Andrei Eremeev

THE SPANNING TREE BASED APPROACH FOR SOLVING THE SHORTEST PATH PROBLEM IN SOCIAL GRAPHS

JYVÄSKYLÄN YLIOPISTO
TIETOJENKÄSITTELYTIETEIDEN LAITOS
2016

TIIVISTELMÄ

Eremeev, Andrei
Virittävän puun käyttö ratkaisuna sosiaalisen graafin lyhimmän polun ongelmaan
Jyväskylä: Jyväskylän yliopisto, 2016, 70 s.
Ohjelmistotuotanto, pro gradu – tutkielma
Ohjaajat: Semenov, Alexander; Korneev, Georgiy.

Tämä pro-gradu tutkielma käsittelee lyhimmän polun ongelmaa sosiaalisia verkostoja mallintavissa sosiaalisissa graafeissa. Tämän työn kohteena ovat sosiaalisen median sivustot, joissa kullakin käyttäjällä on profiili ja käyttäjät voivat olla esimerkiksi toistensa ystäviä. Sivustoa mallintavan sosiaalisen graafin solmut mallintavat näitä profiileja ja suuntaamattomat kaaret profiilien välisiä ystävyyssuhteita. Laajemmin tällaisia graafeja käytetään esim. vaalien tulosten ennustamiseen, tai suosittelujärjestelmissä suositusten koostamiseen. Monet sosiaaliseen graafin ominaisuudet vaativat etsimään polkujoukkoja eri solmujen ja solmuryhmien välillä. Sosiaalisen graafin analyysi vaatii usein laskemaan paljon lyhimpiäpolkuja kahden solmun välillä. Tätä tarvitaan esimerkiksi määritettäessä solmun polkukeskeisyyttä. Työn keskeisenä tavoitteena on kehittää lyhimmän polun etsintään tehokas yhdistelmäalgoritmi. Työssä esitellään ensin sosiaalisten graafien ominaisuuksia. Tämän jälkeen esitellään keskeiset tunnetut lyhintä polkua etsivät algoritmit, jotka vastaavat luotua vaatimusmäärittelyä. Työn tuloksena esitetään tehokas algoritmi, joka perustuu Atlas-algoritmiin ja joka kattaa myös muiden esiteltyjen algoritmien toiminnallisuuden. Opinnnäyte kertoo myös miten algoritmi toteutetaan *Java*-kielellä tehokkaasti. Kehitetty algoritmi on käyttöönottovaihessa Odnoklassniki - nimisellä sosiaalisen median sivustolla, jolla toimii venäjänkielinen verkkoyhteisö. Ko. sivustolla on kaikkiaan 205 miljoonaa käyttäjää ja 44 miljoonaa kävijää päivässä (se on kahdeksanneksi suosituin sivusto Venäjällä ja entisen Neuvostoliiton tasavalloissa). Ehdotettu algoritmi ratkaisee lyhimmän polun ongelman eo. sivustosta muodostetussa sosiaalisessa graafissa suorituskykyisesti vasteajan (50 ms per kysely), muistin käytön (alle 15 GBs ensisijaisen muistin) ja saavutetun tarkkuuden (yli 90%) suhteen. Algoritmi tukee myös dynaamisia sosiaalisia graafeja.

Avainsanat: sosiaalinen graafi, sosiaalisten verkostojen analyysi, lyhimmän polun ongelma, Odnoklassniki, *Atlas* algoritmi.

ABSTRACT

Eremeev, Andrei
The spanning tree based approach for solving the shortest path problem in social graphs
Jyväskylä: University of Jyväskylä, 2016, 70 p.
Software Engineering, Master's thesis
Supervisors: Semenov, Alexander; Korneev, Georgiy.

This thesis is devoted to the shortest path problem in social graphs. Social graphs represent individuals and social relationships between them. As for social networking sites, their users are represented as vertices of the social graph, and the relationship which indicates whether two users are friends in the social networking site are represented as edges of the social graph. Therefore, social graphs are widely investigated by sociologists in order to determine rules and properties of various social processes. Analysis of such social graphs may be used in prediction of results of election, or recommendation systems. Calculation of many social graph metrics requires computation of shortest paths between vertices of the social graph. Often, analysis of social graphs requires calculation of plenty of shortest paths, for instance, paths between each pair of vertices. Searching of plenty of shortest paths is needed in calculation of betweenness centrality of a vertex. The goal of the Master's thesis is to synthesis an efficient shortest path searching algorithm. First, characteristics of social graphs are reviewed; thereafter, existing shortest path searching algorithms are reviewed based on defined requirements. Then, an efficient algorithm which is based on the *Atlas* algorithm, one of the existing algorithms, is synthesized. The Master's thesis also tells how to implement the algorithm in *Java* more efficiently. The developed algorithm is under deployment into the Odnoklassniki social networking site, a Russian social networking site, which contains 205 million of users and 44 million of visitors per day (the eight most visited site in Russia and former Soviet Republics). The proposed algorithm solves the shortest path problem in social graphs with acceptable performance (50 ms per query), memory usage (less than 15 GB of the primary memory) and applicable accuracy (more than 90%). Also, the algorithm supports dynamic social graphs.

Keywords: social graph, social network analysis, shortest path problem, Odnoklassniki, the *Atlas* algorithm.

FIGURES

TABLES

TABLE OF CONTENTS

LIST OF TERMS WITH DEFINITIONS

Graph is an ordered pair (V, E) comprising a finite nonempty set V of *vertices* (points) and together with a set E of *edges* (lines), which is a subset of Cartesian product of the set of vertices. Vertices may represent some objects; edges may represent relations between objects.

Social network is a social structure made of a set of individuals and a set of ties between the individuals. Ties can represent friendship, acquaintance, kinship or disease transmission. Social networks can be modeled as graphs in which vertices represent individuals and edges represent ties between the individuals. Graphs which represent social networks are named *social graphs*.

Social networking site is a platform to build social networks or social relations among people who share interests, activities or real-life connections.

Social network analysis is a strategy for investigating social structures through the use of social and graph theories.

Path (walk) in a graph can be defined as a finite sequence of vertices and edges $v_0 e_1 \ldots v_k$ in which each edge connects the preceding and following vertices, so $e_i = (v_{i-1}, v_i)$.

Weighted graph is a graph with weight function which assigns a real-value to each edge.

A graph, which has a weighted function which returns one for all edges, is called an *unweighted graph.*

The *length of a path* in an unweighted graph is the number of edges which comprise the path.

In a weighted graph the *length of a path* is the sum of the weights of edges which belong to the path.

The *shortest path* between a pair of vertices is the path where the length of the path between these vertices is minimized.

A *heuristic algorithm* is an algorithm which produces a solution in a reasonable time frame that is good enough for solving the problem. The solution may simply approximate the exact solution, but it is valuable because finding it does not require significant time.

1 INTRODUCTION

The emergence of online social networking sites is changing the investigation of the structure of human relationships. Social network analysis has gained a significant popularity in computer science, political science, communication studies and biology. Individuals bring their social relationships to online social networking sites and make previously invisible social structures be explored to determine social processes. Social networks can be modeled as graphs; hence, the methods of graph theory can be applied for analysis of social networks. The methods of the graph theory can be used to investigate kinship patterns, community structures, information diffusion and many other problems (Marcus, Moy, & Coffman, 2007).

Additionally, information left by users on social networking sites can be used, for instance, in predicting the results of elections (Wang, Can, Kazemzadeh, Bar, & Narayanan, 2012; Tumasjan, Sprenger, Sandner, & Welpe, 2010). Also, social networks analysis is used to identify money laundering and terrorists (Zhang, Salerno, & Yu, 2003). Moreover, social networks were broadly used in organizing mass riots and violence during the Arab Spring (Semenov, 2013). The National Security Agency (NSA) has been performing analysis of call records, since the September 11 attacks, and analysis of collected Internet communications since 2007, known as surveillance program PRISM (Greenwald & MacAskill, 2013).

Some of the problems which need to be solved during data aggregation and analysis require large numbers of shortest path computations between two vertices in a graph. These problems involve calculations of such metrics as betweenness centrality, closeness centrality and others (Freeman, Roeder, & Mulholland, 1980). Thus, the shortest path problem is one of the basic problems which are used in the analysis of graph structure. The problem is comprised of finding the sequence of vertices, a path, which joins a pair of vertices in a graph in such a way that the number of edges on the path is minimized. Many shortest path algorithms have been developed, however they do not perform well on large graphs.

The goal of this work is to synthesize an algorithm that is able to solve the shortest path problem in large social graphs with acceptable accuracy, performance and memory usage. Also, social graphs are very dynamic. Indeed, changes in the friend lists of users are very frequent (Wilson, Boe, Sala, Puttaswamy, & Zhao, 2009). Thus, the algorithm should be able to handle dynamics of social graphs.

1.1 Motivation for the research

Social networking sites provide various services to entertain users. Users are creating their profiles, fill them with various personal data (name, age, pictures and others) and make connections between each other. Other services may include capability to watch online videos (Youtube, VK, Odnoklassniki), listen to music (VK, Spotify), develop and play games (VK, Odnoklassniki) and many others. Odnoklassniki ("Classmates" in English), a Russian social network site, has 205 million users and 44 million visitors per day in spring 2015. Currently Odnoklassniki intends to add a service which can find the shortest path between a pair of users while spending a reasonable time for calculation. Odnoklassniki has requested the development of the algorithm on which the mentioned service will be based. User A may select another user B and get the shortest path to user B via friends, friend-of-friends, and etc. Additionally, the approach can be used in data aggregation and data analysis of the Odnoklassniki social network. The solution should be fast, because users expect that a service responds quickly, and allows for serving hundreds of requests simultaneously. The algorithm should solve the shortest path problem in less than 50 ms per query. The proposed algorithm should be accurate as well. The error rate, the rate of that the found path is not the shortest one, should be less than 10%. Also, if the algorithm makes a mistake, then the length of the returned result should not be longer than the length of a correct (shortest) path plus one. The algorithm can be used in graph analysis. However, these kinds of mistakes lead to incorrect statistics. Furthermore, the algorithm should not make mistakes in the case of short paths (less than three edges), because if the algorithm is deployed as a standalone service, results of the algorithm can be easily checked by the users for short paths. Hence, if a user realizes that the algorithm returns wrong results, then it could lead to lowering the prestige of the social networking site. As for longer paths (more than three edges), vertices locating on the distance less than three edges from any vertex comprise a half of the vertices of the whole graph (Ugander, Karrer, Backstrom, & Marlow, 2011). Thus, longer than three edges paths cannot be checked by the users easily. Another metric which is taken into account is the usage of both the heap and the disk memory. The API of Odnoklassniki has been written in *Java*; thus, the algorithm should be implemented in *Java*. The usage of the heap memory should be restricted by 1 GB, since wasteful usage of the *Java* heap memory could lead to large garbage collector pauses after several executions of the algorithm. Additionally, heuris-

tic algorithms usually have a pre-computation step which pre-calculates some data. It is supposed that the implementation of the algorithm uses memory mapping I/O (Pai, Druschel, & Zwaenepoel, 1999); in other words, all of the pre-computed data should be in the primary memory in order to increase the speed of the algorithm. Thus, the amount of pre-computed data should not exceed the volume of the primary memory of a machine. The algorithm is assumed to be run on a machine with 64 GB of the primary memory. Thus, the algorithm cannot use more than 64 GB of the primary memory to avoid swapping of memory pages (Bach, 1986).

The shortest path problem is broadly explored nowadays for small graphs. There are algorithms which solve this task, like the breadth-first search (Lee, 1961). However, application of these algorithms to large graphs requires a lot of resources. For instance, the breadth-first search requires at least 1.5 GB (200 000 000 * 8 bytes) of the primary memory to store all vertices, each vertex is represented by an 8 bytes long integer, in its inner queue of a graph which contains 200 million vertices. According to the performance estimation (Potamias, Bonchi, Castillo, & Gionis, 2009), it takes roughly a minute in a standard desktop computer to calculate the shortest path using the breadth-first search between two vertices in a graph that contains four million vertices and 50 million edges. While one of the most popular social networking sites, Facebook, has circa one and a half billion active users (Statista, 2015).

Researchers have suggested several approaches that are able to handle large graphs. Algorithms, like *A** (Hart, Nilsson, & Raphael, 1968) or contraction hierarchies (Geisberger, Sanders, Schultes, & Delling, 2008), can solve the shortest path problem for large road graphs in an acceptable time. However, the problem is not broadly explored for social graphs. Algorithms, like landmark-based algorithms (Kleinberg, Slivkins, & Wexler, 2004; Potamias, Bonchi, Castillo, & Gionis, 2009; Zhao, Sala, Wilson, Zheng, & Zhao, 2010), which extract a set of vertices, called landmarks, can only estimate the shortest distance between two vertices, but not find the actual path. In addition, the algorithms contain a pre-computation step; thus, once the graph changes, the pre-computation should be performed again. According to (Wilson, Boe, Sala, Puttaswamy, & Zhao, 2009), 50% of user actions per day are actions related to adding and removing friends. Thus, repeatedly performed pre-computation step may take significant resources in a social graph context.

Thus, one can argue that there is no acceptable algorithm which can solve the shortest path problem for large social networks with acceptable accuracy, memory usage and performance.

1.2 Objectives

The shortest path problem is a basis problem which is used in graph analysis. The shortest path problem can be defined as searching the path, which contains the set of edges with the minimized sum of edges' weights, between the men-

tioned vertices. Fig. 1 depicts a graph and the shortest path between vertices u and v. The path is marked with blue color.

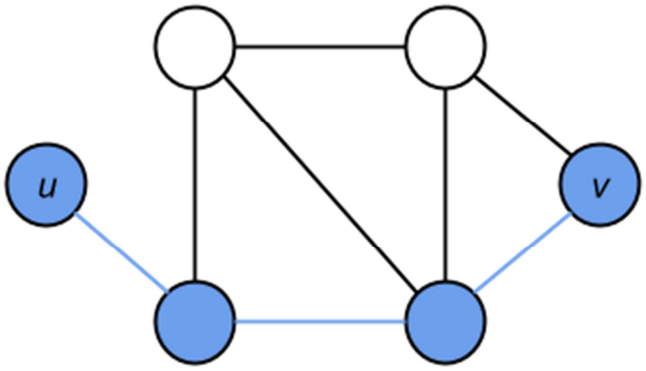

FIGURE 1 The shortest path between two vertices in the graph

Several variations of the shortest path problem can be formulated (Cormen, Leiserson, Rivest, & Stein, 2001):

- *single-pair shortest path problem*, in which the shortest path(s) between two vertices are found;
- *single-source shortest path problem*, in which shortest paths from all vertices to all other vertices in the graph are found;
- *single-destination shortest path problem*, in which shortest paths from all vertices to a single destination vertex are found;
- *all-pairs shortest path problem*, in which shortest paths between every pairs of vertices in the graph are found.

The single-destination shortest path problem can be reduced to the single-source shortest path problem by reversing arcs in a directed graph and solving the single-source shortest path problem in the built graph. The single-pair shortest path problem is a particular case of the single-shortest path problem.

The main objective of this thesis is to synthesize and to implement an algorithm which efficiently solves the single-pair shortest path problem with acceptable accuracy, performance and memory usage. Heuristic algorithms which are based on some properties of models, in this case social graphs, try to reduce the space of possible solutions. Therefore, characteristics of social graphs should be provided and analyzed in order to remove irrelevant vertices and edges from the set of possible solutions while searching for the shortest path. The current Master's thesis formulated the requirements of the algorithm for review and analysis of existing algorithms. According to the requirements, existing shortest path searching algorithms were reviewed and analyzed. According to the analysis of the algorithms, the most acceptable algorithm, *Atlas,* was chosen. The new algorithm, *Atlas+*, was synthesized based on the *Atlas* algorithm which uses a set of spanning trees to approximate paths in social graphs. As it is shown in the paper written by (Cao, Zhao, Zheng, & Zhao, 2013) the *Atlas* algorithm shows acceptable accuracy and performance in some applications, like

ranked social search (searching for top-*k* closest vertices of a set of vertices to a vertex), but the accuracy of the algorithm is circa 25-30%, which is not acceptable.

Shortly, there are the following objectives:

1. Review of characteristics of social graphs;
2. Review of existing precise and heuristic algorithms;
3. Synthesizing, implementation and evaluation of an algorithm which is able to solve the shortest path problem in large social graphs more efficiently than existing ones.

1.3 Summary of the results

Overall, the following results have been achieved:

1. Characteristics of social graphs modelled by social network sites have been reviewed;
2. Existing shortest path searching algorithms have been reviewed and have been analyzed;
3. A new shortest path searching algorithm has been synthesized, implemented and evaluated on real social graphs extracted from social networks.

2 RESEARCH PROBLEM AND METHODOLOGY

The following section defines the research questions of the Master's thesis, describes the research method which is used and how it is applied in the Master's thesis.

2.1 Research questions

Heuristic approaches are usually based on some assumptions or characteristics of the model of a real world phenomenon, in the current Master's thesis social graph. For instance, roads can be modeled as graphs with road crossings as vertices and roads between them as edges. Roads graphs are planar, and the planarity of road graphs is utilized in the *A** algorithm. Hence, **the first research question** arises:

Which characteristics of the model, social graph, can be used in the development of the new algorithm?

Now the area of algorithms which solve the shortest path problem for small graphs, that contain less than one million vertices, is broadly explored, but these algorithms cannot be applied to large graphs, because it may take too much resource to process data and too much time to get the answer.

As it is said in the introduction, Odnoklassniki intends to start a new service which allows users to find a path to some desired other users. In addition, the solution may be used in further analysis of the Odnoklassniki social networking site. To determine whether the algorithm is designed in an efficient way, the requirements of the algorithm should be defined. The requirements of the synthesized algorithm are based on the requests of the Odnoklassniki social networking site and limits of hardware and software. In addition, existing algorithms should be reviewed in order to determine whether there is an acceptable algorithm or there is an algorithm which solves the problem inefficiently, but it can be enhanced. The efficiency of the algorithms is measured according to the

defined requirements. Thus, **the second research question** of the Master's thesis is:

Which existing algorithms can be used to solve the shortest path problem in social graphs? The algorithms should fit the defined requirements.

Since, as it is shown in Section 4, existing algorithms do not fit the requirements, t**he third research question** would be:

How to design the shortest path searching algorithm which fits the formulated requirements?

As stated above, users in social networks are allowed to create relationships between each other as well as destroy them. According to (Wilson, Boe, Sala, Puttaswamy, & Zhao, 2009) 50% of user actions per day relates to changing in users' friends lists. Some of the algorithms (Cao, Zhao, Zheng, & Zhao, 2013; Zhao, Sala, Wilson, Zheng, & Zhao, 2010) use some pre-computed data which requires a lot of resources and time for rebuilding in case of changes in graphs. Therefore, a **sub-task of the third research question** appears:

What is the impact of changes in a social graph to accuracy of the synthesized algorithm? How can the algorithm be modified to support dynamic social graphs?

