Data Structures for Efficient Computation of Influence Maximization and Influence Estimation

Diana Popova
University of Victoria
British Columbia, Canada
dpopova@uvic.ca

Akshay Khot
University of Victoria
British Columbia, Canada
akshay03@uvic.ca

Alex Thomo
University of Victoria
British Columbia, Canada
thomo@uvic.ca

ABSTRACT

Algorithmic problems of computing *influence estimation* and *influence maximization* have been extensively studied for decades. We researched several data structures for implementing the Reverse Influence Sampling method proposed by Borgs, Brautbar, Chayes, and Lucier in 2014. Our implementations solve the problems of influence estimation and influence maximization in large graphs fast and using small memory footprint. For instance, we are able to produce results 3 times faster and scale 8 times more than a state-of-the-art algorithm, all this while preserving the theoretical guarantees of Borgs *et al.* for Reverse Influence Sampling.

1 INTRODUCTION AND DEFINITIONS

The most popular definition of *influence* relies on probabilistic reachability [5, 9]. The network is modeled as a directed graph and a node influence is calculated as the (expected) number of other nodes reachable from it. Given a set of *seeds* (initial nodes), *influence estimation* is calculated as the total number of nodes reachable from all the seeds in the set. To find a set of seeds that gives the maximum influence *spread* (the number of influenced nodes) is the *influence maximization* problem.

Running time and required space are the primary considerations for the algorithms solving influence estimation and influence maximization problems; the networks of interest are usually quite massive in size. Kempe *et al.* [9] showed that the influence maximization problem is NP-hard, and Chen *et al.* [4] showed for the influence estimation problem that computing the exact influence of a single seed is #P-hard. Moreover, as Feige [6] proved in 1998, the problem is hard to approximate to anything better than $1-(1-1/s)^s$ of the optimum for a seed set of size s.

To model the influence spreading, Kempe *et al.* proposed the Independent Cascade (IC) model [9]: Starting from a seed, influence spreads in rounds/steps: each node after getting influenced has one possibility to influence its neighbors. IC selects edges from the seed neighborhood with *independent* probabilities. Influenced neighbors, in their turn, have one possibility to influence their neighbors forming a *cascade* of information propagation. Kempe *et al.* proved that influence maximization on the IC model is *monotone* and *submodular*, and therefore the approximate Greedy algorithm produces near-optimal solutions with a theoretical guarantee. Approximate Greedy starts with an empty seed set S. In each iteration, it adds to S a seed - the node with maximum *marginal* gain. IC became a standard model of information diffusion, and we are using it for our algorithms.

Building on the Kempe *et al.* results, several approximate algorithms with theoretical guarantees have been developed [4, 8, 14, 15, 21, 22]. However, the problem of scalability remains.

Recently, a new approach was proposed by Borgs *et al.* [2]: the Reverse Influence Sampling (RIS) method. RIS selects (uniformly at random, with replacement) a node and finds a set of nodes that *would have influenced* it. The set of found nodes is stored in a structure called *hypergraph*. This process is repeated many times. If a node appears often in sets of "influencers", then this node is a good candidate for the most influential node in the graph. RIS is a faster algorithm for the influence maximization problem, obtaining the near-optimal approximation factor of $(1-1/e-\epsilon)$, for any $\epsilon > 0$, in time $O((m*k*\epsilon^{-2}*log(n))$, where k is the number of seeds. RIS can be modified to allow an early termination: if it is terminated after $O(\beta*m*k*log(n))$ steps, then it returns a solution with an approximation factor that depends on β (the greater the β, the better the approximation is, and the guarantees are made precise in [2]). However, still RIS needs to sample nodes many times and consumes vast amounts of memory. The problem of scalability remains.

We note that there are several works that propose better bounds on the number of samples that need to be taken to achieve the same theoretical approximation (cf. [8, 13, 21]). Our research is orthogonal to these works. We aim at optimizing the computation and storage of sketches in the hypergraph; the aforementioned works aim at reducing the number of sketches needed.

Algorithms in this paper. Our main goal is to scale-up computing of influence maximization and influence estimation to large graphs with tens of millions of edges. We use several data structures all aiming at reducing the required memory and speeding up the computation.

