertex degrees into account only as the second priority, the obtained subgraph can be non-optimal. That's why we need to do something with H^*_{min} to satisfy the condition for vertex degrees.

You may now think that phase 2 was redandunt, because we had the *k-core* which was densely connected and we obtained the subgraph which is not *k-core* and thus is not densely connected. That's not true, because the initial *k-core* contained all query vertices and new subgraph doesn't contain query noise. Also, as you will see, we will apply some heuristics and new subgraph H^*_{min} will become a dense *k-core* of bigger order.

To begin with, let's remove weakly connected vertices. Because our new subgraph H^*_{min} is not a *k-core*, we can suggest logical conditions for that. Let's remember parameter β which we introduced to be responsible for the minimal query vertex degrees. Let's remove all query vertices that have degree less than β, i.e. which doesn't follow our invariant. After deletion, let's run phase 1 one more time on the left query vertices. Actually, we don't need to run the whole phase 1, we only need to find the new value k^{opt} — the maximal order of *k-core* in which all remaining query vertices lie in the same component.

Now we will do the main step of the final phase. Its idea is in the following: we will take all remaining query vertices and find the minimal number of non-query vertices that connect them. In other words, taking the remaining subset of query vertices $Q' \subseteq Q$ we want to find such set of vertices $V_{opt} \subseteq V(H^*_{min}) \setminus Q'$ of minimal size so that $G[V_{opt} \cup Q']$ is connected (to remind, $G[V]$ is a subgraph originated by the set of vertices V). It will give us a «skeleton» of subgraph H^*_{min} which we will build up almost the same way as we were doing in phase 2.

The problem that we have described above is a *Steiner task*. The original version of Steiner task sounds as follows: given a weighted graph G and a subset of selected vertices Q in it, you need to find a minimal spanning tree on the selected vertices. It's quite interesting that for $|Q| = 2$ the task is just to find the minimal path between two vertices, if $|Q| = |V(G)|$, the task is just to find the minimal spanning tree, and in all other cases the task is NP-hard. You can mention that in our case graph is unweighted, but actually Steiner task remains NP-hard even on unweighted graphs. Steiner task have been investigated many years ago, and Kou et al. [17] gave us an optimal algoriths for solving it with approximation $2 - \frac{2}{|Q|}$ and proof that it's impossible to solve it better in polynomial time. We will take this solution and apply it to our task.

After finding the needed «skeleton», we will apply slighly modified phase 2 to it. First, skeleton is already connected and we don't need priority for uniting the components. Second, we want to guarantee the density of the final subgraph, so we need to modify condition a little bit:

а) We will add new vertices only by second priority;

б) We will stop only when the minimal vertex degree is at least k^{opt}, because we know that it's possible to build k^{opt}*-core* on the remaning query vertices;

в) We won't update the answer during these iterations, we will take the final subgraph as the answer.

After changing the phase 2 as described above, we apply it on H^*_{min} with deleted weakly connected query vertices and get the final answer to the task.

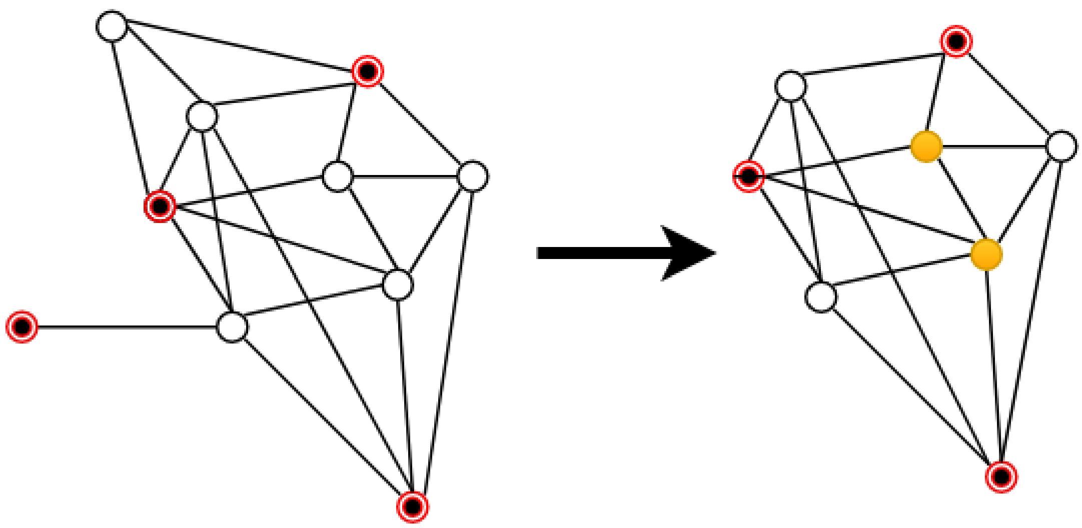


Figure 3 – Phase 3 example

The work of this phase is presented on picture 3 where subgraph H^*_{min} (3-core) is places on the left and the final subgraph obtained after phase 3 is placed on the right. First, we remove weakly connected query vertex and then found two yellow vertices which along with remaining query vertices give as a skeleton of 5 vertices. After that we apply modified phase 2 on the skeleton and obtain the final subgraph. As you can see, it is smaller and more dense than the first one — it is even 4-core when the previous was only 3-core.

CHAPTER 3. EXPERIMENTS

In this chapter we will describe practical experiments of our algorithm and their results. First, we will describe on which kind of data the experiments were made and how the experiments look like, and after that we will show diagrams describing how well our algorithm did during these experiments.

3.1. Experiments description

Experiments were made on two real datasets: *DBLP* — graph, where vertices are the article authors and an edge between two vertices means that there is a common article between these two authors. Also, we made our experiments on *Youtube* dataset — social graph where vertices are Youtube users and the edge between two vertices means mutual subscription.

For our experiments we've taken 50 random queries without noise, 50 queries with noise and 5 queries with absolute noise, i.e. containing vertices that are have almost no common between each other. We will describe queries building more detailed below.