2.2 Research design

The following section describes the methodology of the research. To answer the research questions mentioned above, two research methodologies are utilized. The first one is the literature review (Creswell, 2007). It involves review of the graph theory, characteristics of social graphs and analysis of existing shortest path searching algorithms. The literature review is done in Sections 3 and 4. Additionally, the reviewed algorithms are analyzed according to the requirements defined in Section 4.2. Thus, the first and the second research questions are answered.

The following is the description of the design science research methodology (DSRM) (Hevner, March, Park, & Ram, 2004; Hevner & Chatterjee, 2010). According to the DSRM framework, the research is guided by *business needs* and *applicable knowledge*. Applicable knowledge involves such foundations and methodologies as theories, models, methods and data analysis. Regarding business needs, the algorithm can be employed as part of a shortest path searching service deployed into a social networking site. The research process comprises of two phases: the development phase, during which a new artifact is created, and the evaluation phase, during which the new artifact is evaluated. Usually, the evaluation follows development, but then the further development might be needed in order to fix defects of the artifact. Thereafter, the research process

may continue with further development and further evaluation. The two phases of the design science approach are as follows:

- During the development phase, a new shortest path searching algorithm is created. Firstly, after reviewing of existing solutions, the most acceptable (*Atlas*) is chosen, is implemented and analyzed. Thereafter, the chosen algorithm is enhanced based on the characteristics of social graphs.
- During the evaluation phase, the proposed algorithm is evaluated on paths pre-calculated by the breadth-first search. Several thousand of paths are computed. Thereafter, the new algorithm is run on the correct paths and the output of the algorithm is compared with the expected paths. Performance, accuracy and memory usage of the algorithm are measured.

The rest of the Master's thesis (Sections 5-7) is devoted to synthesizing and evaluation of the algorithm. Sections 5-7 answer the third research question.

The DSRM is used because the research conforms the following guidelines suggested by (Hevner, March, Park, & Ram, 2004):

- *Design-science research must produce a viable artifact.* Indeed, a new artifact is expected to be produced by the research. The new artifact is a new shortest path searching algorithm which can handle large social graphs.
- *Development of technology-based solutions to important and relevant business problems.* The solution can be used as the base of a service which calculates the shortest path between a pair of users of a social networking site.
- *Evaluation of the artifact utility, quality and efficacy must be rigorously demonstrated via well-executed evaluation methods.* It is proposed to use paths precomputed by BFS in order to run the new algorithm on them and calculate accuracy of the new algorithm. Additionally, memory usage and performance are measured to check that the algorithm fit the requirements.
- *Clear and verifiable contributions in the area of design artifact and its foundations.* It is assumed that the first contribution, the review of algorithms and characteristics of social graphs, will be used in another research, and the second contribution, a new algorithm, will be used in services of social networking sites as well as the base of tools which analyze social graphs. Moreover, the algorithm may be published as a scientific paper, thus, scientific community would be able to use or improve it.
- *Application of rigorous methods for research for both construction and evaluation of artifact.* The artifact construction is based on published research from academic journals. While for the artifact evaluation, the rigor is kept because algorithm evaluation is based on pre-defined measures, like performance, accuracy and memory usage, which are compared between different modifications of the new approach.

- *The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.* The research process is defined in this thesis. As it is mentioned in the introduction and in the current section, the elicitation of requirement is performed firstly; thereafter, existing solutions are reviewed and the most acceptable algorithm is chosen. The new algorithm is created on the basis of this algorithm. Finally, the new algorithm is evaluated as it is mentioned in the previous bullet-point.
- *Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.* The outcome of the research will be published as the Master's thesis; therefore it can be accessed by scientists in the research area. Additionally, the artifact can be published in a paper. Regarding management-oriented audiences, the algorithm can be embedded into a service which calculates the shortest path between a pair of users of a social networking site.

Software development is conducted according to one of the following basic software development models or their modifications:

- The *waterfall model* which is comprised of a sequence of processes from specification to implementation and deployment (Benington, 1983).
- The *iterative model* in which development is done as a sequence of iterations in which each iteration contains a stage of planning, implementation, testing and analyzing (Larman & Basili, 2003).
- The *spiral model* which is based on looped improvement of the project goal; in the spiral approach new elements of product are added when they become known (Boehm, 1986).

In this Master's thesis software development is conducted according to the spiral model, since the development contains several cycles which contains analysis, development and evaluation. The spiral of the traversed research phases is shown in Fig. 2. First, the domain of research was grasped and algorithm requirements were elicited (Sections 3-4); thereafter, the first version of the algorithm was implemented (Section 5). After the analysis of bottlenecks of the first solution, a new open-addressing hash map was implemented and added to the algorithm (Section 5.4). As it is shown further, the first version of the synthesized algorithm does not conform to the performance constraints (Section 5.5). Therefore, the properties of the algorithm were analyzed, and according to them, the second version of the algorithm was created (Section 6). After that, the algorithm was parallelized, and the hash map was implemented as lock-free (Section 6.2). Thereafter, new bottlenecks related to data communications appeared; the algorithm spent a half of the query time for data transmission through a computer network. Therefore, a new heuristic, which decreases the volume of data transmitted through the network, was suggested. Finally, the *Atlas+* algorithm was evaluated on dynamic graphs. It was shown that *Atlas+* is not impacted by changes in the social graph significantly. Also, two strategies

to handle the dynamics of social graphs were suggested and evaluated (Section 7). It was shown that these strategies are able to keep the accuracy of *Atlas+* on the acceptable level.

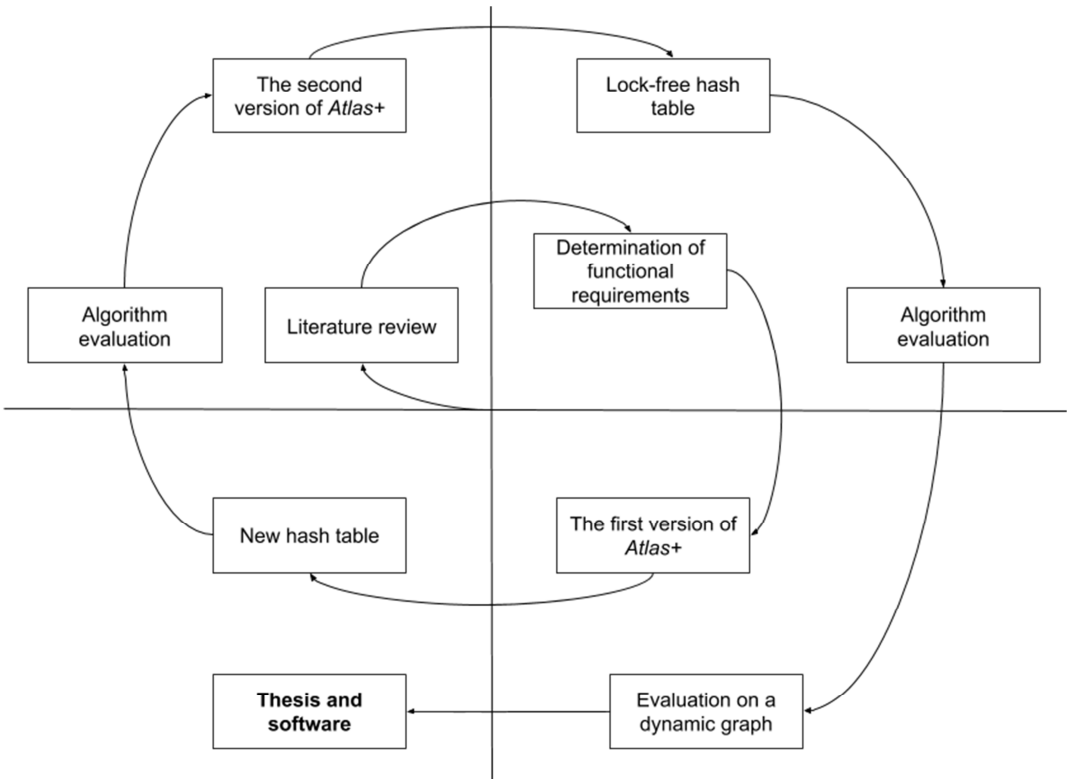

FIGURE 2 Spiral of the traversed research phases

3 GRAPH THEORY

This section provides the basic concepts and terminology of graph theory. The section is comprised of four sub-sections. The first one tells about common set theory and big o-notation. The second section defines such concepts as graphs, paths, connected components and others. The next one provides information about how graphs can be represented in computer memory. And the forth section focuses on social graphs and their characteristics.

3.1 Mathematical preliminaries

This section contains some concepts of mathematics that are needed in this Master's thesis. The section defines big O-notation and provides the basis of set theory.

The common set theory provided in the section is based on the fundamental work of (Cantor, 1877). A *set* is a collection of objects which are called the *elements* of the set. Let S be a *set*, then notation $e \in S$ means that object e is an element of set S. While $e \notin S$ means that e is not an element of set S.

A subset A of a set S, denoted $A \subset S$, is a set for which the following statement is true. Each element of the subset is an element of set S.

Cartesian product of sets A and B, denoted $A \times B$, is a set which contains ordered pairs (a, b) where a is an element of set A and b is an element of set B.

O-notation (big o-notation) describes the limiting behavior of a function when the argument tends to a particular value or infinity (Knuth, 1976). In computer science it is used to classify processing time of algorithms. $O\big(g(n)\big)$ denotes the set of all $f(n)$ such that there exist positive constants M and N such that for all values $n \geq N$, the absolute value of $f(n)$ is less or equals to M multiplied by the absolute value of $g(n)$. In other words, $f(n) \in O\big(g(n)\big)$ <=> $\exists\, M, N : \forall\, n \geq N, |f(n)| \leq M\, |g(n)|$. Thus, the O-notation is used to estimate the behavior function f when it is not necessary to think about what the exact function is. It behaves asymptotically essentially in the same way as a known function g.

3.2 Basic concepts and definitions

This section is based on the fundamental work "Graph theory" published in 1969 by an American mathematician Frank Harary (Harary, 1969).

A *graph* G is an ordered pair (V, E) comprising a finite nonempty set V of *vertices* (points) together with a set E of *edges* (lines), which is a subset of Cartesian product of the set of vertices, i.e. $E \subset V \times V$. Each pair of vertices $e = (u, v) \in E$ is an *edge* and it is said that e connects u and v. Hence, the vertices u and v are *adjacent* vertices. Vertex u and edge e are *incident* with each other; as well as v and e. Moreover, if two distinct edges e and e' are incident with a common vertex, then they are said to be *adjacent* edges. In Fig. 3 an example of a graph is shown.

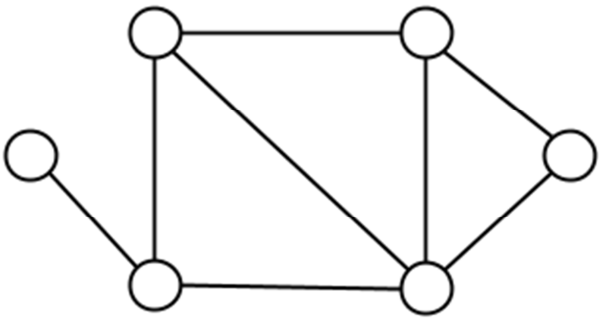

FIGURE 3 An example of a graph

A *directed graph* or *digraph* is a graph which consists of a finite nonempty set V of vertices and a set of ordered pairs which are named directed edges or arcs. An *undirected graph* is a one where for each edge (u, v) it holds that there is an edge (v, u).

A *complete (directed) graph* is a directed graph in which every pair of vertices is connected by a pair of unique edges (one in each direction).

A *planar graph* is a graph that can be embedded in the plane. This means that it can be drawn in such a way that its edges do not intersect each other, the edges can only intersect in their endpoints. Fig. 3 provides a planar graph. In Fig. 4 a graph which cannot be planed is shown.

Another graph that should be mentioned is a *weighted graph*. A weighted graph is a graph with weight function which assigns a real-value to each edge. A graph, which has a weighted function which returns one for all edges, is called an *unweighted graph.*

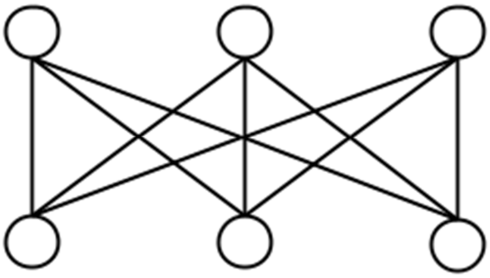

FIGURE 4 An example of a non-planar graph

A *subgraph* of G is a graph G' which has all vertices and edges in G. This means that subgraph's set of vertices is a subset of the set of vertices of graph G, and subgraph's set of edges is a subset of the set of edges of graph G.

A *path (walk)* in a graph can be defined as a finite sequence of vertices and edges $v_0 e_1 \dots v_k$ in which each edge is incident with the preceding and following vertices, so $e_i = (v_{i-1}, v_i)$. The edges can be omitted in the notation, so the path between two vertices can be denoted as $v_0 v\_1 \dots v_k$. The edges are evident by context. If the first and last vertices are the same, i.e. $v_0 = v_k$, then the path is called a *closed path* in a directed graph. A *cycle* in a graph is an equivalence class of closed paths induced by the following equivalence relation: two paths are equivalent if and only if $\exists j \forall i : e_{i \bmod k} = e'_{(i+j) mod k}$ where e_i are edges of the first path and e'_i are edges of the second one. In other words, this definition means that there is such a shift of indices that all edges have the same indices in the two paths.

The *length of a path* in an unweighted graph is the number of edges which comprise the path. In the similar way, the length of the path can be defined for weighted graphs. In a weighted graph the *length of a path* is the sum of weights of edges which belongs to the path. In other words, $l(p) = \sum_{i=1}^{k} w(e_i)$. A *shortest path* between two vertices is a path where the length of path between these vertices is minimized. The *diameter of a graph* is the longest shortest path between any pair of vertices of the graph if the graph is connected. If it is not connected the diameter is infinite.

If each pair of vertices of an undirected graph is connected by a path, then this graph is called *connected*. A *connected component* or simply a *component* is a subgraph of an undirected graph that is connected and maximal in terms of inclusion. Thus, the connected components of an undirected graph are equivalence classes in which pair connectivity is an equivalence relation.

Relying on the definition of cycles and connected components the terms *tree* and *forest* can be defined. A graph is called *acyclic* if it does not have cycles. A *tree* is a connected acyclic graph. Any graph without cycles is a *forest*. Thus, the connected components of a forest are trees. A subgraph G' of a graph G is called a spanning tree if and only if G is a tree and contains all vertices of the graph G. In Fig. 5 an example of a forest with two trees is depicted.

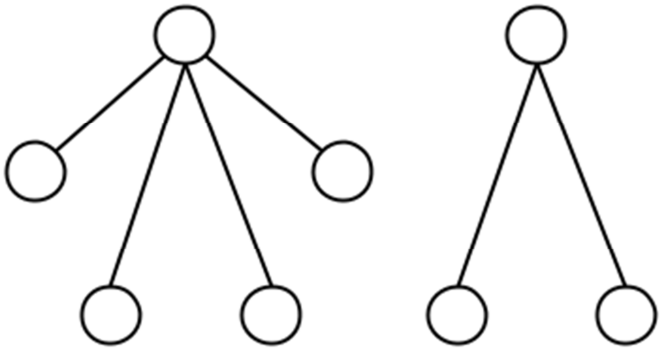

FIGURE 5 An example of a forest with two trees

The *degree* of vertex v, denoted as $\deg v$ or d, is the number of edges which are incident with vertex v. The *in-degree* of vertex v is the number of edges which are incident and end in vertex v. The *out-degree* of vertex v is the number of edges which are incident and begin in vertex v. The latter definitions are relevant for directed graphs.

3.3 Graph representation in computer memory

This section provides information on how graphs can be represented in computer memory. There are two standard ways to represent a graph. The first one is to represent a graph as a collection of adjacency lists and the second one is an adjacency matrix (Cormen, Leiserson, Rivest, & Stein, 2001).

The adjacency-list representation of a graph consists of an array of lists. Each vertex of the graph is represented by an entry in the array. The entry of the vertex is the head of the adjacency list. All vertices which are incident with the vertex are stored in the adjacency list of the vertex.

The adjacency-matrix representation of a graph is assumed to have some numbering function for vertices. Then the adjacency-matrix representation of a graph consists of a matrix $|V| \times |V|$ such that:

$$a_{ij} = \begin{cases} 1, (i,j) \in E \\ 0, (i,j) \notin E \end{cases}.$$

Both of the mentioned representations can be used for directed and undirected graphs. The adjacency-list representation of a graph stores only such edges that are really in the graph. While the adjacency-matrix representation of a graph stores all information about all possible pairs of vertices of the graph. Thus, the adjacency-list representation is more suitable for sparse graphs, in which $|E| \sim O(|V|)$, while the adjacency-matrix representation is more suitable for graphs which are close to complete ones. The adjacency matrix needs only one bit for an edge in case of an unweighted graph, thus, the total volume of adjacency lists which is necessary for their storing is larger than for the appropriate adjacency matrix.

The mentioned ways of graph representation can be adapted to weighted graphs. The node in an adjacency list representing an unweighted graph should also store the weight of the edge. The adjacency matrix can be adapted in the same way. Each entry of the matrix should store the weight of an edge if there is an edge between a pair of vertices. If a graph does not contain an edge, then the appropriate entry should store the special value *NIL*. This value is chosen according to an application. For instance, a road graph does not contain negatively weighted edges, so the *NIL* value can be -1.

Another advantage of the adjacency matrix is constant time $O(1)$ of checking whether there is an edge between a pair of vertices. While the same operation cannot be done faster than $O(n)$ for the adjacency lists where n is the length of the appropriate adjacency list.

3.4 Social graph and its characteristics

A *social graph* is a graph with people as a set of vertices and social relationships between them as edges. These relationships can include such types as friendship, kinship, working at the same work and further relationships (D'Andrea, Ferri, & Grifoni, 2010). Thus, social graph is a mathematical representation of a social network. The social network platform Facebook contains circa one and a half billion active users (Statista, 2015). This means this social networking site has a profile for 20% of the population of the Earth. Thus, Facebook is a good data set in data aggregation and data mining for life modeling, determining community structure. This section is mostly based on the paper written by Ugander et al. (2011). In this paper researchers analyze different characteristics of Facebook.

