# Counting triangles in large graphs

**Boris Radovič**[a,1], **Aljaž Verlič**[a], **and Samo Metličar**[a]

[a]University of Ljubljana, Faculty of Computer and Information Science, Večna pot 113, SI-1000 Ljubljana, Slovenia

The manuscript was compiled on June 21, 2021

**In this paper, we present the problem of triangle-counting in large networks, a task that has drawn researchers' attention for the past two decades. We first motivate the reasons for such interest and later on review the most recently developed algorithms that were proposed for this task. The performed tests reveal that the Delegating Low Degree Vertices algorithm and its variants are the algorithms that perform best on medium-size networks, i.e. networks that fit into memory. On the other hand, especially for large graphs, the usage of streaming and other types of approximation algorithms is an unavoidable choice that nevertheless still provides acceptable results. All the code developed for the purposes of this project is available on https://github.com/BorisRado/triangle-counting.**

**Triangles are probably the most basic non-trivial structures that populate graphs.** They consist of three nodes that are connected to each other to form a closed triplet, i.e. a ring of length 3. Counting how many such structures are present in a given graph has been proven to be extremely useful by countless empirical results. For example, triangles may be an important feature for differentiating between spam and legal websites (1) and more in general, for grouping websites based on their content (2); they may also be employed for community detection purposes (3), or even to estimate content quality in social networks (1). Furthermore, there are many measures that either directly or indirectly evaluate the presence of such structures in graphs – clustering coefficient and transitivity coefficient are just two of these –, so in short, counting triangles is a task of primary importance.

It therefore comes as no surprise that many algorithms have been proposed to tackle this task, that is in fact extremely complex considering the huge dimension of today's graphs. Even more, the graphs we face today are constantly changing: consider for example how social networks evolve as new people join the network and form new relations with each other, or even how the web continually changes by the additions and removals of websites. As most researchers in their papers claim that the algorithm they propose outperforms any foregoing algorithm, there is the urge to perform a thorough and unbiased comparison between all these algorithms. In this paper we shall therefore perform such a study, comparing the time and space requirements, the simplicity with which they may be coded, the ability to evolve to dynamic graphs, and other relevant metrics of some selected state-of-the-art algorithms.

## Related work

Throughout this section we denote with $G = (V, E)$ the graph consisting of $n = |V|$ nodes and $m = |E|$ edges. The algorithms we present attempt to calculate the number of triangles, denoted by $\Delta(G)$. Also, $d_v$ denotes the degree of the vertex $v \in V$, while $k_i$ is the $i$-th largest node degree present in the graph.

The most naive algorithm we can think of to count triangles is exhaustive search: by listing all the (not necessarily connected) triplets in a graph, we may count whether they are connected between each other or not. Considering there are $\binom{n}{3}$ such triplets, we derive that the asymptotic time complexity of such approach is $O(n^3)$ - of course, this reasoning is valid if we assume that checking if a set of three nodes forms a triangle takes constant time $O(1)$, which is achievable by representing the neighbourhoods of nodes as sets.

A considerable improvement of this strategy, called *Node-iterator* algorithm (4), is achieved by enumerating all the pairs of neighbors of each node $n$. For each such pair, the algorithm verifies if there exists an edge that connects the pair of nodes – if there is, the algorithm outputs the triangle formed by the pair of nodes and the base node $n$. This algorithm offers a consistent improvement in terms of time complexity, as this decreases to $O(\sum_{v \in V} d_v^2)$ (5). It is worth noting that such a strategy causes each triangle to be detected three times, fact that leaves room for substantial improvements. For example, the algorithm might be further optimized by delegating low degree vertices, i.e. by searching the triangle candidates only on nodes with a sufficiently low degree. One such algorithm is *Delegating Low-Degree Vertices* (*DLDV*) described in (5). Its performance can be further improved at the cost of doubling the memory usage – this algorithm gains the *Plus* suffix at the end of the name, and we denote it as *DLDV+*. Another algorithm that takes advantage of the low-degree vertex delegation is the *forward algorithm* which may be further improved into the *compact* forward algorithm (6). It might be proven, that the time complexity of both these algorithms is $O(m^{\frac{3}{2}})$, whereas the space they require is linear in the number of edges (6). Another similar approach suggests iterating through all edges, and for each edge check the size of the intersection of the neighborhoods of the two nodes, which can be done, assuming that the set of nodes in the neighborhoods are sorted, in $O(d_u + d_v)$ time - this algorithm takes the name *Edge-iterator*.