а) In first experiment we took 50 random queries on graph, each of which contained vertices that highly correlated between each other. We built query by taking *k-core* with quite big k and then taking vertices from it randomly, however, taking into account that vertices shouldn't be far from each other. Here we should point out that Barbieri et al. [8] didn't take it into account and obtained size of components were almost 10 times bigger than ours. As a result of the experiment, we were calculating average size of the community over 50 experiments.

б) In the second experiment we took 50 random queries on graph such that most of the query vertices highly correlated between each other and remaining vertices were noise, i.e. weakly correlated with other vertices. We built query by taking *k-core* with quite big k (i.e. the query generated by the first experiment) and adding few vertices from *k-cores* with less k, i.e. the vertices being the noise. As a result we calculate the average size of the communite over 50 experiments.

в) In the third experiment to analyze the behaviour of our algorithm when the query is almost fully noisy, we choose several vertices from different *k-cores* which are weakly connected and have almost no correlation betwen each other. This experiment has shown that on such queries our solutions build quite big subgraph, but it still filters noise quite good and the resulting size of the community is much smaller than in other algorithms.

After each run of our algorithm, we also do answer validation — that the resulting subgraph contains most of the initial query vertices — at least $\max(\frac{|Q|}{2}, |Q| \cdot \alpha - \epsilon)$, where ϵ is a small number showing that after phase 2 of our algorithm we could remove several weakly connected vertices on phase 3. For our experiments we took $\epsilon = 4$.

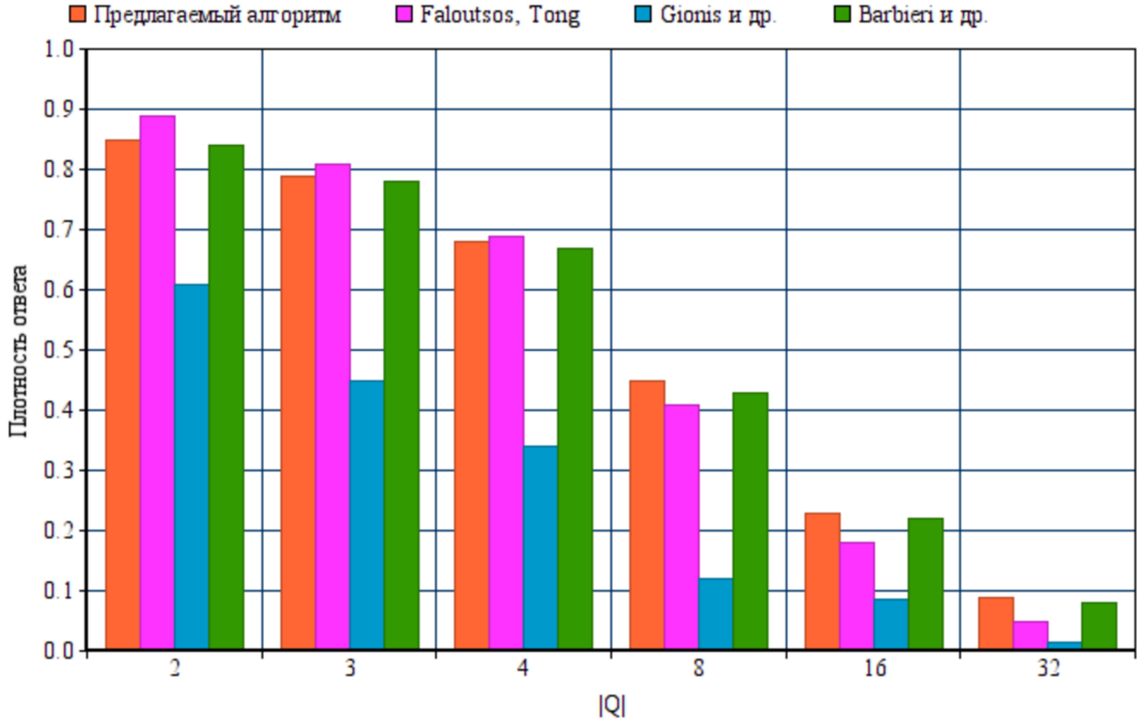
3.2. Results

Here we will provide results of our experiments comparing our suggested algorith with Faloutsos & Tong [5], Gionis et al. [10] and Barbieri et al. [8]. For better vizualization we will provide several diagrams with small description. For each of datasets we will provide 6 diagrams comparing density and size of the obtained subgraphs, and time taken to find these subgraphs (we don't take into account time taken for precalculations, only query time). We build diagrams for 2 different types of queries — with and without noise respectively. In each diagram horizontal axes means different sizes of $|Q|$ (either 1, 2, 3, 4, 8, 16 or 32) and vertical axes maans the density of the obtained subgraph, its size or time taken to find it.

3.2.1. DBLP dataset

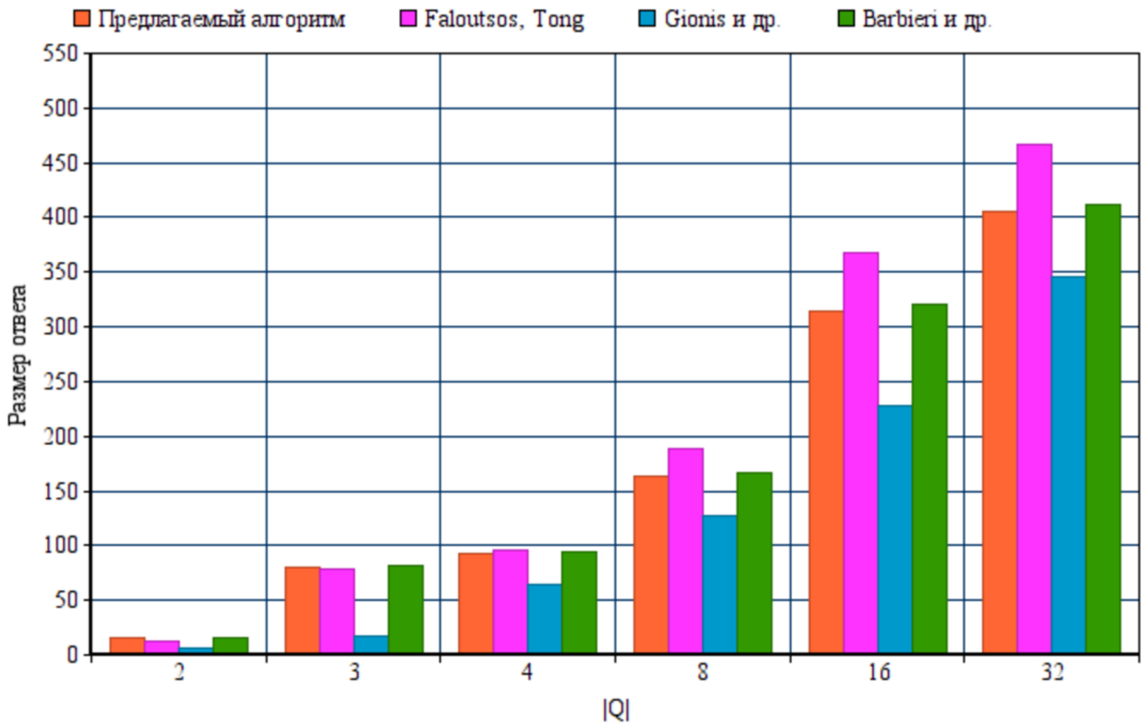
DBLP dataset contains 317080 vertices and 1049866 edges. It consists of article authors, where the edge between two authors means having common article between them.