The *degree of vertices* in an undirected social graph usually has *the power-law distribution*. This means that the probability that the degree of a vertex is k is proportional to k^{-a}, where $a > 1$. Different real-world networks are shown to be power-law distributed. This property is proven for the Internet topology (Faloutsos, Faloutsos, & Faloutsos, 1999), the Web (Barabasi & Albert, 1999), social networks (Adamic, Buyukkokten, & Adar, 2003; Ugander, Karrer, Backstrom, & Marlow, 2011), and neural networks (Braitenberg & Schüz, 1991).

Another class of networks is a *scale-free* network which is a class of networks which have property that a high-degree vertex has the tendency to be connected with other high-degree vertices. This property is discussed in Li et al. (2005). This kind of a network contains a plenty of hubs (high-degree vertices) which are highly connected with each other. Thus, a path between two random vertices often passes through the hubs. Moreover, it is supposed that the network will be splintered into plenty of isolated pieces in case of removal of several hubs (Barabasi & Bonabeau, 2003). The authors announce that social networks are assumed to have the scale-free property. According to Ugander et al. (2011), the major part of individuals in the social network Facebook has less than 200 friends (degree) and median friend count is 99.

Another metric which describes the structure of a social network is *neighborhood function* $N(h)$ (Ugander, Karrer, Backstrom, & Marlow, 2011). This function shows how many pairs of vertices (u, v) such that u is reachable from v along a path in the network with h edges or less. The neighborhood function is more informative than the diameter of a graph, because its value can be large because of a single long path in some region of the graph, while the neighborhood function shows typical distances between vertices. For Facebook this function looks as follows. There are few vertices, circa 0%, which are connected by a two-hop path, 35% of vertices are connected by a four-hop path, and almost all vertices are connected by a six-hop path (Ugander, Karrer, Backstrom, & Marlow, 2011). Also, the average distance between a random pair of vertices was 4.7 in 2011 in Facebook (Ugander, Karrer, Backstrom, & Marlow, 2011).

The characteristics mentioned above do not make sense without analysis of connected component size, because the neighborhood function computation shows distances between vertices only within one connected component. According to Ugander et al. (2011), a social network has one large component which contains most part of vertices (99.91% of the network in Facebook) and plenty of small components. The largest component of small components is comprised of less than 100 vertices.

Thus, the above-mentioned characteristics describe macroscopic view of a social graph. According to the characteristics, a social graph contains one large component where almost all pairs of vertices are connected by a path not longer than six hops. In addition, according to the scale-free property, a path between two vertices usually goes through hubs of the social network.

The *neighborhood graph* of a vertex is a subgraph which is comprised of the adjacent vertices of the vertex and edges between them. So the *local clustering coefficient* of a vertex is a metric which equals to a number of edges in the neighborhood graph, which is divided by $k(k-1)/2$, where k is the degree of the user. Ugander et al. (2011) analyze how the local clustering coefficient depends on the degree. In their work the local clustering coefficient of vertices that have circa 100 adjacent vertices is 14%. This means that 14% of the neighbors of the vertex are connected by edges. Moreover, Ugander et al. (2011) show that the local clustering coefficient decreases with degree. For instance, the clustering coefficient is low for vertices with degree close to 5000.

To describe the sparsity of the neighborhood graph, term *degeneracy* is used. This metric is defined as follows. k*-degeneracy* of an undirected graph G is the largest k for which G has a non-empty k*-core*, where k*-core* of a graph G is the maximal subgraph of G in which all vertices have degree of at least k. Ugander et al. (2011) determine that average degeneracy is an increasing function of user degree within the neighborhood graphs. This can be supposed as the more friends a user has, the larger dense community a user are embedded within. For instance, in Facebook users, which have 500 friends, have average degeneracy 53. This means that they have at least 54 friends who know at least 53 of other friends (Ugander, Karrer, Backstrom, & Marlow, 2011).

Another important property of graphs which should be taken into account in algorithm design is the number of vertices which can be reached in two hops from an initial vertex. Ugander et al. (2011) count two metrics: non-unique friend-of-friends and unique friend-of-friends. Based on the obtained estimations, the number of non-unique friend-of-friends is proportional to k^2, while the number of unique friend-of-friends is proportional to a linear function of degree k.

The main characteristic that should be taken into account during algorithm design is that social graphs are very dynamic. Wilson et al. (2009) have measured how many actions users do per day. These actions include adding and removing friend, commenting and writing posts, and status changes. According to their work user do circa 50% of actions related to changing of the structure of a social graph.

4 SHORTEST PATH SEARCHING ALGORITHMS

This section describes algorithms which are able to solve the shortest path problem. Also, the functional requirements of the desired algorithm are formulated. The described algorithms are analyzed and summarized in the last sub-section of this section according to the functional requirements.

4.1 Algorithms for the shortest path problem

The breadth-first search (BFS) (Lee, 1961) is one of the simplest algorithms for exploring a graph. The algorithm calculates the distance, the number of edges on the path, to each vertex reachable from s and solves the single-source shortest path problem. It works as follows. Given an unweighted graph $G = (V, E)$ and a source vertex s, BFS explores a graph layer by layer to find every vertex that is reachable from the source vertex. BFS is named in such way, because it discovers all vertices at distance k from the source vertex s, and, thereafter, begins discovering the next layer of vertices at distance $k + 1$. It also builds a BFS tree with s as a root of the tree. The tree contains all vertices reachable from s. For any vertices which are in a BFS tree, the shortest path from the source vertex s to any vertex is the path in a BFS tree. The algorithm works on both directed and undirected graphs. It is supposed that the set of black vertices contains already processed vertices, the set of gray vertices contains vertices currently processed, and the set of white vertices contains vertices which have not been discovered at the current step of algorithm. The gray vertices are stored in a first-in-first-out (FIFO) queue. At each step of the algorithm, vertices are divided into three sets: white, gray and black vertices. The algorithm starts from the source vertex s, which is marked by gray color and is added to the queue. Thereafter, BFS dequeues the first vertex of the queue u and looks through the adjacent vertices of the vertex. Each adjacent vertex v, which is still white, is marked by gray color and is added to the queue. The vertex v is added to the BFS tree as a child of the vertex u. When the adjacency list of the current vertex

is processed, the current vertex is marked by black color. The algorithm stops when there are no vertices in the queue. Since the algorithm needs to look through all edges and vertices, the total running time is $O(|V| + |E|)$. The space complexity of BFS is $O(|V|)$ (Lee, 1961).

The Dijkstra's algorithm (Dijkstra, 1959) is supposed to be a generalization of BFS for non-negative graphs, solving the single-source shortest path problem. The vertices are stored in a priority queue, along with the distance to them from the source vertex. As in BFS, the algorithm starts from the source vertex, adds it to the priority queue. Each iteration begins with dequeuing of the vertices with the minimal distance from the source vertex. Thereafter, the adjacent list of the vertex is looked through and distance is updated. The algorithm stops when all edges of the graph are processed. If the algorithm uses Fibonacci's heaps (Fredman & Tarjan, 1987) as the priority queue, the time complexity of the algorithm is $O(|E| + |V| \log |V|)$. The space complexity of the algorithm is $O(|V|)$. The Dijkstra's algorithm can be only applied for a graph with non-negative edges, edges with non-negative weight.

The Ford-Bellman (Bellman, 1956; Ford, 1956) algorithm can solve the single-source shortest path problem for any graph that does not have a negative cycle, a cycle in which each closed path has negative weight. At the first iteration of the algorithm all distances are overestimated, equal to infinity. At each iteration the Ford-Bellman algorithm checks whether there is an edge which can shorten the calculated shortest path to vertices. If the edge exist, then distances are recalculated. After $V - 1$ iterations are done, the algorithm returns an array of distances to the reachable vertices. The Ford-Bellman algorithm has time complexity $O(|V||E|)$. The space complexity of the algorithm is $O(|V|)$.

The idea of the Floyd-Warshall algorithm (Floyd, 1962; Warshall, 1962), which solves the all-pairs shortest path problem, is the same as the Ford-Bellman algorithm. It compares all possible paths through the graph between each pair of vertices. It can be done with $O(|V|^3)$ comparisons in a graph. Thus, the algorithm can find the shortest path matrix between all pairs of vertices of a graph in time $O(|V|^3)$ and needs $O(|V|^2)$ of memory. The matrix can be calculated for any graph without a cycle also with negative weight.

The Johnson's algorithm (Johnson, 1977) uses both the Ford-Bellman algorithm and the Dijkstra's algorithm and it works for any graph without a negative cycle. So, first of all, the Johnson's algorithm adds a dummy vertex to the graph and edges with zero weight from the dummy vertex to all other vertices. After that, the Ford-Bellman algorithm is run from the dummy vertex. The calculated distances are used in reweighing of the edges to make them non-negative. Thereafter, the graph without non-negative edges is built and the Dijkstra's algorithm is run from each vertex of the graph. Thus, the total running time of the Johnson's algorithm is $O(|V|^2 \log |V| + |V||E|)$ if the Dijkstra's algorithm uses Fibonacci heap as a priority queue (Fredman & Tarjan, 1987). The space complexity of the Johnson's algorithm is $O(|V|^2)$.

The A^* algorithm which uses a special cost function to estimate whether vertices are near the destination vertex is similar to the Dijkstra's algorithm

(Hart, Nilsson, & Raphael, 1968). When the algorithm is applied to geographical graphs, the cost function of vertex u can be the distance between vertex u and the destination vertex. Thus, the closer vertex u to the destination vertex, the higher its priority. Therefore, the A^* algorithm might never probe vertices that are located too far from the target vertex. The search space of the A^* algorithm is bounded by $O(b^d)$, where b is an average number of successors per vertex and d is the length of the shortest path. Thus, the time complexity and the space complexity of the algorithm are $O(db^d)$ if the Dijkstra's algorithm is utilized. The A^* algorithm is able to solve the single-pair shortest path problem.

Structure driven techniques utilize the structural properties of graphs to skip non-relevant vertices and edges. For instance, using structural highways in graphs might shrink the search space. The contraction hierarchies (CH) algorithm adds highway edges to a graph (Geisberger, Sanders, Schultes, & Delling, 2008). Let (u, v) and (v, w) be two edges that connect vertices u, v and v, w, correspondingly. In these terms, a *highway edge* is an edge which is added to the graph and connects vertices u and w with the weight which equals to the sum of weights of edges (u, v) and (v, w). The first step of the algorithm is to preprocess the data and to build a set of contracted graphs. After this procedure, the diameter of the unweighted copy of the contracted graph has become much smaller. Thus, this procedure increases the speed of the Dijkstra's algorithm, which is executed on the modified graph, by decreasing the number of edges in the path. The hierarchy of the built contracted graphs is utilized to restore the actual path from the path obtained by the Dijkstra's algorithm in the modified graph. This algorithm is widely used in solving the shortest path problem for road networks. Milosavljević (2012) proved that contracted graphs produced by CH contain $O(h|V|logD)$ edges, where D is the diameter of the graph, V is a set of vertices and h is *highway dimension*, the maximum number of nodes required to hit all shortest paths contained in some region whose length is not too small compared to the size of the region. Thus, the time complexity of CH is $O(h|V|logD + |V|\log|V|)$ (Milosavljević, 2012).

Another approach which was suggested by Fu et al. (2013) is based on the scale-free property of social graphs. This approach cannot find an actual path between a pair of vertices, it can only estimate distance between them. According to the scale-free property, a social graph consists of a small set of popular (high-degree) vertices, modeling hubs, and a large set of non-popular vertices, modeling ordinary users. Thus, it is assumed that the shortest path between two random vertices includes some hubs. From this assumption, Fu et al. (2013) suggest to extract the core-net which is a subgraph consisting of popular vertices and of some other vertices which are added to the graph to make it to form only one connected component. Thereafter, a distance matrix for the core-net is calculated. According to the property that two random vertices in a social network are connected by six hops in average, a distance is searched as follows. Firstly, friend and friend-of-friends lists of the two vertices are calculated, thereafter, they are checked for intersection. If the lists have common vertices, then distance is found. Otherwise, the lists and the core-net are checked for in-

tersection. If they intersect, the distance is calculated, according to the distance matrix. The time complexity of the algorithm is $O(|N_u^2| + |N_v^2| + |C|)$, where N_u^2 and N_v^2 are sets of friend-of-friends vertices and C is a core-net of the graph.

Researchers widely use landmark-based approaches to estimate distances in large graphs. These approaches select a subset of nodes, landmarks, and precompute the distances from each landmark to all other nodes in the graph. The algorithm finds shortest distance through the landmarks and returns the shortest one as the answer for a query. Kleinberg et al. (2004) show that landmarks can be picked randomly with good theoretical results. Potamias et al. (2009) build landmarks according to basic metrics with better result than in the previous work. All the above mentioned landmark-based approaches estimates the lengths of the shortest path in $O(|L|)$, where L is a set of landmarks. Finally, the Orion system, offered in Zhao et al. (2010), embeds a graph into a Euclidean space and distance between two vertices is estimated according to Euclidean distance between them. The time complexity time of Orion is $O(1)$, as calculation of the Euclidean distance between a pair of vertices is needed. The main disadvantage of the mentioned algorithms is that they are only able to estimate distance between vertices, not to calculate an actual path.

A *spanner* of G is a subgraph G' that contains all vertices of G and a small set of edges. There are several papers in which algorithms that produce spanners with $O(|V|^{3/2})$ edges are described. In the papers written by Dor et al. (2000) and Elkin et al. (2004) the produced spanners contain shortest paths between any pair of vertices that are at most two hops longer than the actual shortest paths in the original graph. Althöfer et al. (1993) suggest a spanner production algorithm where shortest paths are at most three hops longer than original path. The time complexity of the algorithms is $O(|V|^{3/2})$. The space complexity of the algorithms is $O(|V|^{3/2})$. The above-mentioned spanner based approaches solve the single-source shortest path problem.

Cao et al. (2011) suggest another approach of building spanners which is named *Atlas*. The algorithm has two stages: building a search index; and fast queries to the built search index. The search index is realized as a set of spanning trees. The algorithm tries to sparse a graph by removing unrelated edges using the small-world property of social graphs and local clustering of the neighborhood graph of a user. In other words, the shortest path between a pair of vertices has tendency to go through hubs. Thus, the Atlas algorithm prunes friend connections between non-popular vertices, but keeps edges to hubs. The two mentioned properties of social graphs lead to the situation in which a pair of vertices has a lot of shortest paths between them. Thus, it is assumed that some of the edges can be removed without significantly decreasing algorithm's accuracy. To find the shortest path between a pair of vertices, the *Atlas* algorithm searches for the shortest path between the vertices in each spanning tree. Thereafter the shortest path of the found paths is selected as the result. To sum up, searching the shortest path between two vertices in a graph is reduced to searching the shortest path between two vertices in a spanning tree. This problem is actually the least common ancestor (LCA) problem (Aho, Hopcroft, &

Ullman, 1976). Cao et al. concluded that 20 spanning trees are enough to approximate social graphs. To handle dynamic graphs, Cao et al. (2011) suggested the strategy of replacement of old trees with new trees. Replacement of one tree per day is enough for not decreasing of the accuracy of the algorithm. Cao et al. (2011) evaluated the algorithm on several social graphs, the largest of which includes 43 million vertices and 1 billion edges. It was shown that changes in social graphs (removing and adding edges, removing and adding vertices) do not impact much the spanning trees. Nevertheless, the paper did not suggest how to perform local modifications of spanning trees. The time complexity of the algorithm is $O(k|L|)$, the space complexity of the algorithm is $O(k|L|)$, where k is the number of trees utilized in the algorithm and L is the maximal depth among the spanning trees.

4.2 Application requirements and the fitness of algorithms

In this section the above-mentioned algorithms are analyzed. The analysis is done, according to the accuracy of algorithms, memory usage, how they fit social network characteristics and how they fit algorithm requirements which are mentioned in the following paragraphs. The analysis is summarized in tables in the end of this section.

Mostly, requirements emerged as business desires of the Odnoklassniki social networking site. An acceptable algorithm should solve the single-pair shortest path problem. The algorithms are analyzed according to the following functional requirements.

1. *Performance*. An algorithm should solve the shortest path problem between a pair of vertices in admissible time; it is assumed that query time should not exceed 50 ms per query, as the algorithm can be used as a real-time service and a solution should be found as fast as possible.
2. *Accuracy*. Algorithm's accuracy should be acceptable. The accuracy is computed as the number of correct results of the algorithm divided by the number of paths used in the evaluation of the algorithm, and it should be more than 90%. This accuracy also includes the requirement that if the algorithm makes a mistake, then the length of a result should not exceed the length of a correct path with more than one. The algorithm can be used in graph analysis; thus, this kind of mistakes leads to incorrect statistics.
3. *Handling of short paths.* The algorithm solving the problem should not make mistakes in the case of short paths (length less than 3), because if the algorithm is deployed as a standalone service, answer found by the algorithm can be easily checked by users. Hence, if a user realizes that the algorithm returns wrong answers, then it could lead to lowering the prestige of a social networking site.

The constraints of the implementation of the algorithm are:

1. The algorithm should be implemented in *Java*, since all APIs of Odnoklassniki has been implemented in *Java.*
2. From the previous constraint, usage of the *heap memory* should be restricted to 1 GB, since in case of wasteful usage of the heap memory, *Java* will have to collect garbage (not used objects) which leads to significant pauses if the algorithm is run several times (Gosling, Joy, Steele, Bracha, & Buckley, Chapter 12. Execution, 2015).
3. Heuristic algorithms usually have a pre-computation step that precalculates some data. It is supposed that the algorithm use memory mapping I/O (Pai, Druschel, & Zwaenepoel, 1999) to store all data in the primary memory. Memory mapping allows storing data in the off-heap memory. Thus, the amount of the pre-computed data, stored in the off-heap, the size of the heap and the application code should not exceed the amount of the primary memory of a machine. The algorithm is assumed to be run on a machine with 64 GB of the primary memory. Thus, the algorithm should not use more than 64 GB of the primary memory. If processes exceed the volume of the accessible memory, the operating system dumps the virtual memory of some processes to the swap (Bach, 1986) which significantly slows the processes.
4. The volume of data transmitted through a computer network should be limited. It will decrease load of servers and waiting time for data transfer between servers.

BFS cannot handle large graphs. For instance, if a graph contains 200 million vertices and BFS is used, BFS uses 1.5 GB (200 000 000 * 8 bytes) of the heap memory to store all vertices (each vertex is represented as an 8 bytes long integer) in the inner queue in the worst case. Other algorithms, the Dijkstra's, Ford-Bellman, Floyd-Warshall and Johnson algorithms, need more memory. In addition, the performance of the algorithms is not acceptable, BFS needs a minute in a standard desktop computer to calculate the shortest path between a pair of vertices in a graph that contains four million vertices and 50 million edges (Potamias, Bonchi, Castillo, & Gionis, 2009). The other algorithms asymptotically take more time to find the shortest path. Thus, obviously these algorithms do not fit the requirements.