A different type of approach is the one that leverages the properties of the adjacency matrix $A$ of the graph. In (7) authors show that the number of cycles of length 3 (i.e. triangles) can be computed using Equation 1,

$$\Delta(G) = \frac{1}{6}\mathrm{tr}(A^3) \qquad [1]$$

where tr is the trace operator of a matrix. The proposed method, called *Cycle counting*, runs in $O(n^\omega)$ time, where $\omega < 2.376$ is the exponent of matrix multiplication. It works especially well on sparse graphs, property that should make the algorithm suitable for real networks.

All authors contributed equally to this work.

[1]To whom correspondence should be addressed. E-mail: br8097@student.uni-lj.si.

Another algorithm that uses the adjacency matrix is proposed in (8) and is called *Sparse matrix with Hadamard product*. It avoids calculating the third power of the adjacency matrix by using Equation 2:

$$\Delta(G) = \frac{1}{6}\sum_{ij}(A^2 \circ A)_{ij} \quad [2]$$

where $\circ$ denotes element-wise product. A related approach, called *MiniTri* (9), follows a similar strategy, though this algorithm also uses the incidence matrix and consequently uses twice as much memory. The pitfall of all three of these approaches, and in general of any technique that requires the graph to be represented as a matrix, is evident: the size of the adjacency matrices are bound to be too large to be held in memory even for modest-size networks. This, however, can be avoided if the network is sparse, as in such a case we can use data types specific for sparse matrices and therefore occupy only a linear amount of space. One such type of representation is the *Compressed sparse row* (*CSR*) format.

In this light, we also propose a novel algorithm called *Sparse and Set*, that uses a sparse adjacency matrix as a base datatype. The algorithm also requires an adjacency set representation of the graph, which allows us to check if two nodes are connected in constant time. The algorithm simply uses the sparse adjacency matrix to traverse over all triads* and then checks for the last missing connection to complete a triangle.

In (10) are proposed two algorithms that leverage the property expressed in Equation 3:

$$\Delta(G) = \frac{1}{6}\sum_{j}\lambda_j^3 \quad [3]$$

i.e., the total number of triangles is proportional to the sum of cubes of the eigenvalues of the adjacency matrix. Though this formula might be exploited to compute the exact number of triangles, the authors settle for an approximation using the Lanczos method, an approximation that allows to speed up the execution time and reduce the space complexity to $O(d \cdot m)$ where d represents number of computed eigenvalues. The algorithm yielded very good results under the assumption that the absolute values of the eigenvalues follow a power law, a property that emerges often in real networks.

Nevertheless, the size of modern networks may prevent the networks from even being stored in the main memory. In order to cope with this issue, we might resort to the *stream* or *semi-stream processing paradigm*, i.e. we may renounce to the random access that allows us for example to retrieve the neighbours of a node in constant time, and only read edges in an arbitrarily ordered sequence (11). Of course, we cannot expect algorithms to return the exact result by scanning only a handful of times a sequence of edges, so in (1) were proposed two semi-streaming *approximation* algorithms that perform $k \leq \log(n)$ passes over the secondary storage. The algorithms approximate the Jaccard coefficients, which we know is defined as $c = \frac{|S_u \cap S_v|}{|S_u \cup S_v|}$ where $S_u$ and $S_v$ are sets of nodes representing the neighbourhoods of nodes $u$ and $v$ respectively. The time complexity is linear in the number of edges $O(k \cdot m)$, while the space complexity is of the order $O(n \log(n))$. In (1) is also proposed an improvement of this algorithm that reduces the access to the secondary memory to speed up the execution time by up to 60%. It is worth noting, that both these algorithms not only count the overall number of triangles, but yield the number of triangles incident to each node.