(1) We use Webgraph, a highly efficient, and actively maintained graph compression framework [1].
(2) We design a new way of storing the hypergraph that significantly decreases the required space, without affecting the theoretical guarantee of the approximation.
(3) We conduct experiments on large graphs on a consumer-grade laptop comparing the data structures, and provide a detailed analysis of the results.

2 PRELIMINARIES

Notations

Let $G=(V,E,p)$ be a directed graph, where V is the node set ($|V|=n$), E is the edge set ($|E|=m$), and $p: E \rightarrow [0,1]$ is a probability function on the edges existence. Let S be a set of seeds. The influence spread of a seed set S under the Independent Cascade (IC) model, denoted by $\sigma(S)$, is defined as the expected total number of reachable nodes for S.

IM and IE Problems

Influence Estimation Problem (IE). Given a graph $G=(V,E,p)$ and a seed set $S \subseteq V$, compute the influence spread $\sigma(S)$ of S.

Influence Maximization Problem (IM). Given a graph $G = (V, E, p)$ and an integer k, find a seed set $S \subseteq V$ of size k that maximizes $\sigma(S)$.

3 PROPOSED DATA STRUCTURES

We developed three different algorithms implementing RIS. Each algorithm uses a distinct data structure for storing the hypergraph. We compared the performance of different data structures on a consumer-grade laptop.

3.1 RIS

The original RIS method (Algorithm 1 in [2]) selects nodes uniformly at random. Let v be such a node. Then RIS determines the set of nodes that *would* have influenced v by running a search in the *inverse* graph (the graph with the directions of edges reversed). RIS stores the found set of nodes, called a *sketch*, in a data structure, called a *hypergraph*. The process continues until the hypergraph reaches a pre-defined *weight*.

The weight of the hypergraph is defined as the number of graph edges "touched" by RIS. RIS selects an edge to follow at random, with a given edge probability p. During search, each edge incident to a visited node is counted as "touched" and contributes to the weight calculation. Note, that the edge is considered "touched" regardless of it being selected by the search or not. How large the weight should be in order to guarantee an approximation to the optimal solution is defined in [2].[1]

The RIS hypergraph is a two-dimensional (2D) list that contains, for each node u in the graph, the IDs of the sketches where u appeared. Further, RIS runs an approximate Greedy algorithm on the hypergraph, which returns a set of k *seeds* (nodes with approximately maximal influence in the original graph).

3.2 Two-Dimensional List (2DL)

We started with a straightforward implementation of RIS described in subsection 3.1, where we use a two-dimensional list structure (list of lists) for storing the hypergraph. Algorithm 1 shows the pseudocode of this implementation.

For a better performance, we added the following improvements: 1. The Webgraph [1] format for the input inverse graph (saves space); 2. Java 8 parallel streams and lambda expressions (speeds up performance by executing several reachability procedures in parallel); 3. BitSet structure for marking deleted sketches (speeds up the marginal influence calculation); and 4. Leskovec *et al.* technique [11] (speeds up the seed calculation).

Influence estimation is part of seeds calculation. With minor changes, the code provides the solution for IE problem, when a seed set is inputted.

We compared the runtime and space of 2DL and DIM, a state-of-the-art implementation of RIS by Ohsaka *et al.* ([16]). For both algorithms, we used the same lower bound on the hypergraph weight from [2]. Our 2DL implementation significantly outperforms DIM. Testing results are discussed in detail in subsection 4.1.

3.3 Flat Arrays (FA)

FA implementation modifies the BuildHypergraph procedure of 2DL (subsection 3.2). The pseudocode is shown in Algorithm 2.

To store the hypergraph, FA creates two flat, one-dimensional, arrays of integers: *sketches* and *nodes*. *sketches* stores the sketch

[1]Theorem 4.1 in [2], version 5, updated June 22, 2016.