Contraction hierarchies, which are synthesized to handle large road graphs, cannot be applied to social graphs. The goal of the algorithm is to decrease the diameter of a graph. Thus, according to the social network characteristics, this strategy is not efficient for social graphs because of small average distance between vertices. Moreover, the algorithm increases the degree of vertices, while the degrees of vertices of social graphs are large. This problem can be resolved by the A^* algorithm, but a social graph is nowhere close to a planar one. Hence, it is not clear how to devise a priority function. Researchers try to work out how to plane a social graph, but after that, there will be another problem with high

dynamics of a social graph. The planar graph has to be updated every time when a user changes their friend list.

Landmark-based algorithms (Kleinberg, Slivkins, & Wexler, 2004; Potamias, Bonchi, Castillo, & Gionis, 2009) can approach the actual shortest distance between a pair of vertices. Orion (Zhao, Sala, Wilson, Zheng, & Zhao, 2010) maps a social graph to a Euclidean space and calculates distance between a pair of vertices as Euclidean distance between the corresponding points in the Euclidean space. The main disadvantages of the mentioned algorithms are that they cannot handle dynamic graphs, and they can only estimate the length of a shortest path. The same can be said about the core-net based algorithm (Fu & Deng, 2013).

And finally, the spanner based algorithms are analyzed. Dor et al. (2000), Elkin et al. (2004) and Althöfer et al. (1993) tell how to build a spanner with $O(|V|^{3/2})$ edges in the graph and suggest finding the shortest path with BFS or the Dijkstra's algorithm. The algorithm is not applicable for social graphs because, for instance, the median degree of vertices in Facebook is 99 (Ugander, Karrer, Backstrom, & Marlow, 2011). Hence, the spanner will not be as sparse as expected. The *Atlas* algorithm, which offers to build a set of BFS trees, is based on solving the LCA problem in spanning trees. The researchers also suggest an algorithm which updates trees according to changes in a graph, but the disadvantage of the algorithm is that the accuracy of algorithm is not acceptable (25-30%). Nevertheless, this algorithm can build a path between vertices in a graph and it was chosen as the starting point for the new one. It is assumed that it will be mixed with other techniques of shortest path searching to improve its accuracy.

From the literature review, it should be mentioned that currently the researches are mostly focused on distance estimation between vertices, not on calculation of actual paths. It can be concluded that searching of the shortest path is not a high priority task in analysis of social graphs today, since distance estimation is enough to calculate such metrics as diameter of a graph or neighborhood function. Also, from the business needs, applications, like ranked social search, can be done only by distance estimation algorithms. Nevertheless, the algorithm review discloses that no efficient algorithm for solving the shortest path problem in social graphs exists.

The above mentioned algorithms are summarized in Table 1. The table contains information about type of a graph for which algorithms are applicable, the time and space complexity of the algorithms.

TABLE 1 Summary of the mentioned algorithms

Algorithm	**Graph**	**Type of problem**	**Time complexity**	**Space complexity**
Breadth-first search (BFS)	Unweighted graph	Single-source	$O(\|V\| + \|E\|)$	$O(\|V\|)$
Dijkstra's algorithm	Weighted graph with non-negative edges	Single-source	$O(\|V\|\log\|V\| + \|E\|)$	$O(\|V\|)$
Ford-Bellman algorithm	Weighted graph	Single-source	$O(\|V\|\|E\|)$	$O(\|V\|)$
Floyd-Warshall algorithm	Weighted graph	All-pairs	$O(\|V\|^3)$	$O(\|V\|^2)$
Johnson's algorithm	Weighted graph	All-pairs	$O(\|V\|^2\log\|V\| + \|V\|\|E\|)$	$O(\|V\|^2)$
A* algorithm	Road (weighted) graph	Single-pair	$O(db^d)$	$O(db^d)$
Contraction hierarchies	Road (weighted) graph	Single-pair	$O(\|V\|(hlogD + \log\|V\|))$	$O(\|V\|)$
Core-net based algorithm	Social (unweighted) graph	Single-pair	$O(\|N_u^2\| + \|N_v^2\| + \|C\|)$	$O(\|C\|)$
Landmark-based algorithms	Social (unweighted) graph	Single-pair	$O(\|L\|)$	$O(\|L\|)$
Spanner-based approaches	Unweighted graph	Single-source	$O(\|V\|^{3/2})$	$O(\|V\|^{3/2})$
Atlas	Social (unweighted) graph	Single-pair	$O(k\|L\|)$	$O(k\|L\|)$

5 THE FIRST VERSION OF ATLAS+

The current section contains description of the proposed algorithm. First, the *Atlas* algorithm is described in detail. After that, improvement for the *Atlas* algorithm is suggested. Thereafter, the accuracy and the performance of *Atlas*+ are analyzed. The section also contains analysis of time complexity.

5.1 The Atlas algorithm

The current section describes the *Atlas* algorithm in detail. The *Atlas* algorithm is comprised of two phases: building of a search index and queries to it. The search index consists of a set of spanning trees which are stored on the hard drive. The tree construction is shown in Listing 1. The algorithm takes the number of spanning trees to be built as a parameter and returns the specified number of built trees. The strategies of the selection of starting vertices and adding new edges to the building tree are described below.

LISTING 1 Building of the searching index

```
1 Tree[] builtTrees(int numberOfTrees)
2     long[] startingVertices = selectStartingVertices(numberOfTrees);
3     Tree[] trees = new Tree[numberOfTrees];
4     for (int i = 0; i < numberOfTrees; ++i)
5         trees[i] = buildTree(startingVertices[i]);
6     return trees;
```

Method *buildTree* returns the built spanning tree which can be represented as an array of entries in which each entry points to its parent in the tree. To build a spanning tree, the strategy of selection of the starting vertex and the strategy of selection of edges should be chosen. Cao et al. (2011) evaluated the following strategies of selection of the starting vertices:

- The top *k*-centrality strategy in which *k* most popular vertices are chosen as the starting vertices;

- The scattered top *k*-centrality strategy in which *k* most popular vertices are chosen in such a way that the distance between a pair of the chosen vertices is at least two edges;
- The random selection strategy in which the starting vertices are chosen randomly.

In Cao et al. (2011) the top *k*-centrality strategy had the best characteristics.

At each step of the *Atlas* algorithm an edge is probed whether it can be added to the building spanning tree. In the paper three strategies of edge selection were evaluated:

- Breadth-first search with random tie-break in which a random edge of the addable edges is added;
- Breadth-first search with complementary tie-break in which the least used edge of the addable edges is added;
- The least covered edge first strategy in which the least used edge in the previous trees is added to the building tree.

The best accuracy is demonstrated by the breadth-first search with complementary tie-break.

Handling of dynamic graphs is done as follows. Several old trees are replaced with new trees. Also, it was shown that changes in social graphs do not impact much the built spanning trees.

To find the shortest path between vertices *s* and *t*, the *Atlas* algorithm finds the shortest path in each spanning tree and selects the shortest path from the found paths. The algorithm is shown in Listing 2.

LISTING 2 The *Atlas* algorithm

```
1 long[] atlas(long s, long t)
2     long[] result = null;
3     for (int i = 0; i < numberOfTrees; ++i)
4         long[] path = trees[i].path(s, t);
5         if (result == null || result.length > path.length)
6             result = path;
7     return result;
```

5.2 Improvement of the Atlas algorithm

The following section describes changes in the *Atlas* algorithm that improve its accuracy. The improvement is based on the properties of social graphs, in this case, the large value of the coefficient of local clustering. Properties of the new algorithm are analyzed, and according to them, two version of the new algorithm are suggested.

The local clustering coefficient describes the neighborhood graph of a vertex, the probability that a pair of adjacent vertices of a vertex is connected by an edge. The local clustering coefficient is large for social graphs, for example Facebook showed the value 0.15 (Ugander, Karrer, Backstrom, & Marlow, 2011) , and a subgraph of LiveJournal 0.13 (Stanford Network Analysis Project, 2015). It means that the probability that adjacent vertices of a vertex are connected by an edge is 15% for Facebook in 2011 and 13% for the subgraph of LiveJournal. Thus, a path between a pair of vertices can be shortened. In Fig. 6 a path between vertices *u* and *v* is shown. The dashed edge connects the adjacent vertices of vertex *w*. Thus, the path between vertices *u* and *v* can be shortened through the dashed edge.

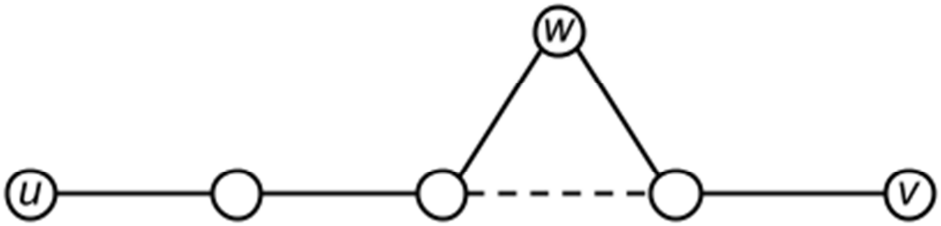

FIGURE 6 Shortening of the path

Hence, the result of the *Atlas* algorithm can be improved with help of some adjacent vertices of the vertices obtained by the *Atlas* algorithm. The proposed algorithm looks as in Listing 3.

LISTING 3 The proposed algorithm (version 1)

```
1 long[] path(long s, long t)
2     long[][] paths = atlas(s, t);
3     long[][] adjacencyLists = getAdjacencyLists(paths);
4     Map<Long, List<Long>> graph = buildGraph(adjacencyLists);
5     return bfs(graph);
```

The new algorithm, first, searches for the shortest paths in the spanning trees (the *atlas* method, line 2). Thereafter, the adjacent vertices of the vertices obtained by *Atlas* are requested (the *getAdjacencyLists* method, line 3). Based on that, a graph is built (the *buildGraph* method, line 4) in which BFS finds the shortest path between the source and the destination vertices (the *bfs* method, line 5). The result of *Atlas+* is the path found by BFS. The building graph is stored in a hash table in which keys are ids of vertices and values are lists of adjacent vertices.

5.3 Time complexity of the algorithm

To measure the time complexity of the shortest path searching algorithm, analysis of the each step is needed. Finding of the shortest path in a tree takes time

linear with regards to the depth L of the tree $O(L)$. At the same time the depth of the trees are bounded by the diameter of social graphs. These values cannot be large, because of the characteristics of social networks they are modelling. E.g., the diameter of social graph of LiveJournal is 16, and 9 for social graph of Orkut (Stanford Network Analysis Project, 2015). Search of k shortest paths in k trees takes time $O(kL)$.

The number of the queried incident edges are bounded by dkL, where d is the maximal degree of any vertice in the original social graph. Also the number of edges queried by the proposed algorithm is bounded by dkL. Thus, the breadth-first search algorithm works in $O(dkL)$ in the worst case. The second version of the searching algorithm works for the same time as the first version, but some steps are done more efficiently and constants in the appropriate time functions are smaller.

Thus, the summarized time complexity of the proposed shortest path searching algorithm depends on the depth of trees, number of trees and the maximal degree of vertices in the social graph and equals to $O(dkL)$. Additionally, some social networking sites limit the maximal number of friends. As for Odnoklassniki, the value is 5000 (API OK, 2015). Therefore, d is assumed to be a constant.

5.4 Algorithm implementation

The following section describes implementation details of the proposed algorithm. First, the API of the Odnoklassniki social networking site with which the algorithm will be shipped is described. A flexible mechanism based on the object-oriented programming is utilized (Cox, 1986). Thereafter, memory mapping I/O (Pai, Druschel, & Zwaenepoel, 1999) which is used for handling large files is discussed. Finally, to make the algorithm multi-threaded, a lock-free open-addressing hash table is suggested.

5.4.1 Description of social graph API

The following section describes accessible API of the graph of the Odnoklassniki social networking site. Each vertex of the Odnoklassniki social graph has id which is an 8 bytes long integer. The vertices of the graph are placed on four servers (shards). To retrieve some information about a vertex, for example, to retrieve the adjacency list of a vertex, the shard in which the vertex is located should be determined. The shard in which a vertex is located is calculated according to the id of the vertex. E.g. currently the algorithm of shard determination takes remnant by dividing id of vertex by 256. After that, each interval of the remainder relates to a shard. In this case, the remainder between 0 and 63 relates to the first shard, the remainder between 64 and 127 – to the second shard and etc. Thus, this operation needs little resources and time. After the

determination of the shard, the adjacency list of the desired vertex is requested from the shard. The shard returns the adjacency list of the vertices that is an array of 8 byte integers. The size of the requests and responses is bounded by 16 MB. Thus, requests should be split in order not to overflow the limit of the size of requests and responds.

The algorithm was evaluated on the Odnoklassniki social graph, a sampled snapshots of LiveJournal and Orkut obtained from SNAP (Stanford Network Analysis Project, 2015) and a locally sampled part of the Odnoklassniki social graph. The algorithm was implemented in *Java*, and was run on an Odnoklassniki's server. To avoid code duplication which can occur when there are several similar entities with different behavior, the object-oriented programming is employed in the work. Thus, the classes which implement social graphs have the same interface, Graph. The Graph interface provides API of retrieving shards of the graph. The Graph interface represents a social graph and has the following public API:

1. Shard[] *getShards*(), a method to retrieve shards of the social graph;
2. int *maxQuerySize*(), a method to get the limit of the size of requests and responses;
3. Shard *getShard*(long id), a method to retrieve a shard by the id of vertex.

The Shard interface represents a server shard with the following API:

1. long[][] *getFriends*(long[] ids), a method to retrieve adjacency lists by a list of vertices;
2. int[] *getDegrees*(long[]), a method to retrieve the list of degrees of vertices.

To implement the proposed API for social graph of the Odnoklassniki social networking site, a software design pattern called *delegation* is utilized (Wolfgang, 1994). The implemented class, *OdnoklassnikiGraph*, relies on another class which represents the social graph of Odnoklassniki and is fed to *OdnoklassnikiGraph* as a parameter of the *OdnoklassnikiGraph*'s constructor and calls appropriate API of the social graph. Concerning the implementation of classes which represent locally sampled social graphs (*LiveJournalGraph*, *OrkutGraph*), locally sampled social graphs are read from a file stored locally. File of a locally sampled graph is comprised of a header and a table parts. The header part contains $N+1$ records which are comprised of the id (8 bytes) of a vertex and the offset (8 bytes) of the adjacency list in the table part, where N is a number of vertices in the graph. The table part of the file contains adjacency lists stored one after another. Retrieving of the adjacency list of vertex v consists of the following steps:

1. read the offset of the adjacency list of vertex v;
2. read the offset of the adjacency list of the vertex which follows vertex v;
3. read values between the obtained values.

To handle the last vertex in the header part of the file, a dummy vertex, with id -1, is written to the header. The dummy vertex has no adjacent vertices. Thus, the dummy vertex indicates the end of the header part of file.

The desirable Graph class is specified through configuration files. Thus, the suggested class hierarchy allow flexible algorithm evaluation without changes in the source code.

5.4.2 The least common ancestor problem

Searching of the shortest path between a pair of vertices in a tree reduces to searching of the least common ancestor problem. The LCA problem may be solved by Bender-Farach-Colton algorithm, the Tarjan's algorithm (Bender & Farach-Colton, 2000; Tarjan, 1976) and many others. Each of the algorithms contains the pre-calculation stage. Hence, the algorithms cannot be applied to the *Atlas* algorithm, because changes in the original social graph triggers changes in the pre-calculated data for the LCA search which cannot be done fast. Additionally the pre-calculated data for the LCA search requires memory space which is not less than the space needed to store the trees. Thus, fast algorithms for the LCA search are not acceptable because of the large memory usage.

Trees built by the *Atlas* algorithm are not deep, since diameter of social graphs is not so large. Thus, the least common ancestor can be found in linear time with regards to the depth, since it does not take much time to find a path in a tree with small depth. Let u and v be a pair of vertices for which solving of the LCA problem in a tree is needed. Firstly, the algorithm searches for paths to the root of the tree from vertices u and v. Thereafter, the obtained paths are compared beginning from the root. The algorithm stops as soon as the vertex from the first path does not equal to the vertex from the second path. Let u_1 and v_1 be the first pair of vertex on which difference occurs and w be the parent of u_1 and v_1, then the result path will be $u \dots u_1 w v_1 \dots v$. If the root vertices are not the same, then u and v do not have common ancestor. It means that vertices u and v belong to different trees. It can happen if the original graph contains several connected components.

Hence, to solve the LCA problem, the parent vertex of each vertex needs to be stored. Thus, trees can be stored as an array of integers. Let p be an array of integers in which a tree is stored and i be the id of a vertex. Thus, $p[i]$ stores the id of parent of vertex i. All vertices of the initial social graphs are enumerated from 1 to N, where N is the number of vertices in the graph. This scheme is used because the ids obtained from the API can have gaps and can be randomized. Generated trees are too large to be stored in the heap, circa 14-16 GBs in total for the graph of the Odnoklassniki social networking site. Additionally, mapping from social graph ids to tree ids should be stored in the primary memory. To overcome the problem memory mapped files, which are mapped to the virtual memory, are utilized. The benefits of the suggested solution are (Bach, 1986):

1. demand paging, i.e. files are loaded into physical memory by pages, and only when that page is referenced; therefore, a part of the file is loaded when it is necessary, and a part of the file can be replaced by another one if physical memory is full;
2. page cache, i.e. several processes can share memory mapped files between each other; thus, the algorithm can be performed in several processes simultaneously sharing a large volume of memory.

5.4.3 Open-addressing hash table

A hash table is a data structure which allows insertion, removing and examination of elements in $O(1)$ time (Cormen, Leiserson, Rivest, & Stein, 2001). A hash table is implemented as an array of elements; the length of the array is m. Let us suppose that each element is drawn from the universe U. A hash function h is a function $h: U \to \{0, \dots, m-1\}$. Thus, each element of the universe has a hash value $h(e)$. The index in which an element e is stored in the hash table equals to $h(e)$. Hence, to insert an element into a hash table, the hash value of the element is calculated, and the element is put to the calculated cell of the array. Removing and examination of elements are performed in a similar way.

Two keys may have the same hash value that is called a *collision*. Therefore, to store two keys with the same hash values, collision resolution strategies are employed:

1. collision resolution by *chaining*, in which the elements which hash to the same index are put in a linked list;
2. collision resolution by *open-addressing*, in which the elements are stored in the hash table itself.

In case of the open-addressing strategy of collision resolution, to insert a key, the hash table is examined (probed) until an empty slot in which the key is put is found. For the open-addressing collision resolution, the slots which are probed should be determined. Thus, the hash function becomes as $h: U \times \{0, \dots m-1\} \to \{0, \dots, m-1\}$.