Another approximation algorithm was proposed in (12), in which the number of triangles is estimated through a single pass over the networks' edges by sampling edges to be kept in memory with reservoir sampling. Edges are picked uniformly at random so as to preserve the properties of $\hat{\tau}$, which is a random variable that depends on the number of edges already analyzed, the degree of reservoir edges' endpoints, and the existance of triangles in the remaining stream. In particular, it turns out that the expected value of this random variable is equal to the number of triangles, i.e. $\mathbb{E}[\hat{\tau}] = \Delta(G)$. The time complexity of the algorithm is $O(m + r)$, $r$ being the number of estimators used by the algorithm. It is also an $(\varepsilon, \delta)$-approximation for triangle counting if

$$r \geq \frac{6mk_1}{\varepsilon^2 \Delta(G)} \log\left(\frac{2}{\delta}\right),$$

where $k_1$ is the highest node degree. In this paper, we shall denote this algorithm with the name *Stream Graph Estimate*.

Most recently, researchers managed to approximate the overall number of triangles by using an algorithm running in sub-linear time (13), yielding good results by observing as little as 3% of the graph. The algorithm, called "Random walk estimate", performs a random walk starting from an arbitrarily selected node $v$. After performing a walk of length $R$ and assigning to each traversed edge a weight proportional to the degree of its two endpoints, it randomly selects $l$ visited edges and checks, whether the selected edge and a randomly selected edge incident to either of the two endpoints form a triangle. This strategy has also the non-negligible advantage, that it does not require the presence of the whole graph, nor does it require any prior knowledge about the number of nodes or links in the graph. Therefore, such an algorithm may be used to estimate the triangle density.

Finally, let us just point out that parallelization has also drawn high interest, and most of the algorithms mentioned thus far can be run concurrently (14). In this light, and considering the recent establishment of MapReduce as the de-facto standard for distributed computing, in (15) is proposed an algorithm that divides the graph into multiple overlapping sub-graphs on which the map function is run concurrently. Then, the reduce function is the one that actually counts the triangles with any triangle counting algorithm.

The most relevant characteristics of all the presented methods are summarized in Table 1.

## Methods

All the algorithms introduced in the previous section were implemented from scratch using the Java programming language. This approach allowed us to represent the graph in the way that best suits the needs of the algorithms so as to assure the comparison to be truly unbiased. Therefore, our implementation only uses a number of classes from the `java.util` package, with the only exception represented by the `SparseRealMatrix` class, which was taken from the `apache` math library. This class was used for computing eigen decomposition of the tridiagonal matrix produced by the Lanczos method.

*A triad is a triplet of nodes with at least 2 connections between them.

| Algorithm name | Time complexity | Space complexity | Enumeration | Exact | Parallel |
|---|---|---|---|---|---|
| Naive | $O(n^3)$ | $O(m+n)$ | ✓ | ✓ | ✓ |
| Edge iterator | $O(m \cdot k_2)$ | $O(m+n)$ | ✓ | ✓ | ✓ |
| Forward | $O(m^{\frac{3}{2}})$ | $O(m+n)$ | ✓ | ✓ | ✓ |
| Compact forward | $O(m^{\frac{3}{2}})$ | $O(m+n)$ | ✓ | ✓ | ✓ |
| Stream graph estimate | $O(m+r)$ | $O(r)$ | ✗ | ✗ | ✓ |
| Random walk estimate | $O(r+l)$ | $O(1)$ | ✗ | ✗ | ✗ |
| Cycle counting | $O(n^\omega)$ | $O(m+n)$ | ✓ | ✓ | ✓ |
| Adjacency matrix search | $O(m \cdot n + n^2)$ | $O(n^2)$ | ✓ | ✓ | ✓ |
| Sparse matrix with Hadamard product | $O(m \cdot n)$ | $O(m+n)$ | ✓ | ✓ | ✓ |
| Delegating Low-Degree Vertices | $O(m \cdot k_2)$ | $O(m+n)$ | ✓ | ✓ | ✓ |
| Eigenvalue estimate | $O(d \cdot m)$ | $O(m+n)$ | ✗ | ✗ | ✓ |
| MapReduce | $O(m^{\frac{3}{2}})$ | $O(\delta \cdot m)$ | ✓ | ✓ | ✓ |
| MiniTri | $O(m \cdot n \cdot k_2)$ | $O(m+n)$ | ✓ | ✓ | ✓ |
| Sparse and Set | $O(m \cdot k_1)$ | $O(m+n)$ | ✓ | ✓ | ✓ |