3.2.1.1. Queries without noise. Community density



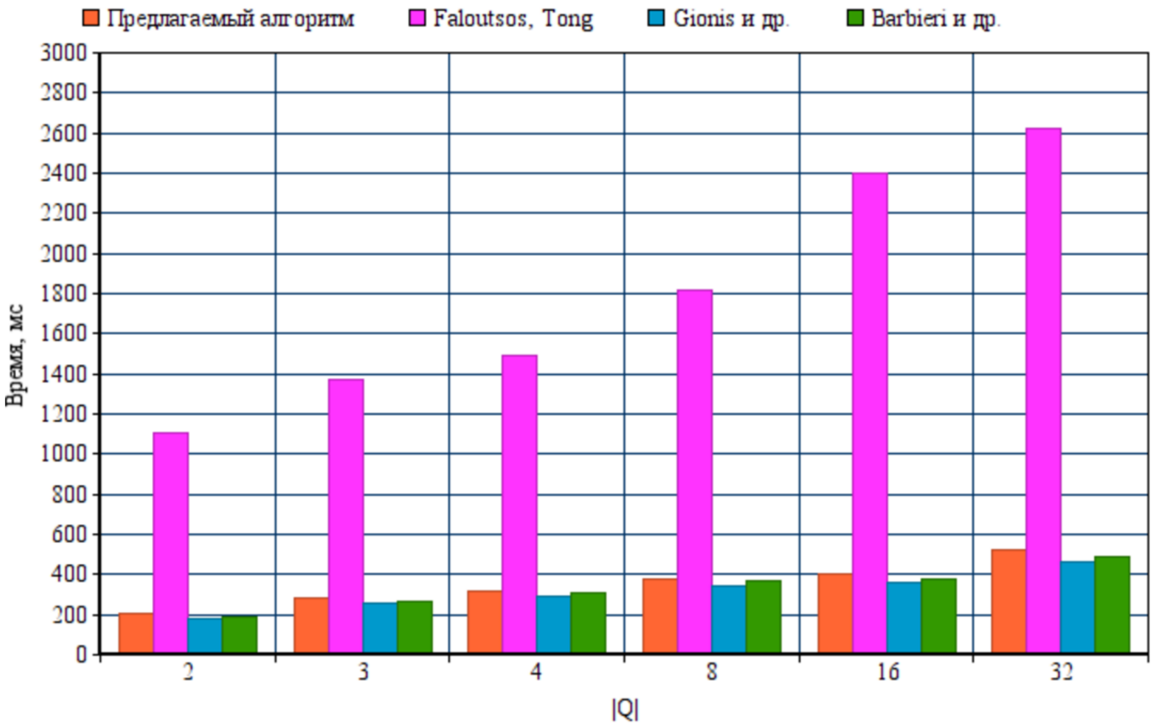
As you can see from the diagram, on random queries without noise the density of the community obtained by our alrogithm is a bit greater than the density of community obtained by Barbieri et al. [8], what means that our algorithm is more optimal. Also, the density of our community is more on average than the density of Faloutsos & Tong [5] and much more than the density of Gionis et al. [10].

3.2.1.2. Queries without noise. Community size



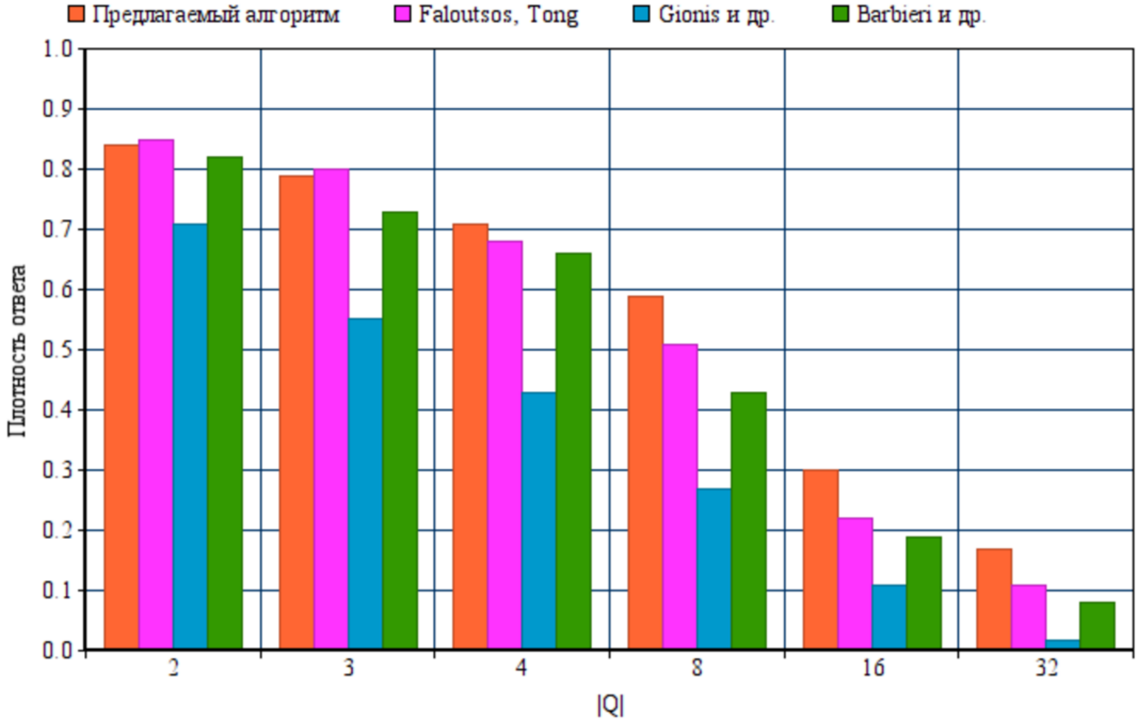
As you can see from the diagram, on random queries without noise the size of our community is a bit less than the size of community obtained by Barbieri et al. [8], which is quite logical from the previous diagram. The size of our community is also less on average than the size of Faloutsos & Tong [5], but greater than the size of Gionis et al. [10], but because our main goal is density, not the size, it's not an issue.