It means that the hash function represents the sequence of positions to be probed. The algorithm of searching or removing element k probes the same sequence of positions as the insertion algorithm.

To compute the probe sequences, three techniques are used for the open-addressing strategy:

- linear probing with the hash function $h(k,i) = (h(k) + i) \; mod \; m$;
- quadratic probing with the hash function $h(k,i) = (h(k) + c_1 i + c_2 i^2) \; mod \; m$;
- double hashing with the hash function $h(k,i) = (h_1(k) + i h_2(k)) \; mod \; m$.

The first strategy suffers from a problem called *primary clustering* which means that the average searching time is increasing with increasing number of occupied slots. For the quadratic probing if two keys have the same initial probe position, then their probe sequences are the same. This property is called *secondary clustering*. Double hashing is proven to be the best methods available for open addressing because the produced probe sequences have many random permutations. Double hashing utilizes $O(m^2)$ probe sequences, rather than *O(m)* in linear and quadratic probing. As a result the performance of double hashing appears to be the best of the three strategies.

Standard *Java* collections may only store objects, which means that primitive types, like long, integer, have to be boxed to class wrappers, e.g the Long class is for long integer. Using standard *Java* collections for primitive types leads to the following problems with performance and memory usage:

1. more heap memory than necessary is used, since the corresponding *Java* object contains headers and other meta information in addition to primitive types;
2. objects need to be garbage collected, while memory for primitive types can be allocated directly from the stack memory;
3. indirect access to primitive types which leads to slowing down program execution;
4. problems with caching: an array is supposed to be stored contiguously; thus, arrays are easy to be cached in order to decrease access time to elements of the array, but in terms of boxed integers, the array is as an array of pointers to objects randomly spread around the heap.

To eliminate the mentioned problems, implementation of the hash table provided by Trove is utilized (Trove, 2015) . Trove is a *Java* library with implementation of *Java* collections for primitive types. In the Trove library hash tables are implemented as open-addressing hash tables with double hashing. Table 2 shows that the performance of Trove's hash table does not fit the requirements of the proposed algorithm. From the analysis, the most part of time is spent while adding elements to the hash table. Thus, to speed up the algorithm, an open-addressing hash table was implemented. The implementation of the hash table utilizes the property that *Atlas*+ only adds new elements to the hash table and makes queries to it. Because of that, the rehashing algorithm in the hash table can be optimized. Let *k* be a maximal number of probes done during insertions to the open-addressed hash table. If elements are not removed, then the searching element *e* cannot lie further than *k* iterations from the *h(e)* cell. Thus, the searching algorithm does not need to make more than *k* rehashings. The implementation of the hash table is shown in Listings 4-8. The following symbols are used in the implementation.

- *hash*(*k*) is a method which represents a hash function.
- *getIndex(h)* is a method which normalizes the calculated hash code.

- *getNextIndex(step, index)* is a method which counts the next index in case of collision.
- *keys* is an array of keys.
- *values* is an array of values.
- *NO_KEY* represents a value which corresponds to empty cell.
- *maxChainLength* is a maximal number of iterations is done during insertion.
- *put(k, v)* is a method which associates the value *v* with the key *k* and returns the previous value associated with the key *k* or null if it does not exist.
- *putIfAbsent(k, v)* is a method which associates the value *v* with the key *k* and returns the value *v* if *k* was not in the hash table, or the existing value *v'* associated with the key *k*.
- *get(k)* is a method which returns the value associated with the key *k* or null otherwise.

LISTING 4 Key insertion

```
 1 int insertKey(int key)
 2     int step = 1;
 3     int index = getIndex(hash(key));
 4
 5     do
 6         if (keys[index] == NO_KEY)
 7             keys[index] = key;
 8             maxChainLength = max(maxChainLength, step);
 9             return index;
10         else if (keys[index] == key)
11             return index;
12
13         index = getNextIndex(step++, index);
14     while (true);
```

LISTING 5 The *put* method

```
1 V put(long key, V value)
2     int index = insertKey(key);
3     V v = values[index];
4     values[index] = value;
5     return v;
```

LISTING 6 The *putIfAbsent* method

```
1 V putIfAbsent(long key, V value)
2     int index = insertKey(key);
3     V oldValue = values[index];
4     if (oldValue != null)
5         return oldValue;
6     values[index] = value;
7     return value;
```

The *put* and the *putIfAbsent* methods look up for a free position or check whether the key has been added before. The *get* method looks through the hash table. Iterations stop as soon as a free position is encountered, the key is encountered in the hash table or the number of done steps is more than *maxChainLength*. Thereafter, the method retrieves the value by index.

The *get* method looks through the hash table. Iterations stop as soon as a free position is encountered, the key has been put to the hash table or the num-

ber of done steps is more than *maxChainLength*. Thereafter, the method retrieves the value by index.

LISTING 7 Retrieving of the index of a key

```
 1 int indexOf(long key)
 2     int step = 1;
 3     int index = getIndex(hash(key));
 4     do
 5         long cur = keys[index];
 6         if (cur == key)
 7             return index;
 8         else if (cur == NO_KEY) {
 9             return -1;
10
11         index = getNextIndex(step++, index);
12     while (step <= maxChainLength);
13     return -1;
```

LISTING 8 The *get* method

```
 1 V get(long key)
 2     int index = indexOf(key);
 3     if (index == -1) {
 4         return null;
 6     return values[index];
```

In Table 2 the performance of the hash tables, the Trove's implementation and the proposed hash table with different sequence computing strategies are analyzed. To analyze performance, 50000 paths of the LiveJournal social graph were calculated. After that, *Atlas+* was run on the calculated paths. The maximal number and the average number of rehashings done during insertion are analyzed. Both values are averaged for all runs of the searching algorithm. From the table, the quadratic hashing shows the best performance.

TABLE 2 Analysis of the proposed hash table

Hash table	Average number of rehashings	Maximal number of rehashings
Trove	2	5.2
Proposed hash table + linear hashing	2.1	5
Proposed hash table + quadratic hashing	1.1	5.5
Proposed hash table + double hashing	1.8	4

5.5 Evaluation of the first version of the algorithm

The following section describes how the proposed algorithm is evaluated. For evaluation of the proposed algorithm Odnoklassniki, LiveJournal and Orkut,

obtained from SNAP (Stanford Network Analysis Project, 2015), were utilized. Table 3 shows size of the social graphs used in evaluation.

TABLE 3 Social graphs used in evaluation

Graph	Vertices	Edges
Odnoklassniki	205 000 000	25 000 000 000
LiveJournal	3 997 962	34 681 189
Orkut	3 072 441	117 185 083

5.5.1 Evaluation of tree building strategies

The following section analyzes combinations of strategies suggested in the paper (Cao, Zhao, Zheng, & Zhao, 2013). These strategies involve strategies of selection starting vertices and strategies of adding new edges to trees.

For analysis of strategies, a sampled subset of the LiveJournal social graph is used. To analyze the strategies shortest paths between 50000 random pairs of vertices of the graph were calculated by the breadth-first search. After that, the proposed algorithm calculated paths between the pairs of vertices using sets of trees built by different strategies. The average length of the paths found by BFS was compared with the average length of the paths found by the proposed algorithm. The average length of the paths obtained by BFS is 5. Thus, according to Table 4, the best results was shown by the top *k*-centrality strategy as the strategy of selection starting vertices and by BFS with random tie-break as the strategy of adding new edges to trees. BFS with random tie-break also allows building trees distributed and separately. In comparison with other strategies of adding new edges, this strategy needs much less memory than the two other strategies, because they need temporary memory for counters, e.g. counters for the calculation how many times each edge are used in the trees. Thus, BFS with random tie-break was selected. The results are presented in Table 4. The columns of the table relate to the strategies of selection of starting vertices, the rows of the table relate to the strategies of adding new edges to the trees.

TABLE 4 Average length of paths

	BFS with random tie-break	BFS with comp. tie-break	Least Covered Edge First
Top k-centrality	5.12	5.1	5.1
Scattered top k-centrality	5.15	5.12	5.1
Random	5.4	5.36	5.36

Overall, the top *k*-centrality strategy was chosen for the selection of starting nodes, BFS with random tie-break was chosen for adding new edges to tree. The suggested solution allows concurrent and independent tree construction. Concerning the building time of the trees for the Odnoklassniki social graph, which contains 205 million nodes and circa 25 billion ties, 80 minutes are needed to build a tree on a machine with *Intel Core i7-4702MQ* CPU and 64 GB of the primary memory. A tree is built in the off-heap memory, and when the construction of a tree finishes, the tree is dumped to the hard drive. Thus, building of 50 spanning trees takes 32 hours if tree construction is done concurrently in two threads. The tree construction time is acceptable.

5.5.2 Evaluation of accuracy

To analyze the accuracy of the proposed algorithm, random pairs of vertices from the Odnoklassniki social graph and the samples of LiveJournal and Orkut were randomly selected. Table 5 shows the number of paths grouped by the length of the path. According to the properties of social graphs, the shortest paths with length more than five edges are very rare. Thus, the selected sets of paths are representative for algorithm evaluation.

The suggested algorithm counted a path between each pair of vertices; after that, the result of the algorithm was compared with the actual shortest path. In addition, the accuracy of the algorithm grouped by the length of the paths was calculated. Fig. 7-Fig. 10 show the accuracy of the algorithm depending on the number of trees used in search. Hence, 25-30 spanning trees are enough to obtain the desirable accuracy, more than 90%, and desirable time, shown in Table 6.

TABLE 5 Number of paths grouped by length

Length	3	4	5	6	Total
Odnoklassniki	7439 (5%)	61004 (41%)	71419 (48%)	8927 (6%)	148789
Sample of LiveJournal	5151 (10%)	18484 (37%)	25061 (50%)	1304 (3%)	50000
Sample of Orkut	3121 (6%)	20531 (41%)	23482 (47%)	2866 (6%)	50000

Additionally, according to Fig. 8-Fig. 10, the accuracy of the algorithm for long paths (four-five edges) is better than for shorter paths (two-three edges), but the difference is insignificant. It should be mentioned that the algorithm is always correct for the paths whose length is less than three edges, because of the query of adjacent vertices. If the algorithm makes a mistake, the difference in paths is not more than one edge. Overall, the proposed algorithm has the required accu-

racy. The graphics for LiveJournal and Orkut was built for 30 trees, because the results do not change considerably after 30 trees.

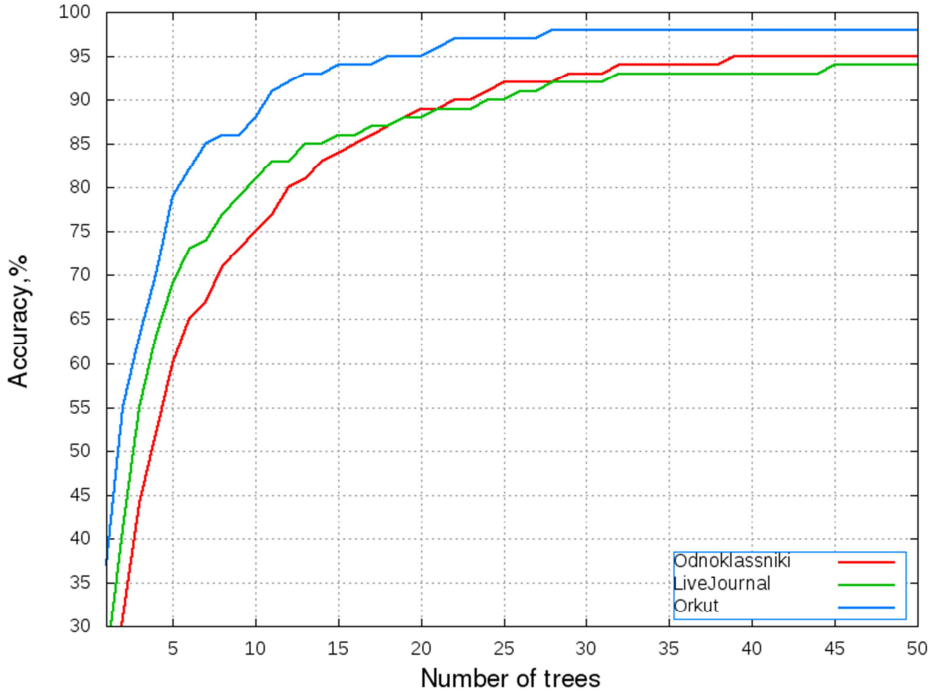

FIGURE 7 Accuracy of *Atlas*+ depending on the number of trees with regards to the number of used spanning trees

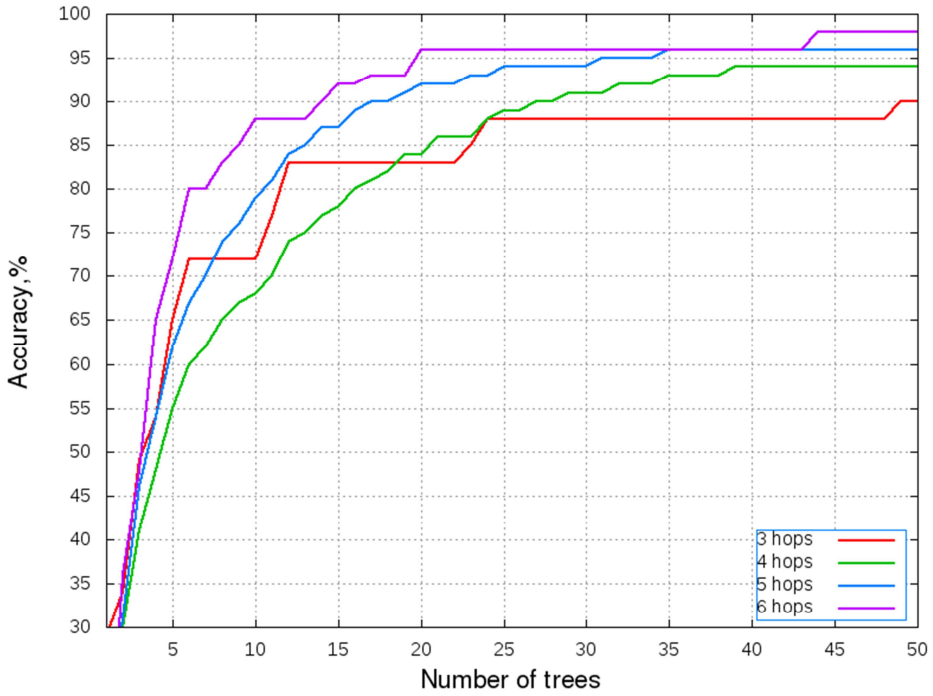

FIGURE 8 Accuracy of *Atlas*+ grouped by length (Odnoklassniki)

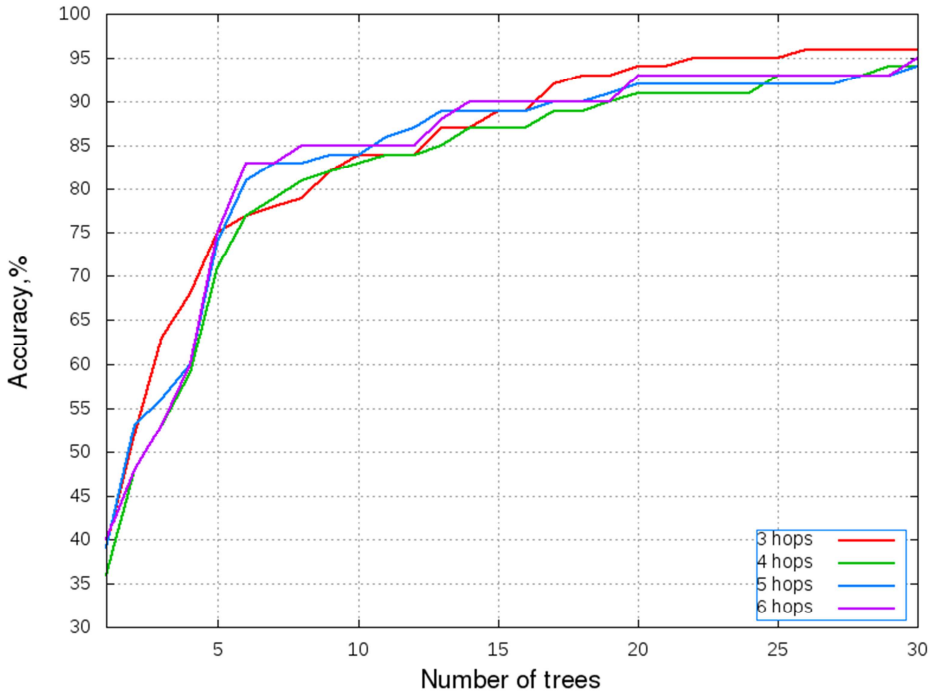

FIGURE 9 Accuracy of *Atlas*+ grouped by length (LiveJournal)

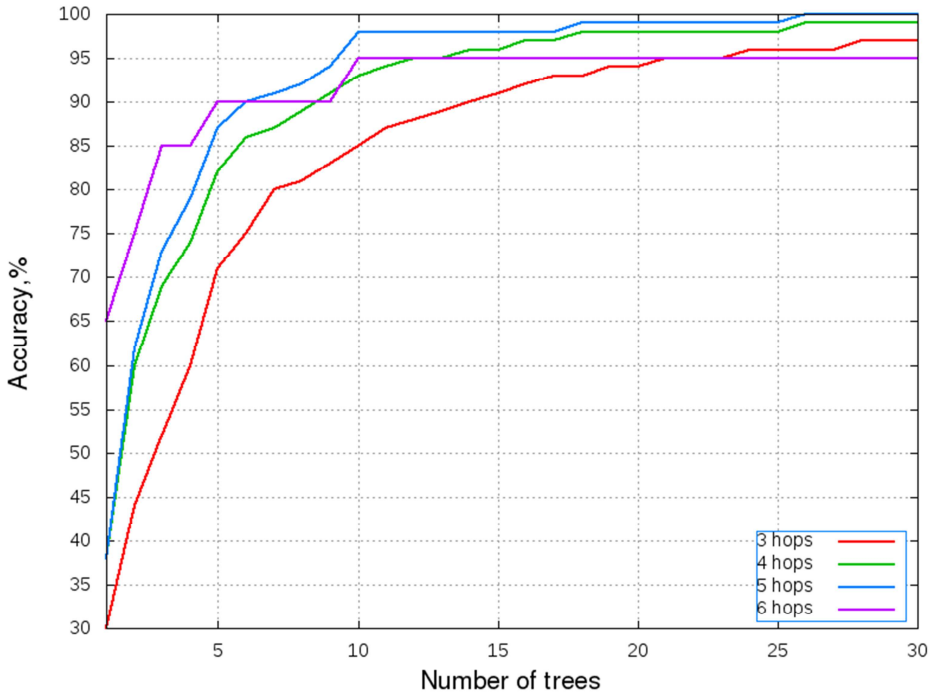

FIGURE 10 Accuracy of *Atlas*+ grouped by length (Orkut)

5.5.3 Evaluation of performance

This section is devoted to the performance of the algorithm depending on parameters and modifications of the algorithm.
Table 6 shows the performance of the algorithm on different social graphs (Odnoklassniki and subgraphs of LiveJournal and Orkut). Table 7 shows the time required to build spanning tree for the selected social media site data, as well as average query time for shortest path query between two random vertices. As it shows, performance of the algorithm is acceptable on all graphs. Nevertheless, since LiveJournal and Orkut were locally sampled, network time is not taken into account in Table 6. Also, tree construction can be done concurrently, thus, the time can be improved. Concerning Odnoklassniki, trees were built in two threads, each tree was built in its thread; hence, the construction time of a tree is forty minutes.