Algorithm 1 2DL

Input: directed graph G with n nodes and m edges, coefficient β, number of seeds k
Output: seeds set $S \subseteq V$ of size k, spread $\sigma(S)$

```
1:  R ← β * m * k * log(n)
2:  H ← BuildHypergraph(R)
3:  return GetSeeds(H)
4:  procedure BuildHypergraph(R)
5:      while H_weight < R do
6:          v ← random vertex of G^T
7:          sketch ← reachable nodes in G^T starting from v
8:          for each node u ∈ sketch do
9:              append sketchID to u's list of sketches
10:             count[u] ← count[u] + 1
11:     return hypergraph H
12: procedure GetSeeds(H)
13:     S ← ∅, σ(S) ← 0
14:     for i = 1, ..., k do
15:         seed v_i ← argmax_v{count[v]}
16:         S.insert(v_i)
17:         σ(S) ← σ(S) + count[v_i]
18:         remove the sketches containing v_i
19:     output S, σ(S)
```

ID, *nodes* stores the node IDs reached by this sketch. Arrays *sketches* and *nodes* are synchronized, so that knowing the index of a sketch, we can easily find the corresponding nodes, and *vice versa*. In the following figure we show an example of a *sketches* array (first row) and corresponding *nodes* array (second row):

0	0	0	0	1	1	2	2	3	4	4	5
0	1	2	3	2	3	1	3	2	2	3	0

In this example, sketch IDs and their corresponding node IDs are divided by double bars from other sketches: sketch 0 contains four nodes: 0, 1, 2, and 3; sketch 1 contains two nodes: 2 and 3; and so on. Note that sketches are listed in ascending order, and the corresponding nodes for each sketch are listed in ascending order as well. We use these features for speeding up the calculation of seeds. Testing FA on real-life graphs shows its better usage of space and faster performance than 2DL (subsection 4.1).

Algorithm 2 FA

```
1:  procedure BuildHypergraph(R)
2:      initialize sketches and nodes arrays to -1
3:      while H_weight < R do
4:          v ← random vertex of G^T
5:          sketch ← reachable nodes in G^T starting from v
6:          for each node u ∈ sketch do
7:              sketches[i] ← sketchID
8:              nodes[i] ← u
9:              count[u] ← count[u] + 1
10:     return hypergraph H = (sketches, nodes)
```

3.4 Compressed Flat Arrays (CS-FA)

Here we present a more efficient implementation, the CS-FA algorithm (Algorithm 3). The main difference between CS-FA and FA is the design of the *sketches* array: CS-FA stores the accumulated

Algorithm 3 CS-FA

```
1:  procedure BuildHypergraph(R)
2:      initialize nodes array to -1
3:      while H_weight < R do
4:          v ← random vertex of G^T
5:          sketch ← reachable nodes in G^T starting from v
6:          for each node u ∈ sketch do
7:              node_count ← node_count + 1
8:              add nodeID to array nodes
9:              count[u] ← count[u] + 1
10:         add node_count to array sketches
11:     return hypergraph H = (sketches, nodes)
```

count of nodes included in sketches, thus making the *sketches* array compressed. Now we do not need to store sketch id's explicitly. Sketch id's are the indexes in array *sketches*. The example below shows that sketch 0 includes three nodes, sketch 1 includes (5 - 3) = 2 nodes, and so on. The *nodes* array lists the corresponding nodes.

sketches:	3	5	6	8	10

nodes:	0	1	3	0	1	0	2	3	3	4

When we want to retrieve a sketch with id, say i, we need to find where its nodes start in the nodes array. This is given by the number stored in $sketches[i-1]$ or 0 if $i = 0$. Testing shows CS-FA's smaller footprint and better run time than 2DL's or FA's (subsection 4.1).

4 EXPERIMENTAL RESULTS

We tested our IM and IE solutions by extensive experiments on several real-world graphs. For brevity, we included in this paper only the most interesting and telling results for IM and their analysis. All the presented results are achieved on a consumer-grade laptop with 16G of main memory.

We implemented the algorithms in Java 8 taking advantage of parallel streams and lambda expressions, and used Webgraph [1] as a graph compression framework. Webgraph is actively maintained and fully documented (http://webgraph.di.unimi.it). Some other works that use Webgraph to scale algorithms to big graphs are [3, 10, 18–20, 23].

We compared our implementations with each other and with the DIM algorithm, implemented in C++ ([16]). We used the DIM code from
https://github.com/todo314/dynamic-influence-analysis.

