Discovering k-Trusses in Large-Scale Networks

Static Graph Challenge: Subgraph Isomorphism (k-truss)

Alessio Conte
National Institute of Informatics, Tokyo, Japan
conte@nii.ac.jp

Daniele De Sensi
Università di Pisa
Pisa, Italy
desensi@di.unipi.it

Roberto Grossi
Università di Pisa
Pisa, Italy
grossi@di.unipi.it

Andrea Marino
Università di Pisa
Pisa, Italy
marino@di.unipi.it

Luca Versari
Università di Pisa
Pisa, Italy
luca.versari@di.unipi.it

***Abstract*—A k-truss is a subgraph where every edge belongs to at least k-2 triangles in the subgraph. The truss decomposition assigns each edge the maximum k for which the edge belongs to a k-truss, and the trussness of a graph is the maximum among its edges. Discovery algorithms for k-trusses and truss decomposition provide useful insight for graph analytics (such as community detection). Even though they take polynomial time, on massive networks they suffer from handling a potentially cubic number of wedges: algorithms either need a long time to recompute triangles several times, have high memory usage, or rely on the large number of cores on graphic units. In this paper we describe EXTRUSS, a highly optimized algorithm for truss decomposition which outperforms existing algorithms. We then introduce a faster algorithm, HYBTRUSS, which finds the trussness of a graph using less time and space than EXTRUSS. Our algorithms take the best of existing approaches having good performance, low memory usage, and no need for sophisticated hardware systems. We compare our algorithms with the state-of-the-art on a set of real-world and synthetic networks. EXTRUSS processes graphs with over a billion edges, which seems difficult for the competitors, and our HYBTRUSS is the first algorithm able to find the trussness of a graph with over 25 billion edges.**

***Index Terms*—k-trusses, truss decomposition, graph algorithms, HPEC 2018 graph challenge**

I. Introduction

The notion of k-trusses has spread in network analysis and graph mining over the years, and is gaining momentum for other purposes other than security [3]. Given an integer $k \geq 2$, the *k-truss* of an undirected graph G is the subgraph H of G in which every edge belongs to at least $k-2$ triangles. Note that k-trusses can be computed in polynomial time [3]. The *k-truss decomposition* of G corresponds to assigning each edge its *trussness* value, i.e., the highest k for which the edge belongs to a k-truss. Given the truss decomposition, it becomes easy to extract the k-truss for any k, and thus for the largest k, which defines the *graph trussness*. Furthermore, it can be proved that the k-truss of a graph can be obtained by recursively deleting edges which participate in less than $k-2$ triangles [3] [1] This simple strategy, sometimes referred as *peeling*, is surprisingly effective, and is at the heart of the best performing algorithm for computing k-trusses and k-truss decomposition [7].

Work partially supported by MIUR, Italy, and by JST CREST, Grant Number JPMJCR1401, Japan.

[1] This recursive deletion reminds how k-cores are obtained by recursively removing nodes with less than k neighbors.

In this paper we present fast algorithms to find the truss decomposition on large real-world networks. Their time complexity is $O(m\,\alpha)$, where m is the number of edges and $\alpha = O(\sqrt{m})$ is the arboricity in the input graph G. The goal is proposing simple, efficient algorithms, which can compete or even outperform state of the art algorithms while keeping a small memory footprint (optimized for a single machine).

In order to get a scalable solution for truss decomposition on massive graphs, a challenge is that high degree nodes create many wedges. Several solution perform node reordering such as forward [10] or reverse CuthillMcKee [8], where the nodes are numbered according to a particular breadth-first traversal where neighboring nodes are visited in order from lowest to highest node order. We obtain the same effect *without* reordering, using the endpoint of min-degree of each edges, but without any bucketing [1]. We circumvent the drawback of dense portions of triangles by indirectly removing several edges using parallel and batch operations. We update the number of triangles without needing to recompute them at every iteration [4].

Other features include small memory requirement during its execution, and the ability to quickly keep the adjacency lists sorted and compacted, which dramatically affects the cache complexity and allows us to use SIMD operations for list intersection and compaction. In several solutions, most of the performance of the code lies in the implementation of the intersection operation between adjacency lists.