TABLE 6 Performance on different social graphs

Social graph	**Size of a tree**	**Vertices**	**Construction time**	**Query time**
Odnoklassniki	572 MBs	150M	80 minutes	51 ms
LiveJournal	15 MBs	3 997 962	20 seconds	17 ms
Orkut	11 MBs	3 072 441	83 seconds	21 ms

The number of vertices in the spanning trees for Odnoklassniki differs from the number of vertices in the original graph, because removed or blocked users were not used in the tree construction.

Table 7 contains the performance measurement of the first version of the algorithm, the version with the proposed hash table. Column *Atlas algorithm* relates to the original *Atlas* algorithm, column *Network query* relates to the *getAdjacencyLists* method of the proposed algorithm. In the performance measurements 25 spanning trees were used, since 25 trees are enough to reach the desirable accuracy. Also, the performance of each step of the algorithm was measured. The measurement was performed on the machine with *Intel Core i7-4702MQ* CPU, 64 GBs of the primary memory and Linux (Ubuntu 14.04).

TABLE 7 Performance of the first version *Atlas+*

Algorithm	**Atlas algorithm**	**Network query**	**Building of a hash table**	**BFS**	**Total**
First version	< 1 ms	32 ms	80 ms	40 ms	152 ms
First version + hash table	< 1 ms	32 ms	61 ms	33 ms	127 ms

According to the table, the first version of *Atlas+* spends the most part of time for the building a graph stored in a hash table.

6 THE SECOND VERSION OF ATLAS+

The first version of *Atlas*+ does not have acceptable performance. The algorithm spends too much time for adding elements to a hash table, and, second, too large volume of memory are transmitted through a computer network. The following section is devoted to the improvement of the performance of the *Atlas*+ algorithm. The accuracy of the second version of the algorithm has not been analyzed, since the improvement suggested in the following section does not impact its accuracy.

6.1 Improvement of Atlas+

Let us call the vertices retrieved at second step of the algorithm *new vertices*. To analyze the algorithm, the paths counted by the *Atlas* algorithm and the paths obtained by the proposed algorithm were compared. From the comparison of the paths, it was concluded that the improved path may be comprised of pieces of the paths obtained by the *Atlas* algorithm and not more than one vertex obtained after the query to the social graph. Hence, the algorithm needs to store only two edges on which the shortest distances to the source and the destination vertices are reached for each vertex. For the analysis, the Odnoklassniki social graph and the sampled subgraphs of LiveJournal and Orkut were utilized (Stanford Network Analysis Project, 2015). 148789 shortest paths were selected randomly from Odnoklassniki and 50000 paths from the samples of LiveJournal and Orkut. Thus, the second version looks as in Listing 9.

The new algorithm, first, searches for the shortest paths in the spanning trees (the *atlas* method, line 2). Thereafter, the adjacent vertices of the vertices obtained by *Atlas* are requested (the *getAdjancencyLists* method, line 3), as in the first version. After that, the proposed algorithm builds a graph comprised the vertices obtained by the *Atlas* algorithm and those requested edges which both vertices are some of the vertices obtained by the Atlas algorithm (the *buildGraph*

method, line 4). In 5-6 lines two trees of shortest paths from *s* and *t* are built by BFS. The *findMinimum* method finds a vertex on which minimal sum of distances from the vertex to *s* and *t* is reached. The *findMinimum* method stores all new vertices in a hash table in which keys are ids of the new vertices and values are objects of the *Vertex* type storing distances to the vertices *s* and *t*. After that, the shortest path is selected from the path counted by BFS (line 8) and the path counted on the 9 line. The *bfs* method returns a tree of shortest paths. A tree of shortest paths is comprised of a map in which keys are ids of vertices and values are ids of parent vertices; parents of root vertices are assumed to be -1. Thus, to find the shortest path between the vertex *s* and another vertex *u*, the algorithm iterates and queries parents of the current vertex starting from *u* until a root vertex (the *getPath* method, line 8). The *Vertex* type is a type comprised of id of the vertex and two other ids of adjacent vertices on which minimal distances to the vertices *s* and *t* are reached. The second *getPath* method (line 9) is presented in Listing 10 and works as follows. First, paths in both trees are found. If one of them does not exist, then the algorithm returns *null*, otherwise, the algorithm returns the shortest path which goes through the vertex *v.id*.

LISTING 9 The proposed algorithm (version 2)

```
 1 long[] path(long s, long t)
 2     long[][] paths = atlas(s, t);
 3     long[][] adjacencyLists = getAdjacencyLists(paths);
 4     Map<Long, List<Long>> graph = buildGraph(paths, adjacencyLists);
 5     Map<Long, Long> treeS = bfs(s, graph);
 6     Map<Long, Long> treeT = bfs(t, graph);
 7     Vertex minVertex = findMinimum(s, t, treeS, treeT);
 8     long[] bfsPath = getPath(treeS, t);
 9     long[] path = getPath(minVertex, treeS, treeT);
10     return shortestOf(bfsPath, path);
```

LISTING 10 Restoring a path from the BFS tree

```
1 long[] getPath(Vertex v, Map<Long, Long> treeS, Map<Long, Long> treeT)
2     long[] toS = getPath(treeS, v.idToS);
3     long[] toT = getPath(treeT, v.idToT);
4     if (toS == null || toT == null) return null;
5     return toS.concat(v.id).concat(toT.reverse());
```

Table 8 contains the number of vertices and edges utilized in the proposed algorithm and the number of vertices which degree equals to one among those vertices. According to Table 8, 339859 of the new vertices (67%) cannot be used in the improvement of paths, as their degree equals to one.

TABLE 8 Analysis of the first version of the algorithm

Vertices	Edges	Vertices having degree equal one
501324	10524245	339859

Thus, the suggested improvement decreased the number of stored edges to 2*N*, where *N* is the number of vertices in the built graph. For example, in this case the number of stored edges is decreased to one tenth of the earlier case. In the

first version of the algorithm the number of edges is dN, where d is the average degree of vertices in the built graph. Since paths have tendency to go through hubs, the value of d is large. Moreover, storing of edges without the suggested heuristics leads to usage of a large volume of the primary memory. Adjacency lists are stored in *ArrayList*, because of the sparsity of a social graph. This approach also allows random access queries to adjacency lists in time $O(1)$. Nevertheless, *ArrayList* pre-allocates an array. Hence, empty lists or not so large *ArrayList*'s, less than capacity of the array, needs much more memory than is really needed. For example, the initial capacity of an empty *ArrayList* implemented in *Java* 8 is 10. Thus, the suggested heuristics decreases the volume of memory which is needed for storing relevant edges.

The suggested algorithm is depicted in Fig. 11-Fig. 14. Let the proposed algorithm search for the shortest path between vertices v_1 and v_{11} in the graph shown in Fig. 11.

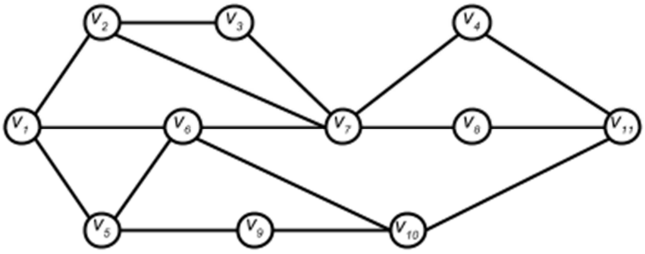

FIGURE 11 The original graph

First, the *Atlas* algorithm finds two paths between the vertices, path $v_1v_2v_3v_7v_4v_{11}$ is drawn by dashes and path $v_1v_5v_6v_7v_8v_{11}$ is drawn by dots. Fig. 12 shows the two paths found by the *Atlas* algorithm. Other vertices and edges of the original graph are marked by gray color.

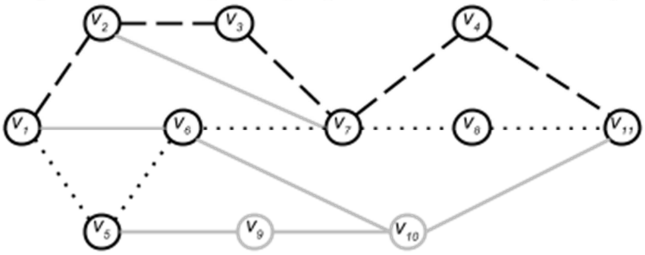

FIGURE 12 The two paths found by the Atlas algorithm

Fig. 13 depicts the graph that consists of the previously obtained vertices and of the additional edges queried from the original graph that connect the vertices.

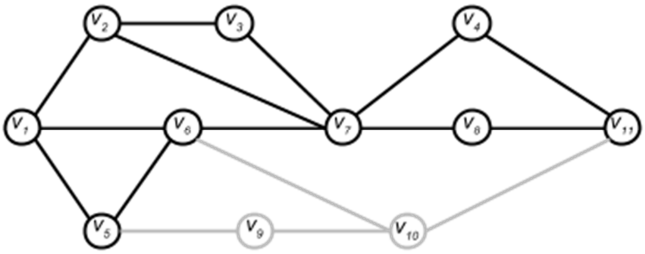

FIGURE 13 The graph with edges queried from the original graph

In Fig. 14 the algorithm looks for a new adjacent vertex that is not in the built graph, on which the shortest path between v_1 and v_{11} is reached. The shortest path, marked with gray vertices, between v_1 and v_{11} is $v_1 v_6 v_{10} v_{11}$.

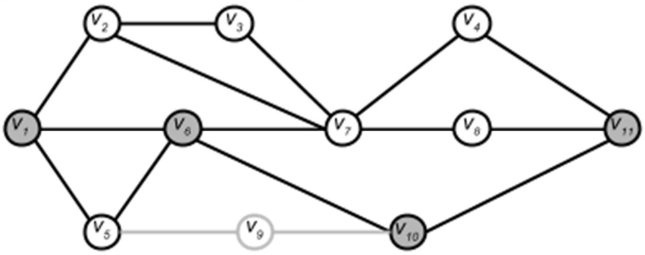

FIGURE 14 The found shortest path

Additionally, the algorithm may be accelerated if the scale-freeness of social graphs is employed. The scale-free property means that social graphs are built around highly-degreed vertices. Since the spanning trees used in the *Atlas+* algorithm are built around popular vertices too, the responses for the requests of adjacent vertices appear to be large (more than 2 MB). Thus, the number of requested vertices should be bounded. This heuristic leads to:

1. decreasing of size of data queried via network;
2. decreasing of amount of data which is needed to be processed by the algorithm.

Let a query "get at least k vertices or vertices with degree more than some bound d" be named as a query of the popular adjacent vertices. To find a reasonable value for the degree d, the following plot in Fig. 15 is utilized. The degrees of vertices queried in the original graph that shorten the shortest path obtained by the *Atlas* algorithm were assessed. If the *Atlas+* algorithm is able to find several shortest paths between a pair of vertices, the path in which the degree of such vertex is largest is selected. The plot in Fig. 15 shows the cumulative normalized number of vertices that shortens the paths with regards to their degree. According to the diagram, the shortest path is shortened through very popular vertices; only 2-3% of all paths are improved through vertices with degrees circa 100 - 200 which are also rather popular vertices. According to the analysis of degree distribution in the Odnoklassniki social graph, only 7% of

vertices of the social graph have degree more than 200. Thus, if adjacent vertices the degree of which is more than some fixed threshold are requested, the volume of sent and processed data decreases essentially. As a trade off the accuracy of the algorithm decreases by 1-2% which is still acceptable if the threshold is 200. Thus, by setting the threshold d at 200, only 7% of the vertices are returned to the query of the popular adjacent vertices above, by among them are all those that have up to 5000 adjacent vertices.

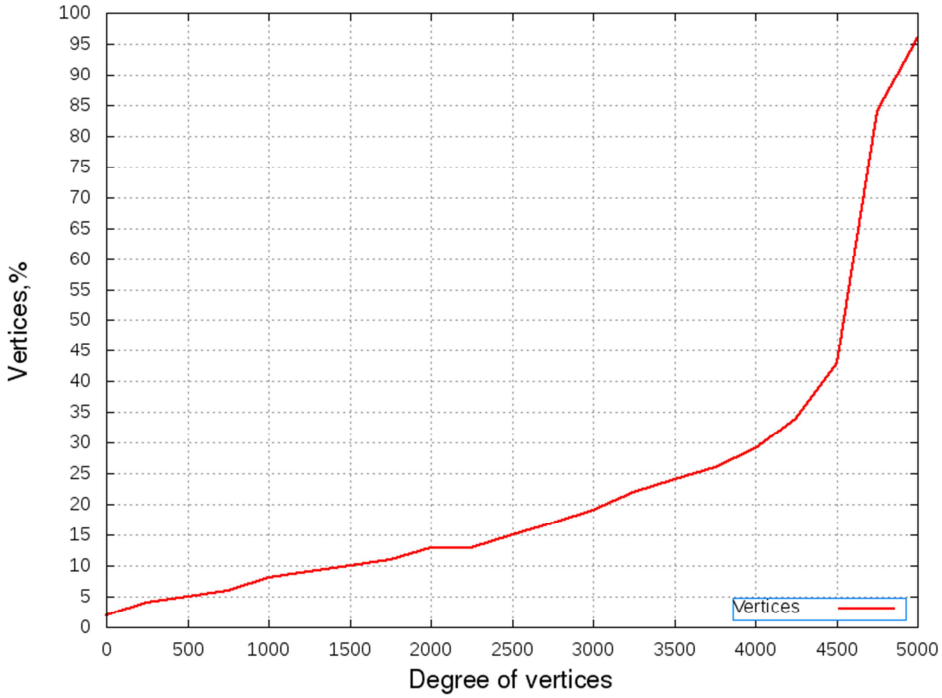

FIGURE 15 Cumulative share of new vertices that shorten the paths depending on the degree of the vertices

Also, the *findMinimim* method can be done in several threads. All vertices probed for shortening of paths are split into several groups, the number of groups equals to the number of the threads. All of the threads share the common hash table to find a vertex on which the shortest distance is reached. The hash table should be thread-safe, i.e. guarantee consistency of data stored in it. The implementation of the hash table is presented in Section 6.2.

6.2 Open-addressing lock-free hash table

The hash table suggested in Section 5.3.3 can be implemented as *lock-free*. A concurrent object implementation is *lock-free* if it guarantees that some threads will complete an operation in a finite number of steps, regardless of the relative execution speeds of the threads (Herlihy & Moss, 1993). The current thesis proposes the lock-free implementation of the hash table. The parallelized version of

the hash table is shown in Listings 11-15. For the thread-safety *AtomicInteger* and *AtomicLongArray* are employed in implementation. The following symbols are used in implementation.

- *hash*(k) is a method which represents a hash function.
- *getIndex(h)* is a method which normalizes the calculated hash code.
- *getNextIndex(step, index)* is a method which counts the next index in case of collision.
- *keys* is an array of keys; the type of the array is *AtomicLongArray*, implements atomic *Compare-And-Set* operations on arrays of longs.
- *values* is an array of values; the type of the array is *AtomicReferenceArray*, implements atomic *Compare-And-Set* operations on arrays of references.
- *NO_KEY* represents a value which corresponds to empty cell.
- *maxChainLength* is a maximal number of iterations is done during insertion; the type of *maxChainLength* is *AtomicInteger*, which implements atomic Compare-And-Set operations.
- *put(k, v)* is a method which associates the value *v* with the key *k* and returns the previous value associated with the key *k* or null if it does not exist.
- *putIfAbsent(k, v)* is a method which associates the value *v* with the key *k* and returns the value *v* if *k* was not in the hash table, or the existing value *v*' associated with the key *k*.
- *get(k)* is a method which returns the value associated with the key *k* or null otherwise.

LISTING 11 Key insertion

```
 1 int insertKey(int key)
 2     int step = 1;
 3     int index = getIndex(hash(key));
 4
 5     do
 6         if (keys.compareAndSet(index, NO_KEY, key))
 7             maxChainLength.updateAndGet(v → max(v, step));
 8             return index;
 9         else if (keys.get(index) == key)
10             maxChainLength.updateAndGet(v → max(v, step));
11             return index;
12
13         index = getNextIndex(step++, index);
14     while (true);
```

The *put* and the *putIfAbsent* methods look up for a free position or check whether the key has been added before. The Herlihy construction is utilized to put value atomically (Herlihy & Moss, 1993). The Herlihy construction is comprised of the sequence of operations: copy pointer, modify and save it if it has not been changed since copy. The sequence of operations repeats until the pointer's content was not changed after modification by another thread.