3.2.1.3. Queries without noise. Work time



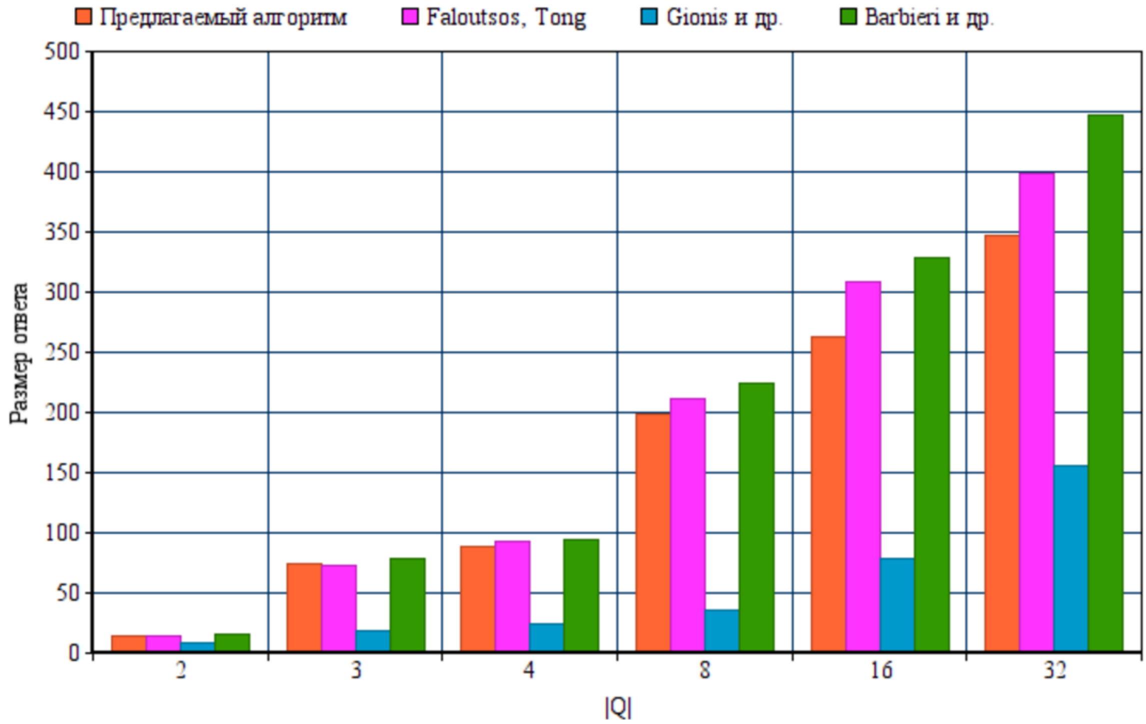
As you can see from the diagram, our algorithm takes a bit much time to process than Barbieri et al. [8], but this loss is very minor. Faloutsos & Tong [5] takes much more time to find the answer and Gionis et al. [10] works faster, but as we saw previously, it finds not dense subgraph, so it's ok for us.

3.2.1.4. Queries with noise. Community density



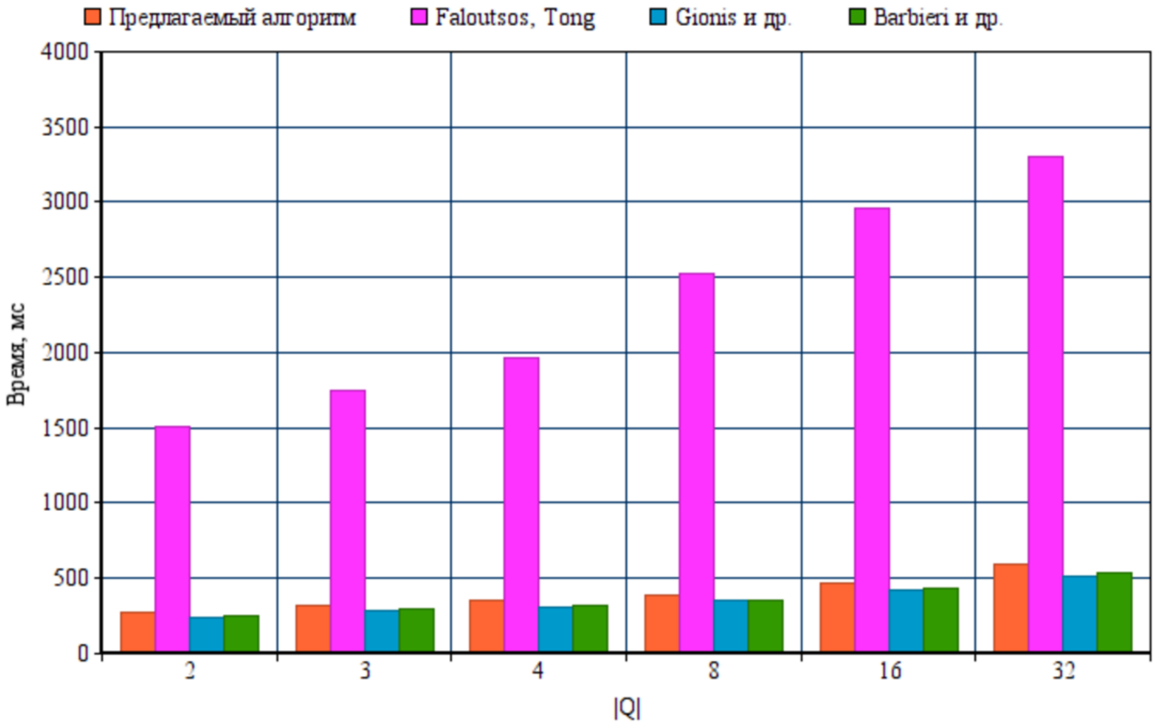
If the query contains some noise, our algorithm starts working much more optimal comparing to other algorithmis. You can see it on the diagram above: on small $|Q|$ our win is not so big (because there is almost no noise), but on bigger $|Q|$ we are winning much more in terms of density and size.