Finally we introduce a very fast approximate algorithm, APXTRUSS, to estimate the graph trussness. Combining the strengths of EXTRUSS and APXTRUSS, we obtain a new algorithm, HYBTRUSS, that computes the exact edge trussness when sufficiently large, and that exhibits all the best traits of known algorithms: good performance, low memory usage, and no need for sophisticated hardware systems.

We compared EXTRUSS to the best known approaches: [12] and [6]. The former is a finalist, and the latter won a student innovation award in the 2017 GraphChallenge [7]. We have show in Section VI that our algorithm outperforms the competitors and scales up to more than 1 billion edges, while competitors do not always terminate. Finally, show that HYBTRUSS further improves the performance while still finding the exact result in practice. This allows us to find, in less than 12 hours, the trussness of a web graph with about 1 billion nodes and over

Algorithm 1: Structure of algorithm EXTRUSS

Input : Graph $G = (V, E)$
Output: Decomposition $truss[]$ and trussness t_G of G

```
1  foreach edge e ∈ E do compute support[e]
2  t ← 0
3  while E ≠ ∅ do
4      t ← max(t, support_min(G))
5      Q ← {e ∈ E : support[e] ≤ t}      // edges to delete
6      parallel foreach e ∈ Q do
7          truss[e] ← t + 2
8          E ← E \ {e}                   // remove e from G
9          Update support[]
10 Return truss[], t_G = t + 2
```

25 billions of edges in less than 12 hours. As far as we known this the largest network in known literature whose trussness has been computed.

II. Exact Algorithm

We introduce our first algorithm, EXTRUSS, which finds the truss decomposition of a given graph $G = (V, E)$. The algorithm is based on the well known *peeling* strategy. We keep track in t of the maximum support found so far for a removed edge, and mark the trussness of removed edges accordingly. Its pseudo-code is shown in Algorithm 1. Using the queue Q to collect all edges with support below t, we can parallelize both their removal and the consequent update of the support, which is a costly operation. In turn, removing many edges at once may cause many other edge to fall below the required support, and these can be quickly added to Q and removed in the next step.

Data Structures: As we deal with massive graphs, the data structures used by EXTRUSS must at the same time be fast to access and as small as possible in order to fit in memory (in practice, using more than linear space does not allow to scale up to very large graphs). Each node $v \in V$ has a unique identifier $id(v) \in [0, \ldots, n-1]$, and each edge $e \in E$ has a unique identifier $id(e) \in [0, \ldots, m-1]$, where $n = |V|$ and $m = |E|$. When referring to a node or edge x, for simplicity also refer to its identifier as x. This is a summary:

1) G: an array containing all adjacency lists, consecutively (length $= 2m$)
2) Map each position (i.e., index) in G to
   - Id of the corresponding edge
   - Edge has been removed or not (boolean flag)
3) Map each node id to position in G of
   - Start and end of its adjacency list
   - First neighbor in its list larger than itself
4) Map each edge id to ids of its extremes and support

This information takes 40–60 bytes per edge.[2]

Operations: The algorithm is designed to only use operations which can be highly optimized in practice. For example, it never requires random testing of adjacency between any two nodes, which is likely to cause a cache miss.

- EXTRUSS does not use node ordering, a costly step (for large graphs) found in other algorithms (e.g. [8], [10]).
- When removing edge $\{u, v\}$, we update the supports by decreasing by 1 that of edges from u or v to a node in $N(u) \cap N(v)$, which can be implemented quickly as shown later. Updates take constant time each thanks to our data structures.
- If the edges incident to any one node are less than $t+1$, we can remove them. We track degrees using counters to quickly spot the nodes and add their edges to Q.
- If the degree of both extremes of a removed edge is below $t+1$, we skip updating the support of incident edges, as they will all surely be removed.

We adopted the following strategy for intersecting two lists A and B (assuming $|A| \geq |B|$):

- If $|B| \leq |A| \leq 4|B|$, i.e., the lists are balanced, we perform fast intersection with SIMD operations [5]: e.g., we divide them in chunks of size 4, comparing one against rotations of the other with SSE/AVX. Thanks to the pipeline, these 4 have the cost of a single instruction.
- If $|A| > 4|B|$, we use classic ordered list intersection with two indices i on A and j on B. j is increased by one each time (as usual), but i is increased by logarithmic search: we first try to increase i by 4, then double until we go over $B[j]$. Finally we use binary search between the last two values of i, at a total cost of $O(|B| \log(|A|/|B|))$.