LISTING 12 The *put* method

```
 1 V put(long key, V value)
 2     int index = insertKey(key);
 3     AtomicInteger count = operations.get(index);
 4
 5     while (true)
 6         V v = values.get(index);
 7         if (values.compareAndSet(index, v, value))
 8             if (count.get() > 0)
 9                 synchronized (count)
10                     count.notifyAll();
11             return v;
```

LISTING 13 The *putIfAbsent* method

```
 1 V putIfAbsent(long key, V value)
 2     int index = insertKey(key);
 3     AtomicInteger count = operations.get(index);
 4
 5     if (!values.compareAndSet(index, null, value)
 6         return values.get(index);
 7     if (count.get() > 0)
 8         synchronized (count)
 9             count.notifyAll();
10     return value;
```

LISTING 14 Retrieving of the index of a key

```
 1 int indexOf(long key)
 2     int step = 1;
 3     int index = getIndex(hash(key));
 4     do
 5         long cur = keys.get(index);
 6         if (cur == key)
 7             return index;
 8         else if (cur == NO_KEY) {
 9             return -1;
10
11         index = getNextIndex(step++, index);
12     while (step <= maxChainLength.get());
13     return -1;
```

LISTING 15 The *get* method

```
 1 V get(long key)
 2     int index = indexOf(key);
 3     if (index == -1) {
 4         return null;
 5     if (values.get(index) != null)
 6         return values.get(index);
 7
 8     AtomicInteger count = operations.get(index);
 9     count.incrementAndGet();
10
11     if (values.get(index) == null)
12         syncronized (count)
13             while (values.get(index) == null)
14                 count.wait();
15     count.decrementAndGet();
16     return values.get(index);
```

The *get* method looks through the hash table. Iterations stop as soon as a free position is encountered, the key has been put to the hash table or the number of done steps is more than *maxChainLength*. Thereafter, the method retrieves the value by index. The following situation may occur when a new key has been

added, but its associated value has not been put to the hash table and the value is queried by the key. Thus, the *get* method has to wait until the value is added to the hash table. To handle the situation, locking is used. This kind of synchronization between threads takes significant time in comparison with lock-free algorithms. Thus, an array of *AtomicInteger*'s counts the number of *get* operations pending for values which are going to be inserted by *put* or the *putIfAbsent* operations. If there are no *get* operations waiting for a value, the *put* and *putIfAbsent* methods are lock-free algorithms, as the synchronization block is utilized only if the counter is more than zero. Such construction as "check a statement, get a lock, check a statement" is called double-checked locking.

Let us prove that the operations on the hash table are atomic. To prove that the operations are atomic (linearizable), which means that each operation appears to take effect instantaneously at some point between the operations' invocation and response, the *happens-before* concept is employed (Herlihy & Wing, 1990). The *happens-before* relation on events is the smallest relation satisfying the following conditions.

- If e_i and e_j are events from the same thread and e_i happens before e_j in the sequence of events, then $e_i < e_j$.
- If e_i is the sending of a message and e_j is reception of the message, then $e_i < e_j$.
- $<$ is transitively closed.

Let *A* and *B* be operations which are being executed on a shared resource simultaneously. Then operation *A* can see only the states of the shared resource that existed before operation *B* or after that. In addition, if event *A* happens-before event *B*, then it is guaranteed that all changes done by *A* are observable by *B*.

Depending on the memory model of programming languages the *happens-before* relation between threads is defined differently. As for JVM-based languages, the Java Memory Model (Gosling, Joy, Steele, Bracha, & Buckley, 2015) mentions that:

1. An unlock on a monitor *happens-before* every subsequent lock on that monitor.
2. A write to a volatile field *happens-before* every subsequent read of that field.
3. A call to start() on a thread *happens-before* any actions in the started thread.
4. All actions in a thread *happen-before* any other thread successfully returns from a join() on that thread.
5. The default initialization of any object *happens-before* any other actions (other than default-writes) of a program.

It is obvious that insertions of different keys are linearizable, since *compareAndSet* and *get* of *AtomicLongArray* are atomic operations. The happens-before

relations between the operations are guaranteed because of the second point of Java Memory Model, since the read and write operations are done on volatile fields. Thus, if two concurrent operations see the *NO_KEY* value in the same index, only one *compareAndSet* may succeed. Therefore one of the operations succeeds in setting its value and increases *maxChainLength* if it is necessary. The unsuccessful operation will try another cell to save its value. If the value has been already saved, the *insertKey* method returns the index of the value.

Let us prove that *put(k, v_1)*, *put(k, v_2)* are linearizable. *put(k, v_1)* may return either null or v_2 as the previous value and the associated value of the key k is v_1. *put(k, v_2)* may return either null or v_1 as the previous value and the associated value of the key k is v_2. First, both operations save k to the same cell of the key array. Then both operations try to save their own value to the cell of the array of values. As it said before, *compareAndSet* is atomic; therefore, only one concurrent operation may succeed in saving its value. Thus, the first successful operation will put its value to the array of values and will return null, while the second operation will put its value and will return the previously saved value.

Let us prove that *put(k, v_1)*, *putIfAbsent(k, v_2)* are linearizable. *put(k, v_1)* may return either null or v_2 as the previous and the associated value of the key k is v_1. *putIfAbsent(k, v_2)* may return v_1, in this case the associated value of the key k is v_1, or v_2, in this case the associated value of the key k is v_2. The same as above, both operations save k to the same cell of the key array. The *putIfAbsent* operation checks whether the value is null. If it is null, then *putIfAbsent* saves its value and returns it. Since *compareAndSet* is employed, the check and save are done atomically. If the key has already had a value, then *putIfAbsent* returns the associated value. The proof for the put operation may be done in the same way.

Let us prove that *put(k, v)* and *get(k)* are linearizable. *Put(k, v)* returns v and associates the key k with the value v. *get(k)* may return either null or v. If the *put* operation has already saved the key k to the array of keys, but the associated value does not exist, *get* operations waits until *put* or *putIfAbsent* saves something to the appropriate cell. When the value is saved, the *get* operation is notified. The operations are linearizable since there is the happens-before relation between unlocks and locks on the same monitor. The proof for *putIfAbsent* and *get* can be done in the similar way.

Finally, when a key is associated with a value, then *maxChainLength* has correct value which means that the *get* operation has enough steps to iterate until encountering with the queried key. There is a situation when the key is saved to the array of keys, but *maxChainLength* is not enough to iterate to the saved key. Nevertheless, in this case the key does not have the associated value, as *updateAndGet* has not been called. The case when *maxChainLength* is enough for accessing by *get*, but the value is not saved to the array of values is covered by the previous proof.

Thus, the content is always consistent. Moreover, the hash table is lock-free, as locks are not used in the implementation, except the corner case when a key has already been inserted in the array, but the associated value is not in the map yet, and another thread queries the associated value of the key. Nevertheless,

the implementation of the shortest path searching algorithm does not use *put* and *get* operations concurrently, thus, the algorithm is lock-free.

6.3 Evaluation of performance of the second version of Atlas+

The accuracy of the second version of *Atlas*+ is not impacted by the improvements suggested in Section 6.1. Therefore, the following section is devoted only the analysis of the performance of *Atlas*+. Table 9 shows the performance of the second version of *Atlas*+.

TABLE 9 Performance of the second version of the algorithm

Algorithm	**Atlas algorithm**	**Network query**	**Building of a hash table**	**BFS**	**Total**
Second version	< 1 ms	32 ms	35 ms	9 ms	76 ms
Second version + lock-free hash table	< 1 ms	32 ms	20 ms	9 ms	51 ms

According to the scale-freeness of social graphs, the shortest paths between vertices have tendency to go through popular vertices. Hence, the algorithm can be accelerated if only some amount of adjacent vertices are queried, not the whole adjacency list. Section 6.1 shows that the network time query can be significantly improved. Also, the less vertices and edges obtained by the network requests, the less time is needed for adding elements to the hash table and running BFS.

Unfortunately, the API of the Odnoklassniki social graph does not support the query of popular adjacent vertices. That is why the performance of the query of popular adjacent vertices was not measured. However, obviously performance of the algorithm would be improved.

7 HANDLING OF DYNAMIC GRAPHS

7.1 Description of modifications of spanning trees

Social networking sites are very dynamic, e.g. 50% of actions of users of social networking site per day relates to changes in their friend lists (Wilson, Boe, Sala, Puttaswamy, & Zhao, 2009). Algorithm for searching the shortest path between two vertices should always return the relevant path. Thus, changes in social graph have to be reflected in spanning trees impacted by them. Rebuilding of all trees takes too much resources and too much time. Building of a tree takes for the Odnoklassniki social networking site 1 hour and 20 minutes on average. Hence, only a part of built trees or a part of a tree should be rebuilt. The current Master's thesis utilizes the replacement strategy suggested in Cao et al. (2011) and suggests local modifications of trees.

Replacement of trees is assumed to be done once a day; and it should take a couple of hours for the graph of the Odnokl assniki social networking site. Local modifications of trees should be done if a spanning tree is not a tree of the breadth-first search. The impacted tree is modified in such a way that it will become a breadth-first search tree again. The following changes can occur in a social graph:

1. adding an edge;
2. adding a vertex;
3. removing an edge;
4. removing a vertex.

Let uv be a new edge between vertices u and v. Adding of a new edge does not impact spanning trees before the difference between the depth of the vertices is not more than one. If the difference is more than one, then the highest vertex should become a child of the second vertex. The tree modification is shown in Fig. 16. In the picture vertex v is deeper than vertex u; vertex w is descendant of vertex u. The modification needs to calculate the depth of the vertices and

change the parent pointer of the highest vertex. Thus, time complexity of the modification is $O(L + 1) = O(L)$ where L is the depth of the tree.

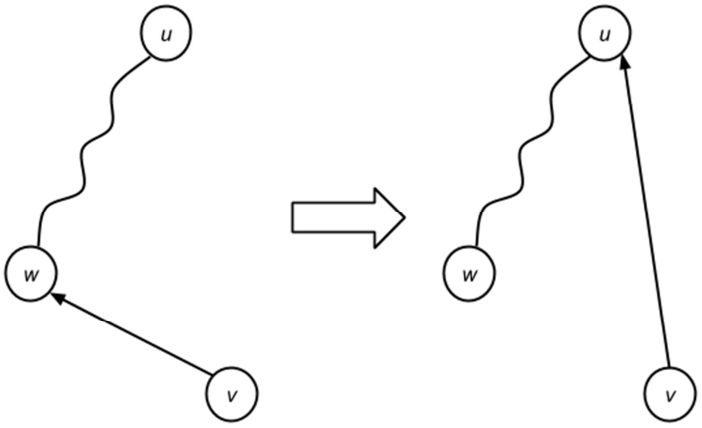

FIGURE 16 Modification for adding an edge

Adding a new vertex does not impact built trees until an edge connecting the vertex and another component of the social graph is added. E.g. this can occur if a new just registered in a social networking site user connects with another user.

Removing an edge from the social graph may split a tree into two unconnected components. Let vertex v be the parent of vertex u in a spanning tree and edge uv was removed. Then such a vertex w should be found that vertex w should be an adjacent vertex of u, vertex w should be connected in the modified tree, and after setting the parent of u to w, the tree should become a breadth-first search tree. Since the depth of a tree should be as small as possible, vertex w is sought in the following groups of the vertices. The adjacent vertices of vertex u are split into three groups: vertices the depth of which equals to the depth of vertex u minus one, the vertices the depth of which equals to the depth of vertex u and the vertices the depth of which equals to the depth of the vertex u plus one. If such a vertex w cannot be found, then such vertex y is found among the adjacent vertices of w for which vertex y is not an ancestor of vertex w. If vertex y does not exist among adjacent vertices of w, then the algorithm is repeated recursively for all adjacent vertices of vertex w until suitable vertex y is found. Let $w_k \dots w_1$ be such a path that the tree contains edge $w_k v$, edge vu be removed from the original graph, w_1 be a vertex which is an adjacent vertex of vertex y and vertex y be suitable. Thus, vertex y should become the parent of w_1 and path $w_k \dots w_1$ should be inverted. A suitable vertex may not be found if all vertices of the subtree rooted at vertex v do not have adjacent vertices in the original graph from another subtree of the spanning tree being modified. This means that edge uv is a *bridge edge (cut-edge)*, an edge of a graph whose deletion from the graph increases its number of connected components (Harary, 1969). Thus, in this case, no modifications are needed. Nevertheless, this scenario very rarely occurs in practice, since the social networks tend not to have just one connection two subgroups of users.

To perform the modification, calculating the depth of some vertices is needed. Since the modification algorithm has to process the whole subtree rooted in vertex v and query the adjacent vertices of all vertices of the subtree in the worst case, the time complexity of modification is $O(E)$, where E is a set of edges in the original graph. The modifications are depicted in Fig. 17-Fig. 18. In the pictures edge between vertices u and v is removed and the tree is modified.

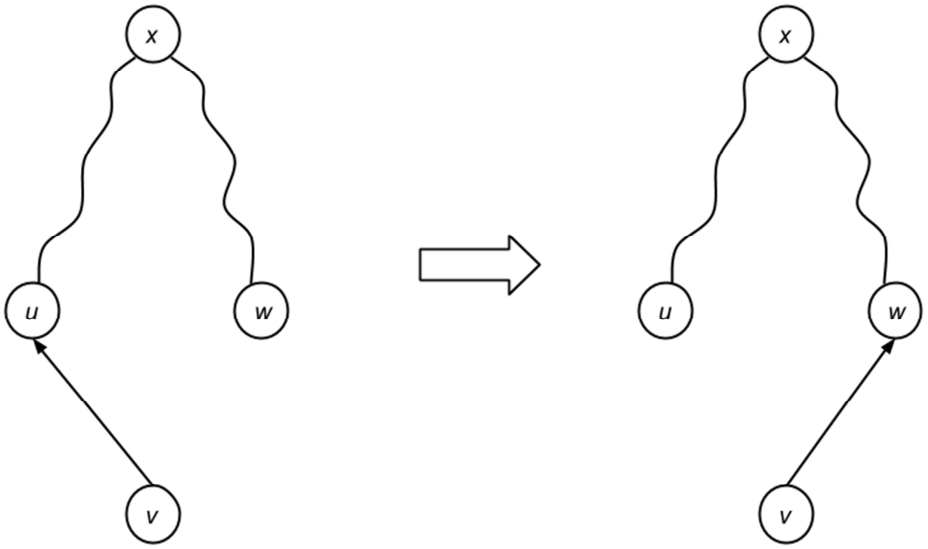

FIGURE 17 Modification for removing an edge

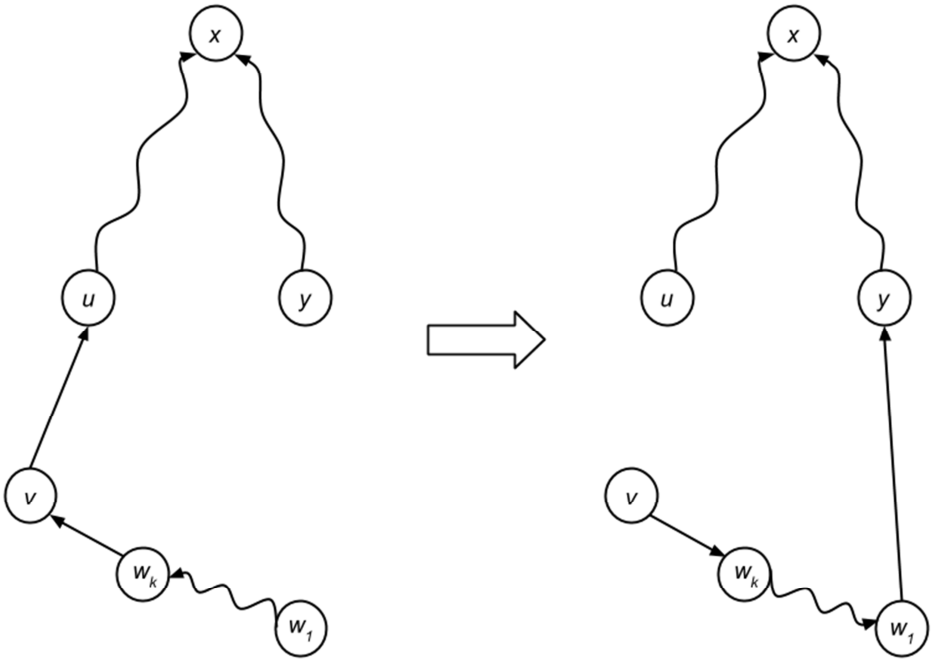

FIGURE 18 Modification for removing an edge (worst case)

Removing a vertex is similar to removing all edges incident to the vertex. Thus, this case is covered by the previous modification.

7.2 Evaluation of accuracy on dynamic graphs

The current section analyzes accuracy of the algorithm on dynamic graphs. Also the section analyzes the proposed modifications of trees to handle changes in the social graph. To analyze the accuracy of *Atlas*+ on dynamic graphs, a subgraph of graph Odnoklassniki, *the graph of Latvia*, is utilized. The graph of Latvia is comprised of users which mentioned Latvia as their origin country as vertices and ties between them as edges. The subgraph contains 515000 of vertices and 25 million of edges. To emulate dynamics of the graph, a log of changes occurred in the graph during a week, is utilized. The log of changes includes only adding and removing edges. Hence, two graphs appear: the graph of Latvia, the graph with users from Latvia as vertices and ties between them as edges, and the graph of Latvia with the applied log of changes.

As was mentioned above, spanning trees should be changed in case of adding an edge for which the difference in depth of vertices of the edge is more than one or in case of removing an edge which is presented in the trees. Table 10 shows number of added edges grouped by difference in depth. Thus, trees are impacted by adding of new edges only in 0.03%. Concerning removals of edges, only 0.07% of removals of edges impact the built trees. Thus, the built trees still are able to approximate the initial graph.

TABLE 10 Difference of depth of the vertices of edges

Difference	0	1	2	3
Adding edge	54.17	45.8	0.03	0

Local modifications of trees are evaluated as follows. First, 20000 of shortest paths were calculated in both the graph of Latvia and the modified graph of Latvia. Thereafter, 30 spanning trees were built for the graph of Latvia. The accuracy of the proposed algorithm was measured on the initial graph (97%) and the modified graph (95%). After that, the modifications suggested in Section 7.1 were applied to the built spanning trees. Using the modified spanning trees, the accuracy of the algorithm is 96%. Thus, the local modifications increase accuracy of the algorithm slightly.

The accuracy of the algorithm grouped by length of shortest paths is depicted in Fig. 19. According to the diagram, changes in the graph influence the accuracy of the algorithm on short paths (3 edges), while the accuracy on longer paths (more than 4 edges) does not change considerably. Local modifications of trees increase accuracy of the algorithm on short paths.