3.2.1.5. Queries with noise. Community size



As you can see from the diagram, if the noise exists, the size of our community is less than the size of community of other algorhtms (besides Gionis et al. [10], but it works worse in terms of density).

3.2.1.6. Queries with noise. Work time



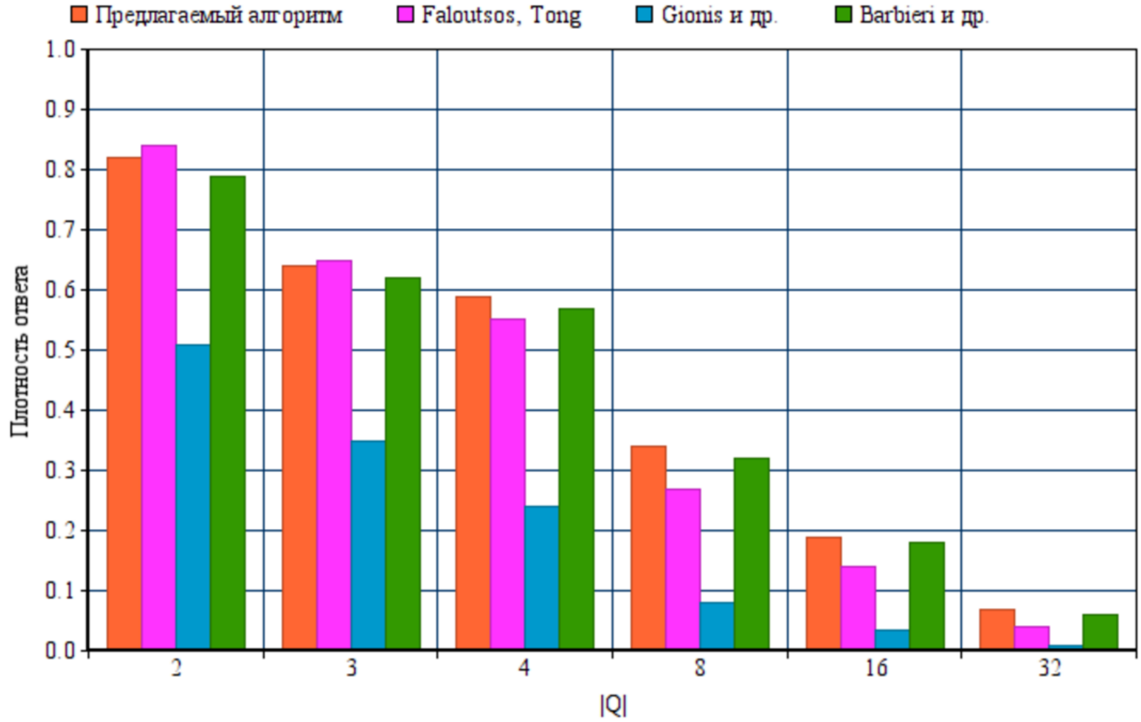
Work time of our algorithm on queries with noise is a bit more than work time of Barbieri et al. [8], however this loss is very minor and we won't take it into account.

3.2.2. Youtube dataset

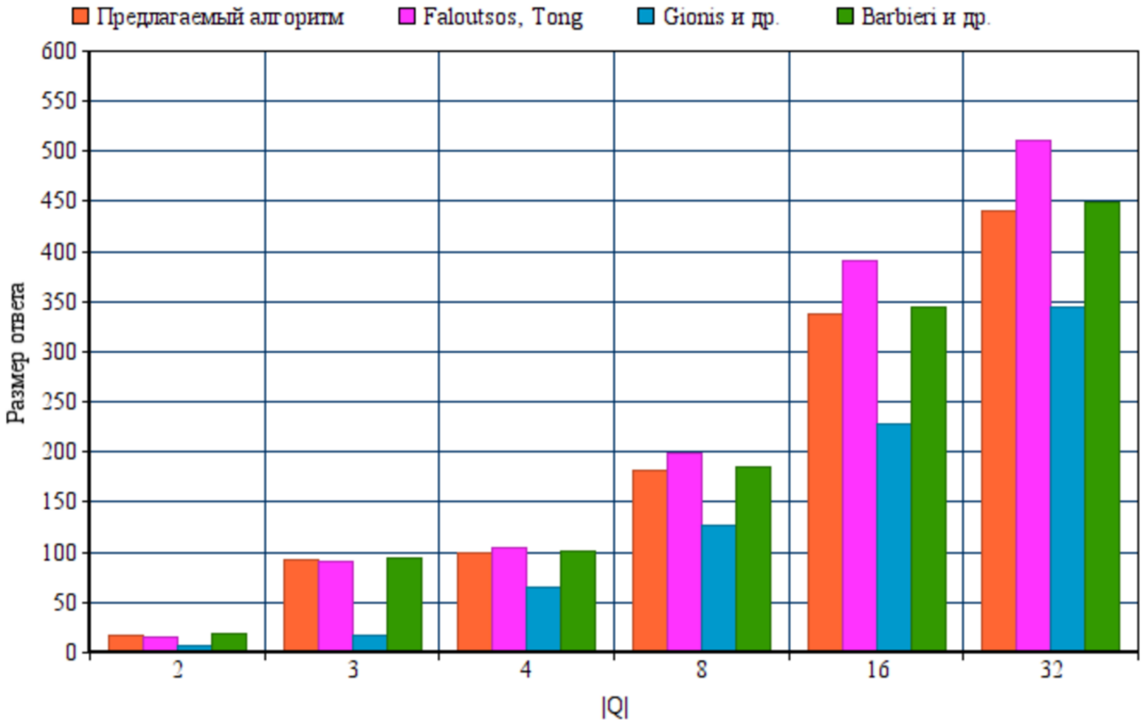
Youtube dataset contains 1134890 and 2987624 edges. This dataset consists of Youtube users and an edge between two vertices means that these two users have a common subscription, i.e. they are friends of each other.