**Table 1. Summary table of the algorithms under observation.**

Regardless of the underlying graph representation structure, nodes were always stored as an `Integer` object. This choice was done in order not to give an advantage to those algorithms, that are able to use primitive `int` values. For example, when representing the neighbourhood of a node using a Java `Set`, we cannot store the node IDs as primitive `int` values – recall that in Java, generic classes, such as `Set` and `List`, cannot be parameterized with primitive types. The advantage we refer to is given by the fact, that storing `int` values requires only 32 bits of memory, while the wrapper `Integer` object take up 128 bits of memory. It goes without saying, that in production situations, a better approach would be to code all the data structures from scratch so as to be able to reduce memory consumption by a factor of 4, and to improve at the same time the performances.

Apart from the bare algorithms, we also coded utility classes that read the pajek files, in which the actual graphs are stored, as well as a class that implements the CSR format. The latter allows us to store a condensed representation of the adjacency matrices and is also equipped with the most commonly used operations, such as vector multiplication.

All the algorithms were evaluated on a system with the following characteristics: Ubuntu 20.04, CPU Intel(R) Core(TM) i5-7200U, 2.50GHz. Also, all the algorithms were executed 14 times in order to increase the accuracy of the estimate on one hand and to estimate the variability of the results on the other. The reported times are hence the average time of these multiple executions.

Let us finally point out, that in this paper, we only focus on snapshots of simple undirected graphs.

## Results

We begin our analysis by observing how the considered algorithms behave on synthetic graphs. In particular, we devote our attention to the Albert-Barabasi graphs, which we know are able to replicate the short distances and the power-low distribution of degrees that characterize many real-world networks. These graphs typically have low – but not trivial – clustering.

The performances of a number of selected exact algorithms are reported in Figure 1. In the figure, we only show the performances of the most promising algorithms, because some of the algorithms under consideration already proved to be severely sluggish even on modest-size networks. These unappealing algorithms not only include the naive search and all the algorithms that require storing the adjacency matrix as-is, but also more involved methods such as the MiniTri algorithm with sparse adjacency matrix representation. In the figure, we can see, that the algorithm that yields the best results is the DLDV+, closely followed by the related DLDV algorithm, that, as discussed already, trades speed for a diminished space complexity. It is quite surprising, that the node iterator, which is one of the most straightforward algorithms, is able perform so closely to these two more involved methods. Also, the edge iterator algorithm performs relatively well on large networks even though it is among the slowest ones on small networks (up to ±20.000 nodes). The algorithm we proposed, i.e. the "Sparse and set" algorithm, yields the expected results, as it outperforms the compact forward algorithm which uses the same amount of memory, and fits in the middle of the pack. The MapReduce algorithm uses Compact forward as its underlying triangle counting algorithm and even though it has utilized all available threads, it did not surpass its sequential counterpart. One of possible reasons is that MapReduce does more redundant triangle calculations and also does some overhead operations such as splitting graphs into sub-partitions in the map process.

As a final note, let us here point out the fact, that all the algorithms are able to scale without difficulty to graphs with several millions nodes even on limited hardware as the one used to carry out the tests.

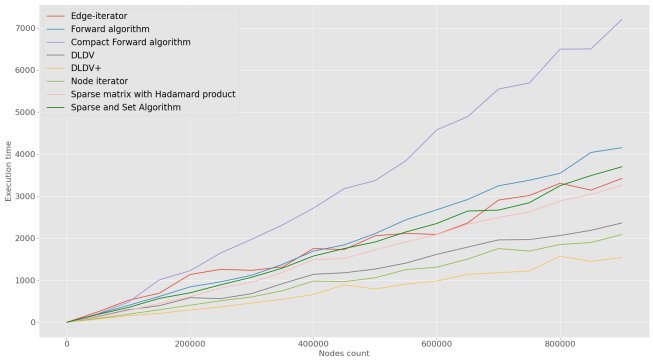


**Fig. 1.** Execution time of selected exact algorithms on Albert-Barabasi graphs.

We next focus our attention on a number of real-world graphs taken from (16). The main characteristics of these graphs are summarized in Table 2, while the performances