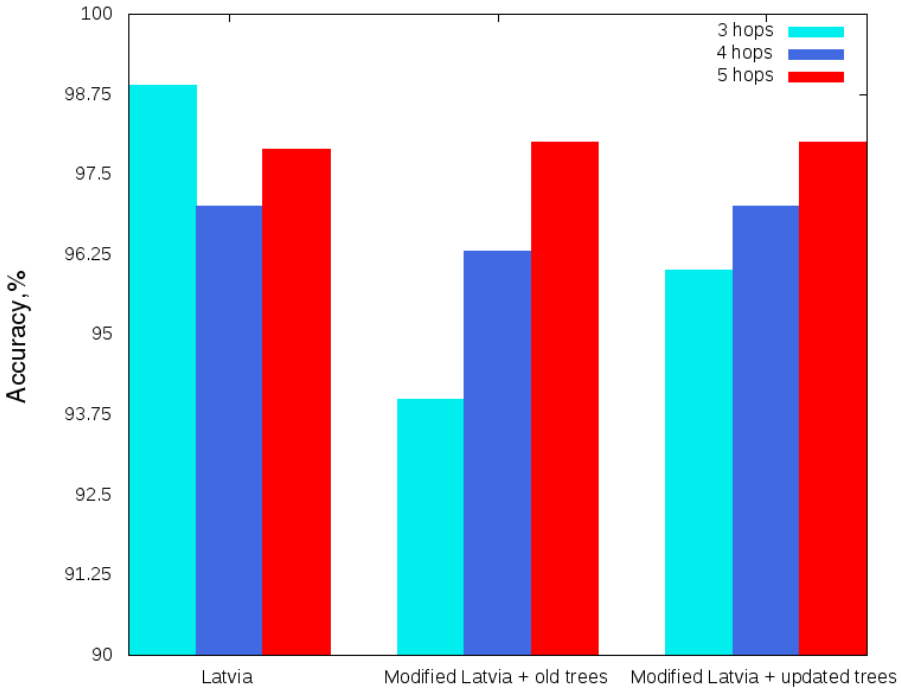

FIGURE 19 Accuracy of the algorithm (local modifications of spanning trees)

The replacement strategy is evaluated as follows. As well as for local modifications, 20000 of shortest paths were calculated in the graph of Latvia and in the modified graph of Latvia. Thereafter, some number of old trees is replaced with new trees. Fig. 20 demonstrates the accuracy of the algorithm depending on the number of replaced trees. According to the picture, the replacement of 14 trees increases accuracy of the algorithm:

- 95.3% against 98% on paths whose length is three edges;
- 96.8% against 97% on paths whose length is four edges;
- 98% against 97.9% on paths whose length is five edges.

Since *Atlas+* has not been deployed into the API of the Odnoklassniki social networking site, modifications and replacements of the spanning trees have not been done for large dynamic graphs. Fig. 21 shows degradation of the accuracy of the algorithm on Odnoklassniki during a month without any changes in the spanning trees. The plot shows that the accuracy of the algorithm decreases in 3% for two weeks. As 25 spanning trees are supposed to be used, two weeks are enough to replace all 25 trees if two trees are replaced per day. Thus, replacement of old trees improves accuracy of the algorithm; the algorithm is able to handle large dynamic graphs.

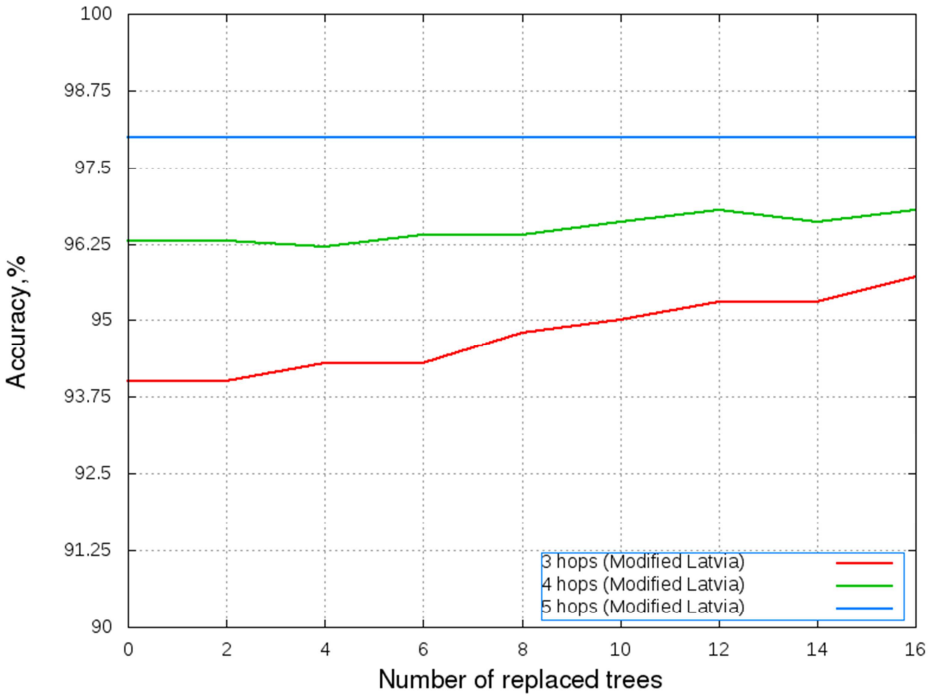

FIGURE 20 Accuracy of the algorithm (replacement of spanning trees)

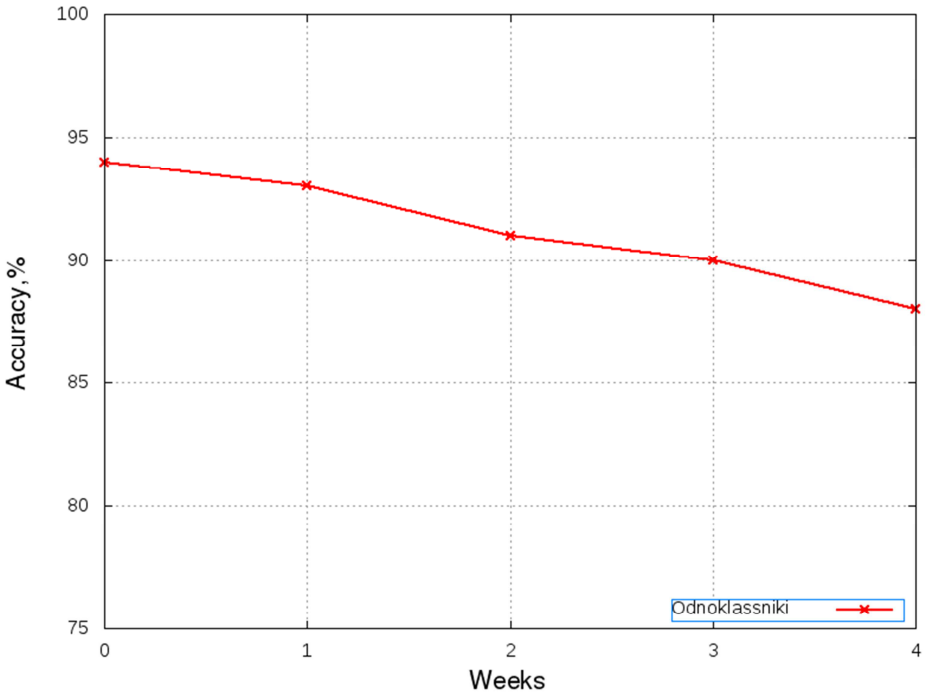

FIGURE 21 Accuracy of the algorithm depending on age of trees (Odnoklassniki)

8 DISCUSSION

The beginning of the Master's thesis is devoted to the review of several fields which are important for creation of the new algorithm that is able to solve the shortest path problem for social graphs more efficiently than the existing solutions. First, the knowledge about the graph theory was obtained; Section 3 tells about the base of the graph theory and the terminology employed in the Master's thesis is given. Also, Section 3 reviews characteristics of social graphs constructed by social networking sites. Further, existing shortest path searching algorithms were analyzed according to the characteristics of social graphs and requirements for the desirable algorithm (Section 4). From the analysis, none of the algorithms fits the defined requirements. The obtained knowledge allowed answering the first and the second research questions which are:

- *Which characteristics of model, social graph, can be used in development of the new algorithm?*
- *Which existing algorithms can be used for efficient solving the shortest path problem in social graphs?*

The rest of the thesis (Sections 5-7) is devoted to the remaining research questions which are:

- *How to design the shortest path searching algorithm which fits the formulated requirements?*
- *What is the impact of changes in a social graph to the accuracy of the synthesized algorithm? How can the algorithm be modified to support dynamic social graphs?*

The proposed algorithm, *Atlas*+, is based on the *Atlas* algorithm which approaches social graphs by a set of spanning trees. The *Atlas* algorithm is comprised of two phases which are pre-computation of a searching index, a set of spanning trees, and queries to it. The queries are searching of the least common ancestor of a pair of vertices in the spanning trees. The result path is obtained as the concatenation of the paths from each vertex to the least common ancestor.

The *Atlas*+ algorithm utilizes the first phase of the *Atlas* algorithm as it is, while the second phase was changed based on the characteristics of social graphs. The proposed algorithm was evaluated on several social graphs: locally sampled social graphs, obtained from SNAP, and the social graph of the Odnoklassniki social networking site. The first version of the proposed algorithm has perfect accuracy, more than 90%, the result of the algorithm may not be longer than in an edge than the shortest path. 15 GBs of the disk memory is used for the searching index which is acceptable. Nevertheless, it turned out that the performance of the new algorithm was unacceptable, response time is more than 150 ms per query; therefore, bottlenecks of the algorithm were analyzed. The thesis analyzed four main steps of the algorithm which are searching for the least common ancestor of a pair of vertices, query of the adjacency lists of the previously obtained vertices, construction of a subgraph comprised of the obtained adjacency lists and searching for the shortest path by the breadth-first search in the built subgraph. Based on the analysis, the second version of the algorithm was designed. Also the algorithm was parallelized which improved performance of the algorithm essentially. Furthermore, the thesis proposes the implementation of a lock-free hash table, which is utilized in one of the steps of the algorithm. The mentioned modifications of the algorithm improved the performance of the last two steps of the algorithm essentially. Also, the improvement of queries fetching adjacency lists, that would decrease the volume of data transmitted via a network, was discussed. Nevertheless, the analysis of its performance was not performed because the API of the Odnoklassniki social network site does not support queries of partial adjacency lists. It is argued, however, the accuracy of the algorithm would not decrease, should the mentioned improvement be realized. Anyway, if the API of Odnoklassniki allowed to query of most popular adjacent vertices of a vertex, the speed of the algorithm could be increased.

Social networks of social networking sites are very dynamic. Therefore, each change in the social graph should possibly be reflected in the built spanning trees. The analysis of this aspect revealed that, first, changes in the social graph do not influence the accuracy of the proposed algorithm essentially. According to the analysis, the accuracy of the algorithm decreases in 5% during a month without modifications of the search index. Anyway, to handle the dynamics of social graphs, two approaches were utilized. The first approach suggests the replacement of several old spanning trees by the same number of new spanning trees. The second approach resorts to local modifications of the spanning trees. The two approaches were evaluated on the social graph modelling Odnoklassniki and it was shown that replacement of circa two spanning trees per day is enough to keep the accuracy of the algorithm on the desired level. Local modifications of the spanning trees also slightly improve the accuracy of the algorithm.

Overall, the proposed algorithm was implemented in the *Java* programming language, and currently it is under deployment into the API of the Odnoklassniki social graph. It might be utilized in the development of a service that finds

the shortest path between a pair of users on the social networking site. In the future work, the time of the network queries can be investigated more precisely. In addition, the algorithm needs to be shipped with the API of a social network site in order to investigate the impact of the dynamics of social networks on the algorithm. The proposed algorithm might also be extended to answer top k shortest paths between a pair of vertices.

REFERENCES

Adamic, L., Buyukkokten, O., & Adar, E. (2003). A social network caught in the web. *First Monday, 8(6)*.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1976). On finding lowest common ancestors in trees. *SIAM Journal on computing, 5(1)*, 115–132.

Althöfer, I., Das, G., Dobkin, D., Joseph, D., & Soares, J. (1993). On sparse spanners of weighted graphs. *Discrete & Computational Geometry, 9(1)*, 81–100.

API OK. (2015, February 15). Retrieved from API OK: https://apiok.ru/wiki/display/api/friends.get

Bach, M. J. (1986). *The design of the UNIX operating system. Vol. 5*. Englewood Cliffs: NJ: Prentice-Hall.

Barabasi, A. L., & Albert, R. (1999). Emergence of scaling in random networks. *Science, 286(5439)*, 509–512.

Barabasi, B. A., & Bonabeau, E. (2003). Scale-free networks. *Scientific American, 288.5*, 50-59.

Bellman, R. (1956). On a Routing Problem. *Quarterly of Applied Mathematics, 16*, 87–90.

Bender, M. A., & Farach-Colton, M. (2000). The LCA problem revisited. *Theoretical Informatics. Springer Berlin Heidelberg*, 88-94.

Benington, H. (1983). Production of Large Computer Programs. *Annals of the History of Computing 5(4)*, 350–361.

Boehm, B. (1986). A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes 11(4)*, 14–24.

Braitenberg, V., & Schüz, A. (1991). Anatomy of the cortex: Statistics and geometry. *Springer-Verlag Publishing*.

Cantor, G. (1877). Ein Beitrag zur Mannigfaltigkeitslehre. *Journal für die reine und angewandte Mathematik, 84*, 242–258.

Cao, L., Zhao, X., Zheng, H., & Zhao, B. Y. (2013, September). *Atlas: Approximating shortest paths in social graphs*. Retrieved from Computer Science, UC Santa Barbara: https://www.cs.ucsb.edu/research/tech-reports/2011-09

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms.* Cambridge: MIT press.

Cox, B. J. (1986). *Object oriented programming: an evolutionary approach.* Reading: Addison-Wesley.

Creswell, J. (2007). *Research Design: Qualitative, Quantitative, and Mixed Method Approaches.* Thousand Oaks: Sage Publications.

D'Andrea, A., Ferri, F., & Grifoni, P. (2010). An overview of methods for virtual social networks analysis. *Springer London*, 3–25.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik, 1(1)*, 269–271.

Dor, D., Halperin, S., & Zwick, U. (2000). All-pairs almost shortest paths. *SIAM Journal on Computing, 29(5)*, 1740–1759.

Elkin, M., & Peleg, D. (2004). (1+a, β)-spanner constructions for general graphs. *SIAM Journal on Computing, 33(3)*, 608–631.

Faloutsos, M., Faloutsos, P., & Faloutsos, C. (1999). On power-law relationships of the internet topology. *In ACM SIGCOMM Computer Communication Review 29, 4. ACM*, 251–262.

Floyd, R. W. (1962). Algorithm 97: shortest path. *Communications of the ACM, 5(6)*, 345.

Ford, L. R. (1956). Network Flow Theory. *Rand Corporation Santa Monica California*.

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM), 34(3)*, 596–615.

Freeman, L. C., Roeder, D., & Mulholland, R. R. (1980). Centrality in social networks: II. Experimental results. *Social networks, 2(2)*, 119-141.

Fu, L., & Deng, J. (2013). Graph calculus: Scalable shortest path analytics for large social graphs through core net. *IEEE/WIC/ACM International Joint Conferences. IEEE*, 417–424.

Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. *Springer Berlin Heidelberg*, 319–333.

Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015, February 13). *Chapter 12. Execution*. Retrieved from Java Language Specification: https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html

Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015, February 13). *Chapter 17. Threads and Locks.* Retrieved May 2015, from Java Language Specification: https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html

Greenwald, G., & MacAskill, E. (2013). NSA Prism program taps in to user data of Apple, Google and others. *The Guardian, 7(6)*, 1-43.

Harary, F. (1969). *Graph theory.* Reading, MA: Addison-Wesley.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on, 4(2)*, 100–107.

Herlihy, M. P., & Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS), 12(3)*, 463-492.

Herlihy, M., & Moss, J. E. (1993). Transactional memory: Architectural support for lock-free data structures. *Proceedings of the 20th annual international symposium on computer architecture*, 289-300.

Hevner, A., & Chatterjee, S. (2010). Design science research in information systems. *Springer US*.

Hevner, A., March, S., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS quarterly 28, no. 1*, 75–105.

Johnson, D. B. (1977). Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM), 24(1)*, 1–13.

Kleinberg, J., Slivkins, A., & Wexler, T. (2004). Triangulation and embedding using small sets of beacons. *In Proceedings 45th Annual IEEE Symposium on IEEE*, 444–453.

Knuth, D. E. (1976). Big omicron and big omega and big theta. *ACM Sigact News, 8(2)*, 18–24.

Larman, C., & Basili, V. R. (2003). Iterative and incremental development: A brief history. *Computer, 36(6)*, 47-56.

Lee, C. Y. (1961). An Algorithm for Path Connections and Its Applications. *Cambridge : MIT press, IRE Transactions on Electronic Computers*, 346-365.

Li, L., Alderson, D., Doyle, J. C., & Willinger, W. (2005). Towards a theory of scale-free graphs: Definition, properties, and implications. *Internet Mathematics, 2(4)*, 431–523.

Marcus, S., Moy, M., & Coffman, T. (2007). Social network analysis. *Mining graph data*, 443-467.

Milosavljević, N. (2012). On optimal preprocessing for contraction hierarchies. *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science. ACM.*, 33-38.

Pai, V. S., Druschel, P., & Zwaenepoel, W. (1999). IO-Lite: A unified I/O buffering and caching system. *In OSDI , Vol. 99*, 15-28.

Potamias, M., Bonchi, F., Castillo, C., & Gionis, A. (2009). Fast shortest path distance estimation in large networks. *In Proceedings of the 18th ACM conference on Information and knowledge management. ACM*, 867–876.

Semenov, A. (2013). Principles of social media monitoring and analysis software. *Jyväskylä Studies in Computing*, 168.

Stanford Network Analysis Project. (2015, May 14). Retrieved May 14, 2015, from http://snap.stanford.edu

Statista. (2015). Retrieved from Statista: http://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/

Tarjan, R. E. (1976). Edge-disjoint spanning trees and depth-first search. *Acta Informatica, 6(2)*, 171-185.

Trove. (2015, February 13). Retrieved from High Performance Collections for Java: http://trove.starlight-systems.com/

Tumasjan, A., Sprenger, T. O., Sandner, P. G., & Welpe, I. M. (2010). Election Forecasts With Twitter: How 140 Characters Reflect the Political Landscape. *Social Science Computer Review, Vol. 29*, 402-418.

Ugander, J., Karrer, B., Backstrom, L., & Marlow, C. (2011). *The anatomy of the Facebook social graph.* Retrieved from http://arxiv.org/abs/1111.4503

Wang, H., Can, D., Kazemzadeh, A., Bar, F., & Narayanan, S. (2012). A system for real-time twitter sentiment analysis of 2012 US presidential election cycle. *Proceedings of the ACL 2012 System Demonstrations*, 115-120.

Warshall, S. (1962). A theorem on boolean matrices. *Journal of the ACM (JACM), 9(1)*, 11–12.

Wilson, C., Boe, B., Sala, A., Puttaswamy, K. P., & Zhao, B. Y. (2009). User interactions in social networks and their implications. *In Proceedings of the 4th ACM European conference on Computer systems*, 205–218.

Wolfgang, P. (1994). *Design patterns for object-oriented software development.* Reading: Addison-Wesley.

Zhang, Z. M., Salerno, J. J., & Yu, P. S. (2003). Applying data mining in investigating money laundering crimes. *In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 747-752.

Zhao, X., Sala, A., Wilson, C., Zheng, H., & Zhao, B. Y. (2010). Orion: shortest path estimation for large social graphs. *Proceedings of the 3rd Wonference on Online social networks*, 9.