Results for this dataset are pretty similar to the results for *DBLP*, so we won't leave any comments for the diagrams and let you make all findings youself.

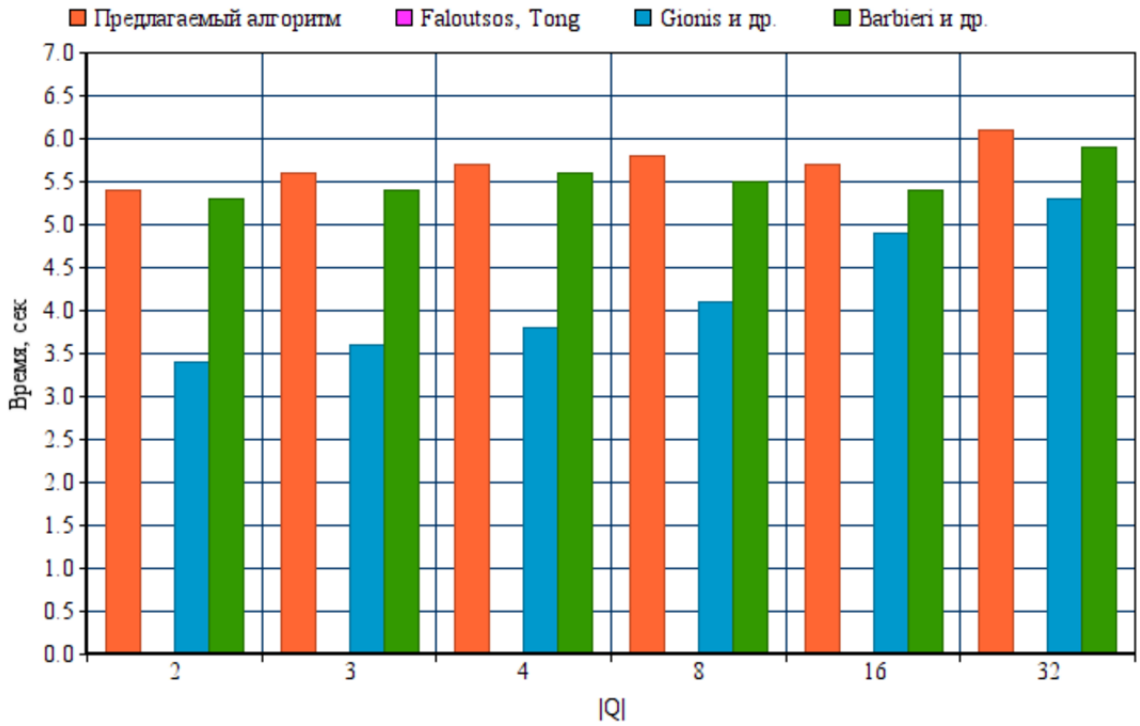
3.2.2.1. Queries without noise. Community density