of the selected exact algorithms are shown in Figure 2. We can see, that the DLDV+ algorithm confirms itself as being the fastest algorithm in all the graphs, while the other results reserve some surprises: for example, we can see that the difference to the DLDV algorithm is substantially larger than the one observed in the synthetic graphs. Also, the node iterator algorithm, becomes extremely slow when the graph has high density and does not follow the power-law degree distribution of degrees as is the case in the java graph. Nevertheless, this algorithm still performs very well in most situations, and in general it always outperforms the edge-iterator algorithm. Finally, these results confirm that the algorithm we proposed, although it might outperform some algorithms in particular circumstances, is in general not trustworthy for production purposes.

**Table 2. Characteristics of considered real-world networks.**

| Graph | Nodes | Edges | Clustering | Density |
|---|---|---|---|---|
| social | 10680 | 24316 | 0.3780 | $4.264 \cdot 10^{-4}$ |
| aps_2010_2013 | 56473 | 200353 | 0.1401 | $1.526 \cdot 10^{-4}$ |
| facebook | 63731 | 817035 | 0.1477 | $4.023 \cdot 10^{-4}$ |
| java | 2378 | 14619 | 0.0144 | $5.173 \cdot 10^{-3}$ |
| www_google | 875713 | 4322051 | 0.0552 | $1.127 \cdot 10^{-5}$ |

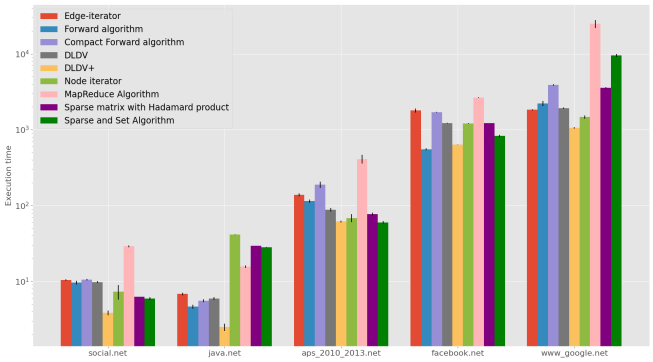


**Fig. 2.** Execution time of the algorithms on a number of real graphs. The height of the bars represents the average execution time, while the black vertical lines represent the standard error of the time estimate.

We also empirically tested approximation algorithms on real networks using different parameters. Table 3 shows the relative error of results obtained using Stream graph estimate algorithm compared to real values at different values of $r$. We observe that by using a high number of estimators, the algorithm works very well for most real networks, except for `www_google`, where the error is noticeable.

**Table 3. Stream graph estimate results**

| | Parameters | | | |
|---|---|---|---|---|
| | $r = \log m$ | $r = \sqrt{m}$ | $r = 0.05m$ | $r = 0.25m$ |
| social | 0.2130 | 0.0880 | 0.0541 | 0.0066 |
| aps_2010_2013 | 0.1080 | 0.0898 | 0.0130 | 0.0031 |
| facebook | 0.4084 | 0.0073 | 0.0201 | 0.0048 |
| java | 0.7588 | 0.2217 | 0.0628 | 0.0603 |
| www_google | 0.2231 | 0.1489 | 0.1397 | 0.1225 |

Similarly, Table 4 shows the relative error of results obtained using Random walk estimate algorithm. The value $R$ represents the number of edges covered by the walk. We notice that after observing 3% of the network we already get very reliable triangle counts, as stated in (13).