Datasets. Due to space constraints, we only present results for three real world graphs. Results for other datasets were similar. The datasets are available from the Laboratory for Web Algorithmics (http://law.di.unimi.it/datasets.php).

Dataset	n	m
UK100K	100,000	3,050,615
CNR-2000	325,557	3,216,152
EU-2005	862,664	19,235,140

Table 1: Datasets ordered by m.

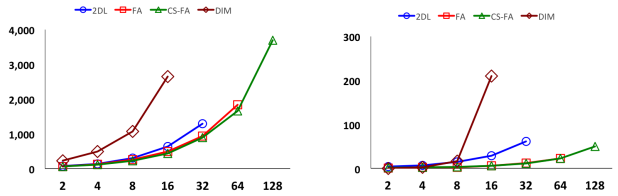
While the size of the networks we considered is in the medium range, since each node can be sampled many times (we use sampling with replacement), the count of edges touched by the algorithms is in the billions. For example, the smallest of presented datasets, UK100K, requires a hypergraph with a weight of at least 5.6 billion, in order to produce the IM solution for $\beta = 16$ and $k = 100$.

Equipment. The experiments were conducted on a laptop with processor 2.2 GHz Intel Core i7 (4-core), RAM 16GB 1600 MHz DDR3, running OS X Yosemite.

Parameters. The parameters we use in our testing are as follows. k is the number of seeds in the seed set, β is a coefficient in Borgs *et al.* formula for the hypergraph weight, and p is the probability of edge existence . The tests are conducted varying k and β for $p = 0.1$.

4.1 Comparison of arrays, 2D list, and DIM performance

Fig. 1 shows the total running time and the time used for seeds calculation by DIM *vs.* our implementation of 2DL *vs.* FA *vs.* CS-FA. The test shown was conducted on CNR-2000, for $k = 10$, $p = 0.1$, and $\epsilon = 0.1$, varying β in powers of 2, from 2 to 128.



(1) Total Time (sec) **(2) Seeds Time (sec)**

Figure 1: Processing time for cnr-2000; k=10, varying β.

The two-dimensional list implementations, DIM and 2DL, run slower and require more memory than array implementations. The reason for comparatively poor performance of 2D list implementations is the fragmentation of the main memory, when allocating space for each second-dimension list of sketch numbers for a node. This causes the memory manager to perform a lot of work trying to rearrange memory blocks. Improvements implemented in 2DL listed in subsection 3.2 allow for a better time performance on both the hypergraph computation and seed calculation, compared to DIM. For example, for $\beta = 16$, DIM took three times longer than 2DL to produce the result. The running times of FA and CS-FA are almost identical with each other. This is good for CS-FA; the compression we perform not only does not slow down CS-FA, but it makes CS-FA slightly faster due to better memory utilization. Both FA and CS-FA are faster than 2DL and DIM.

On both charts in Fig. 1, some data points are missing, because of the required memory being higher than what is available on the machine. 2DL and FA can handle runs with a β up to 32 and 64, respectively, while CS-FA can handle β equal to 128 due to its smaller memory footprint. That is, CS-FA scales the most, about 8 times more than DIM.

Fig. 2 shows the performance of 2DL, FA, and CS-FA when parameters k and β are growing, from the first chart, where all three implementations could run to completion, till the last one, where only the most efficient data structure (CS-FA) could produce one result, for the lowest β. The larger the β and k, the longer it takes for building the hypergraph and calculating the seeds, within one graph. This can be seen in the charts, while following a column from the top chart down. The larger the graph, the longer it takes for building the hypergraph and calculating the seeds. This can be seen in the charts, while following a row from the left chart to the right.