3.2.2.2. Queries without noise. Community size

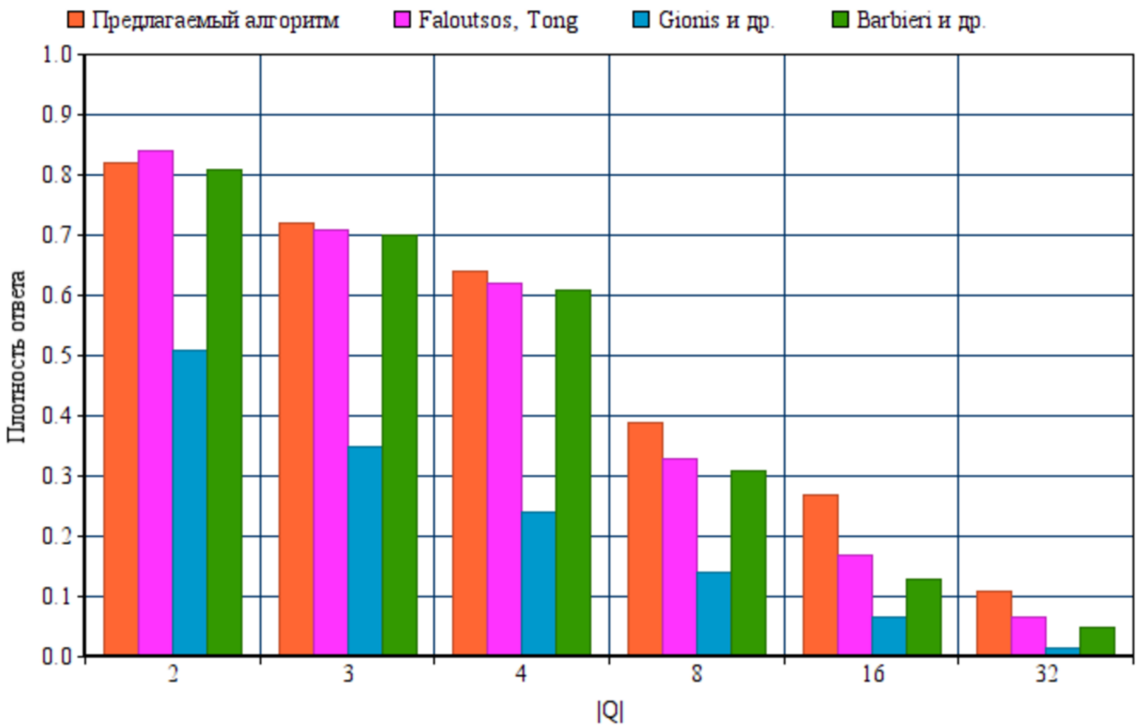


3.2.2.3. Community without noise. Work time

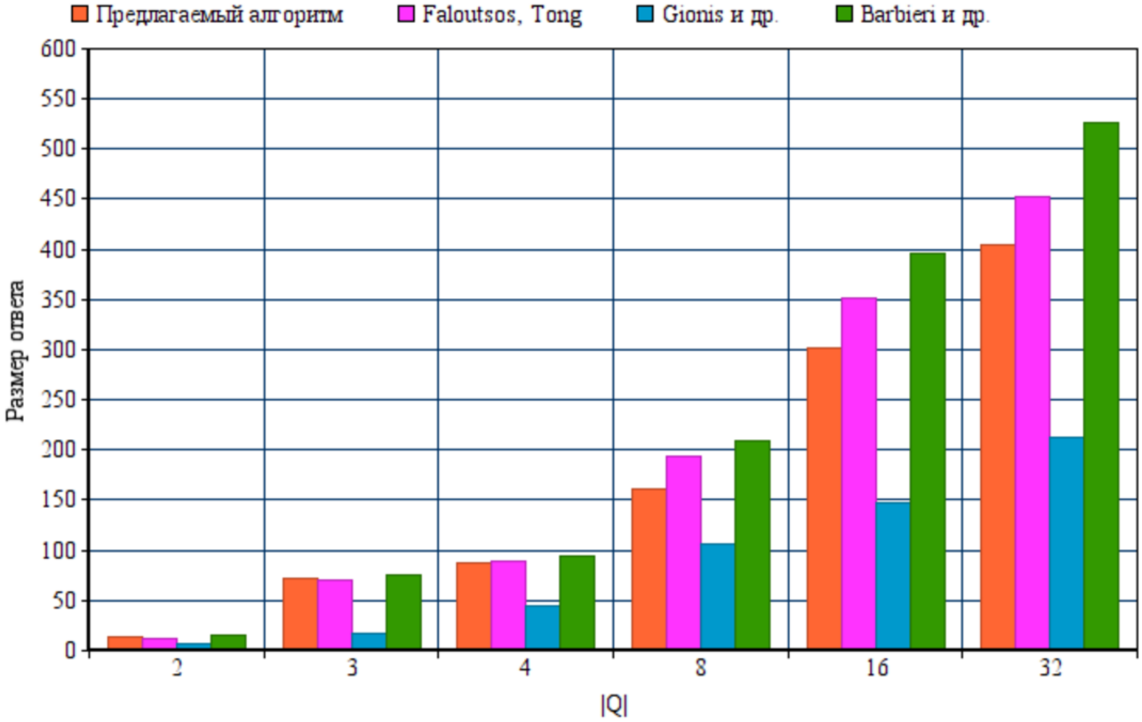


Here we can mention that Faloutsos & Tong [5] algorithm works takes too much time to proceed, so we removed it from diagram to avoid vizualization issues. Also, we should mention that work time is measured in seconds here because of the difference in dataset sizes comparing to *DBLP*.

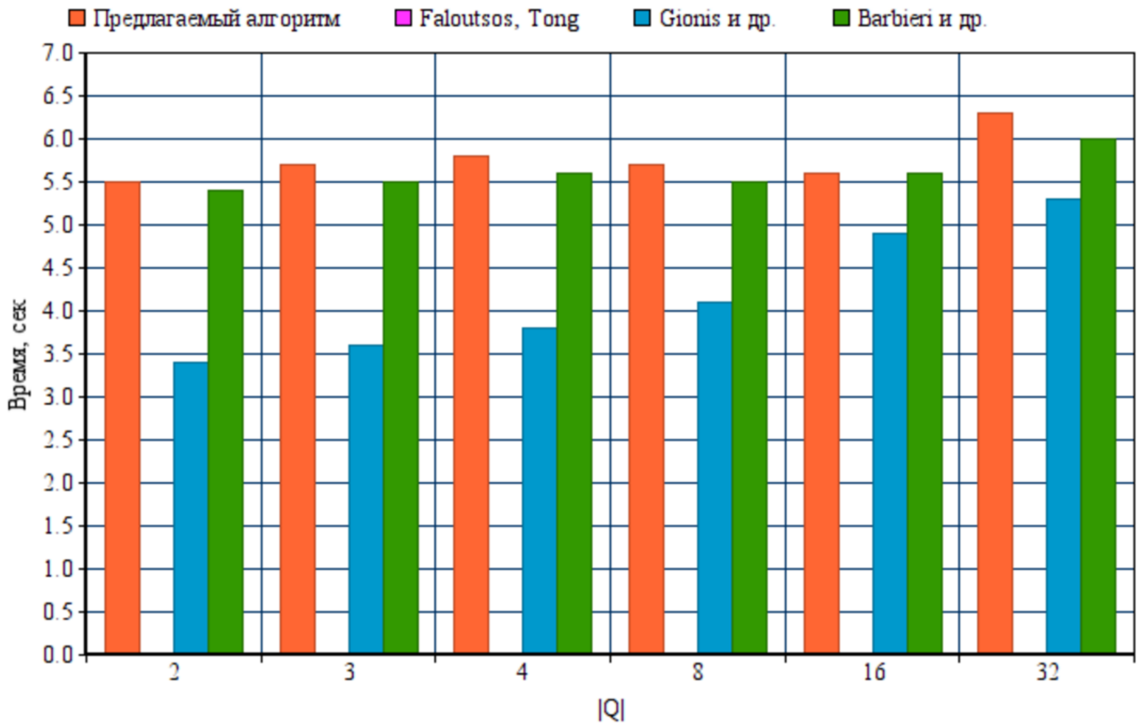
3.2.2.4. Queries with noise. Community density



3.2.2.5. Queries with noise. Community size



3.2.2.6. Queries with noise. Work time



The work time of Faloutsos & Tong [5] is not presented for the same reason.