**Table 4. Random walk estimate**

| | Parameters | | | |
|---|---|---|---|---|
| | $R = \log m$ | $R = \sqrt{m}$ | $R = 0.03m$ | $R = m$ |
| social | 0.4523 | 0.2531 | 0.0089 | 0.0062 |
| aps_2010_2013 | 0.1943 | 0.0787 | 0.0091 | 0.0042 |
| facebook | 0.6242 | 0.0884 | 0.0076 | 0.0072 |
| java | 0.1119 | 0.0500 | 0.0348 | 0.0066 |
| www_google | 0.1406 | 0.0860 | 0.0063 | 0.0056 |

**Table 5. Eigenvalue estimate**

| | Parameters | | |
|---|---|---|---|
| | $tol = 0.01$ | $tol = 0.03$ | $tol = 0.05$ |
| social | 0.2813 | 0.1173 | 0.0881 |
| aps_2010_2013 | 0.5702 | 0.6102 | 0.6401 |
| facebook | 0.4107 | 0.4684 | 0.5394 |
| java | 8.4919 | 8.3213 | 8.0371 |
| www_google | 0.8751 | 0.8821 | 0.8925 |

Another approach we used for estimating triangles is by calculating top eigenvalues with Lanczos method. The eigenvalue estimate algorithm compared worse to Random walk algorithm. Parameter tol marks tolerance factor which determines how many top eigenvalues to sum (10). In table Table 5 we can see most graphs had high relative error with exception of social network having low error and java network having exceptional difference to correct estimation. Lanczos method is prone to numerical errors and could be one of the reasons for such relative error.

Note that we did not report the execution times of the approximation algorithms, as their usage is more often than not subdued to the graph under consideration: for example, the random walk algorithm will typically find place in contexts, in which we only have access to an API providing us the characteristics of the graph, while streaming algorithms are a forced choice when the graph does not fit into memory.

## Conclusion and future work

In this paper, we compared a number of state-of-the-art algorithms for the task of triangle counting. Regarding the exact algorithms, we empirically showed that the fastest algorithm is the DLDV+ one, which has the only pitfall, that it requires the graph to be stored twice in the main memory. Therefore, when aiming for speed and assuming that memory is not an issue, this should be the algorithm of choice, while if we want to reduce memory usage, there are several valid choices, including the DLDV, the node-iterator, and the forward algorithm.

On the other hand, the approximation algorithms yielded conflicting results: on one hand, the random walk algorithm is able to approximate faithfully the true number of triangles by observing as little as 3% of the graph, and the streaming algorithm performs equally well by taking a sufficiently number of estimators $r$, while on the other hand, the eigenvalue estimation algorithm proved to return unsatisfactory results.

In conclusion, we argue that the simplest algorithms turn out to be the most effective when it comes to exact triangle counting. Therefore, in the future, research should be primarily devoted to approximation algorithms, which are bound to become more and more relevant as the size of today's networks keeps increasing.

1. Luca Becchetti, Luca, Paolo Boldi, Paolo, Carlos Castillo, Carlos, Aristides Gionis, and Aris-

tides. Efficient semi-streaming algorithms for local triangle counting in massive graphs. 08 2008. .
2. Jean-Pierre Eckmann and Elisha Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the National Academy of Sciences of the United States of America*, 99:5825–9, 04 2002. .
3. Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint*, 1111.4503, 11 2011.
4. Thomas Schank. Algorithmic aspects of triangle-based network analysis. 2007.
5. Tim Roughgarden. Cs167: Reading in algorithmscounting triangles. 3 2014.
6. Matthieu Latapy. Theory and practice of triangle problems in very large (sparse (power-law)) graphs. 10 2006.
7. R. Yuster N. Alon and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17: 209–223, March 1997.
8. Paul Burkhardt. Graphing trillions of triangles. *Information Visualization*, 16, 09 2016. .
9. Michael Wolf, Jonathan Berry, and Dylan Stark. A task-based linear algebra building blocks approach for scalable graph analytics. pages 1–6, 09 2015. .
10. Charalampos E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *2008 Eighth IEEE International Conference on Data Mining*, pages 608–617, 2008. .
11. M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *External Memory Algorithms*, 1998.
12. A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment*, 6:1870–1881, 09 2013. .
13. Suman Bera and C. Seshadhri. How to count triangles, without seeing the whole graph, 06 2020.
14. Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017. .
15. Ha-Myung Park and Chin-Wan Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. pages 539–548, 10 2013. .
16. Lovro Šubelj. Graph collection, 06 2021. URL https://lovro.lpt.fri.uni-lj.si/ina/nets/.