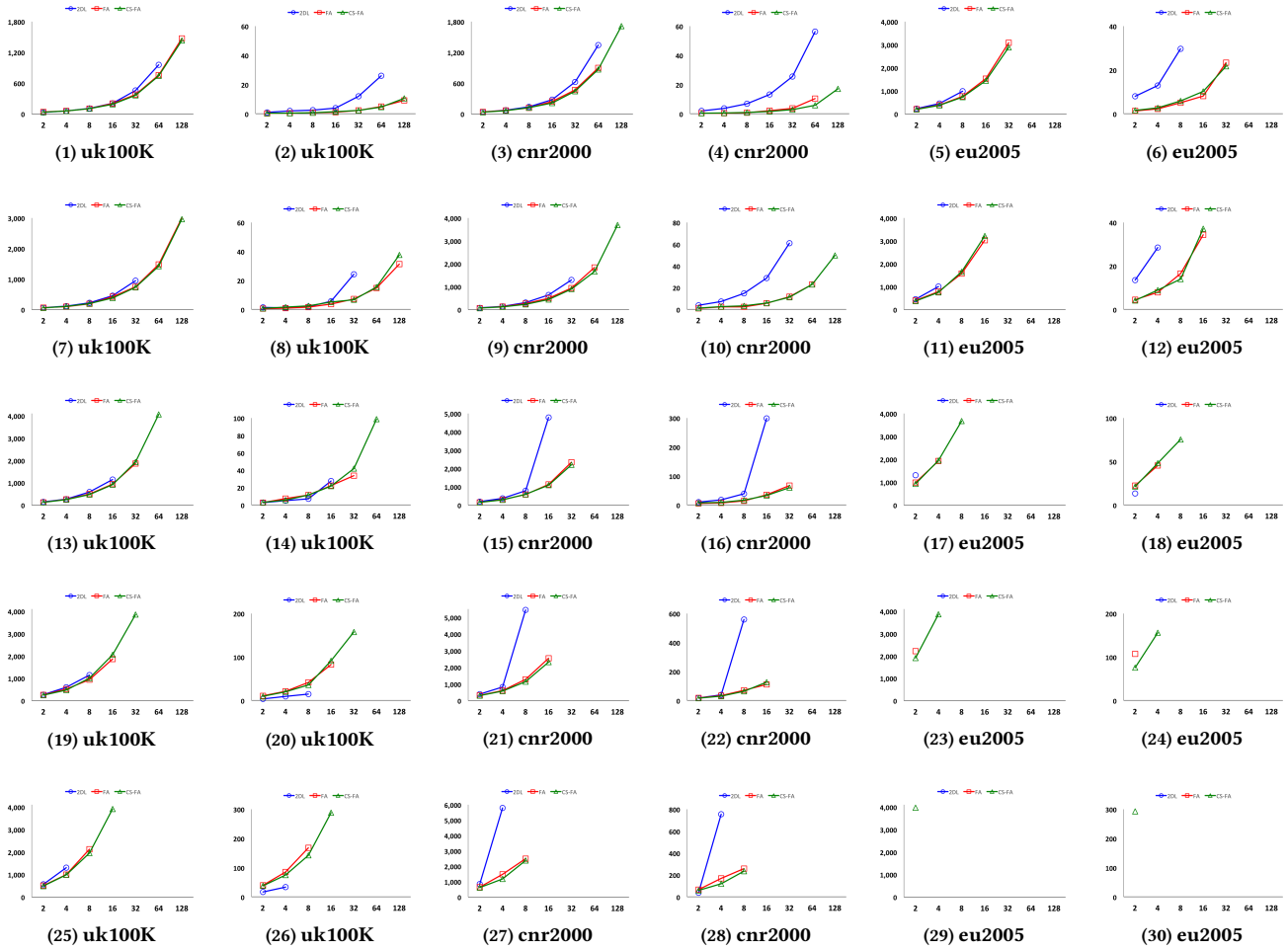


Figure 2: Total time (sec), and seeds time (sec). Per row, k = 5, 10, 25, 50, 100

The largest hypergraph, successfully created and processed on the laptop, touched almost 5.6 billion edges. This hypergraph was processed by CS-FA (subsection 3.4). It took CS-FA one hour six minutes, including five minutes for calculating 100 seeds. We do not know another algorithm that can process such a hypergraph on a comparable machine.

Finally, in both Fig. 1 and 2, we observe that the time for calculating seeds is only a small part of the total time, which influenced our decision to not show separately experiments for IE (due to space constraints).