3.2.3. Final results

Let's collect the results altogether. We collect these results in tables 2 and 3:

In these tables you can find how our solution compares to the other ones. In each cell we provide average result and maximal win respectively. For example,

$18.1\%/54.6\%$ in «density with noise» cell means that in average we won 18.1% by density, but the maximal win among all experiments was 54.6%. In size and work time $-4.7\%/-14.6\%$ means that our size is 4.7% less in average and 14.6% less on the best experiment.

Table 2 – DBLP dataset

–	Faloutsos & Tong	Gionis et al.	Barbieri et al.
Density	+18.2% / +80.0%	+200.6% / +542.9%	4.2% / +12.5%
Size	-4.5% / -14.4%	+97.4% / +17.7%	-2.9% / -6.25%
Time	-80.3% / -83.3%	+11.7% / +7.7%	+5.2% / +2.7%
Density (noise)	+18.1% / +54.6%	+210.4% / +844.4%	+37.6% / +112.5%
Size (noise)	-4.7% / -14.6%	+247.9% / +87.5%	-11.6% / -22.4%
Time (noise)	-82.8% / -84.6%	+12.1% / +10.3%	+8.4% / +6.6%

Table 3 – Youtube dataset

–	Faloutsos & Tong	Gionis et al.	Barbieri et al.
Density	+23.3% / +75.0%	+305.4% / +775.0%	+6.5% / +16.7%
Size	-4.6% / -13.7%	+123.3% / +27.9%	-2.6% / -5.3%
Time	-320.8% / -325.1%	+39.6% / +15.1%	+3.6% / +1.8%
Density (noise)	+24.7% / +69.2%	+252.1% / +685.7%	+45.3% / +129.2%
Size (noise)	-4.3% / -17.0%	+223.3% / +27.8%	-14.8% / -23.7%
Time (noise)	-323.4% / -331.1%	+40.8% / +14.2%	+2.9% / +0.0%

We're also worried about the following thing — is it possible that on queries without any noise our algorithm will throw away some vertices thinking that it's a noise? Our experiments showed that it is possible, but actually because the query doesn't contain any noise, its vertices are densely connected and our algorithm will find almost no noise, only few vertices. Results have shown that we throw away only $2-3$ vertices from Q (if $|Q|=32$), which is quite normal behaviour.

CONCLUSION

By given graph G and a set of query vertices $Q \subset V(G)$ in this article we're solving the task of finding the community containing all or most of the vertices from Q. As we proved before, the task is very actual nowdays and can be applied in many fields.

In this work we try to find *k-core* with maximal k of the minimal size containing most of the vertices from Q. We proof that the task is NP-hard and describe some heuristics for its solution. Experiment results held on the real data show us that on the queries without any noise our suggested solutions works almost the same as the best existing solution. However, after adding a noise to the query, solutions of other authors start to find non-optimal big subgraph when our solution finds densely connected small subgraphs.

Also, the suggested algorithm is backward compatible — we can add parameter $minSize$ which will be equal to the minimal number of query vertices in the answer. This parameter won't affect much on our solution, but will help us to find subgraph containing all query vertices.

In future we plan to spread the suggested idea on weighted graph and maybe some other kind of graphs — attibuted graphs, multigraphs, etc.

LITERATURE

1 *Newman M.* Fast algorithm for detecting community structure in networks. — 2004.

2 *Newman M.* Finding community structure in networks using the eigenvectors of matrices // Phys. Rev. E 74. — 2006. — Vol. 74.

3 *Fortunato S.* Community detection in graphs. — 2010.

4 Online search of overlapping communities / W. Cui [et al.] // Proceeding SIGMOD '13 Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. — 2013. — Vol. 978. — P. 277–288.

5 *Faloutsos C.*, *Tong H.* Center-Piece Subgraphs: Problem Definition and Fast Solutions // Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and Data Mining. — 2006. — Vol. 1. — P. 404–413.

6 The Minimum Wiener Connector Problem / N. Ruchansky [et al.] // Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. — 2015. — Vol. 1. — P. 1587–1602.

7 Approximate closest community search in networks / X. Huang [et al.] // Proceedings of the VLDB Endowment. — 2015. — Vol. 9. — P. 276–287.

8 Efficient and effective community search / N. Barbieri [et al.] // Data Mining and Knowledge Discovery. — 2015. — Vol. 29. — P. 1406–1433.

9 Mining Connection Pathways for Marked Nodes in Large Graphs / L. Akoglu [et al.]. — 2013.

10 *Gionis A.*, *Mathioudakis M.*, *Ukkonen A.* Bump hunting in the dark: Local discrepancy maximization on graphs // 2015 IEEE 31st International Conference on Data Engineering (ICDE). — 2015. — Vol. 978. — P. 1155–1166.

11 *Faloutsos C.*, *McCurley K. S.*, *Tomkins A.* Fast discovery of connection subgraphs // Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. — 2004. — Vol. 1. — P. 118–127.

12 *Sozio M.*, *Gionis A.* The community-search problem and how to plan a successful cocktail party // Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining. — 2010. — Vol. 978. — P. 939–948.

13 *Zhu L.*, *Ng W. K.*, *Cheng J.* Structure and attribute index for approximate graph matching in large graphs // Journal Information Systems. — 2011. — Vol. 36. — P. 958–972.

14 Robust local community detection: on free rider effect and its elimination / Y. Wu [et al.] // Journal Proceedings of the VLDB Endowment. — 2015. — Vol. 8. — P. 798–809.

15 Local search of communities in large graphs / W. Cui [et al.] // Proceeding SIGMOD ’14 Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. — 2014. — Vol. 978. — P. 991–1002.

16 As Strong as the Weakest Link: Mining Diverse Cliques in Weighted Graphs / P. Bogdanov [et al.] // ECML PKDD 2013 Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases. — 2013. — Vol. 8188. — P. 525–540.

17 *Kou L.*, *Markowsky G.*, *Berman L.* A fast algorithm for Steiner trees // Journal Acta Informatica. — 1981. — Vol. 15. — P. 141–145.