5 CONCLUSIONS AND FUTURE RESEARCH

We presented several implementations for computing influence estimation and influence maximization on graphs with multimillion edges. Our algorithms use different data structures. We tested the performance of these data structures on larger graphs, and provided a comparative analysis of test results. We substantially reduce the running time and required memory, without affecting the theoretical guarantees, to the point that multimillion-edge graphs could be processed on a consumer-grade laptop. Future research will involve further compression and parallelism aiming at scaling the computation of influence to bigger networks. Also, we would like to extend our results to richer models of graphs, such as those where the edges are labeled by the type of the connection between users, e.g. family, colleague, classmate, etc (cf. [7, 12, 17]).

The source code for this paper can be found at:
https://github.com/dianapopova/InfluenceMax

REFERENCES

[1] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *WWW*, 2004.
[2] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier. Maximizing social influence in nearly optimal time. In *SODA*, pages 946–957, 2014.
[3] S. Chen, R. Wei, D. Popova, and A. Thomo. Efficient computation of importance based communities in web-scale networks using a single machine. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1553–1562. ACM, 2016.
[4] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *KDD*, pages 199–208, 2009.
[5] E. Cohen, D. Delling, T. Pajor, and R. F. Werneck. Sketch-based influence maximization and computation: Scaling up with guarantees. In *CIKM*, pages 629–638, 2014.
[6] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.
[7] G. Grahne and A. Thomo. Algebraic rewritings for optimizing regular path queries. *Theoretical Computer Science*, 296(3):453 – 471, 2003.
[8] K. Huang, S. Wang, G. S. Bevilacqua, X. Xiao, and L. V. S. Lakshmanan. Revisiting the stop-and-stare algorithms for influence maximization. *PVLDB*, 10:913–924, 2017.
[9] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.
[10] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.
[11] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *KDD*, pages 420–429, 2007.
[12] Z. Miao, D. Stefanescu, and A. Thomo. Grid-aware evaluation of regular path queries on spatial networks. In *Advanced Information Networking and*

Applications, 2007. AINA'07. 21st International Conference on, pages 158–165. IEEE, 2007.

[13] H. T. Nguyen, M. T. Thai, and T. N. Dinh. Stop-and-stare: Optimal sampling algorithms for viral marketing in billion-scale networks. In *SIGMOD*, pages 695–710, 2016.

[14] H. T. Nguyen, M. T. Thai, and T. N. Dinh. A billion-scale approximation algorithm for maximizing benefit in viral marketing. *IEEE/ACM Trans. Netw.*, 25(4):2419–2429, Aug. 2017.

[15] N. Ohsaka, T. Akiba, Y. Yoshida, and K.-i. Kawarabayashi. Fast and accurate influence maximization on large networks with pruned monte-carlo simulations. In *AAAI*, pages 138–144, 2014.

[16] N. Ohsaka, T. Akiba, Y. Yoshida, and K.-i. Kawarabayashi. Dynamic influence analysis in evolving networks. *PVLDB*, 9(12):1077–1088, 2016.

[17] M. Shoaran and A. Thomo. Fault-tolerant computation of distributed regular path queries. *Theoretical Computer Science*, 410(1):62 – 77, 2009.

[18] M. Shoaran and A. Thomo. Zero-knowledge-private counting of group triangles in social networks. *The Computer Journal*, 60(1):126–134, 2017.

[19] M. Simpson, V. Srinivasan, and A. Thomo. Clearing contamination in large networks. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1435–1448, June 2016.

[20] M. Simpson, V. Srinivasan, and A. Thomo. Efficient computation of feedback arc set at web-scale. *Proceedings of the VLDB Endowment*, 10(3):133–144, 2016.

[21] Y. Tang, Y. Shi, and X. Xiao. Influence maximization in near-linear time: A martingale approach. SIGMOD '15, New York, NY, USA.

[22] Y. Tang, X. Xiao, and Y. Shi. Influence maximization: Near-optimal time complexity meets practical efficiency. SIGMOD '14, New York, NY, USA.

[23] B. Tootoonchi, V. Srinivasan, and A. Thomo. Efficient implementation of anchored 2-core algorithm. In *Proceedings of ASONAM'17*, pages 1009–1016, 2017.