In practice, the intersection takes at most one scan of each list, which is cache-friendly and easy to prefetch, and the cost is proportional to the size of the *smaller* list. This is particularly relevant in real-world graphs, where most nodes have few neighbors, but some may have a huge amount of them (see, e.g., scale-free/power-law graphs).

III. Approximated algorithm apxtruss

We wonder how to further speed up the good performance of EXTRUSS, especially for large graphs, which need relatively expensive hardware.[3] We aim to push the bar further, by providing an algorithm which processes graphs larger than the available memory without compromising the performance.

In this section, we discuss APXTRUSS: an *approximate* algorithm which returns an upper bound $t_{\mathcal{U}}$ and lower bound $t_{\mathcal{L}}$ for the graph trussness with $t_{\mathcal{U}} \leq ct_{\mathcal{L}}$ for some constant $c \geq 1$. We apply APXTRUSS in an unusual fashion, as a building block for an exact algorithm, HYBTRUSS, which

[2]Assuming at most 2^{32} nodes and 2^{64} edges, i.e., using 32 bit unsigned integers for node ids, a reasonable assumption even for very large graphs (edges correspond to 2 nodes, and an edge's support is $\leq n-2$). Otherwise, switching to 64 bit integers doubles memory usage in the worst case. The variability depends on the $\frac{n}{m}$ ratio.

[3]And this is not likely to change, as the size of observable real-world graphs seem to increase rapidly.

exploits APXTRUSS and EXTRUSS to return the exact trussness (and the maximum t-truss) using less time and space than EXTRUSS, under suitable assumptions.

A key feature of the algorithm is a time/quality trade-off. Furthermore, its structure is suitable for an efficient external memory implementation, which allows us to overcome the limit of main memory size.

Data Structures: The key difference with respect to EXTRUSS is that APXTRUSS recomputes supports from scratch at each iteration, thus does not need to optimize data structures for updating the supports. As such it only uses the following structures:

1) G as concatenation of the nodes' ordered adjacency lists
2) Map each node id to the start of its adjacency list in G
3) Map each edge id to its support

We can bound space to around 12–20 bytes per edge. Note that this structure is resident in the secondary memory, and mapped to the main memory using `mmap`, so that external memory handling is left to the OS.

Operations: APXTRUSS does not update supports but recomputes them from scratch, which is faster when removing many edges at once. This is done by intersecting the neighborhood of the endpoints of each edge, as explained for EXTRUSS. The process is also sped up by considering, for edge $\{u, v\}$, only neighbors *larger* than both u and v: this process still finds all triangles and is significantly faster. Moreover, we consider all edges incident to the same node at the same time to minimize cache misses. Finally, rather than flagging the removed edges, APXTRUSS removes them from G, making the data structures smaller, and thus later iterations faster.

IV. HYBTRUSS: TAKING THE BEST OF BOTH WORLDS

Finally, we present HYBTRUSS: an algorithm which finds the trussness faster and with less space than EXTRUSS, and thus is able to process much larger networks.

We run APXTRUSS, which starts removing edges from G, then switch to HYBTRUSS as soon as the remaining edges fall below a given threshold $\mathcal{E}$. If initially $E \leq \mathcal{E}$, we can immediately run EXTRUSS and obtain the correct result. On the other hand, if EXTRUSS is never executed, we obtain the result of APXTRUSS (a c-approximation). Otherwise, we obtain $t_{\mathcal{L}}$ and $t_{\mathcal{U}}$ returned by the partial execution of APXTRUSS, and $\tilde{t}_G$ returned by EXTRUSS. We can observe the following:

1) No edge with trussness $t > t_{\mathcal{U}}$ is removed by APXTRUSS (since APXTRUSS never removes edges with support more than $t_{\mathcal{U}}$), so if $\tilde{t}_G \geq t_{\mathcal{U}}$ then $\tilde{t}_G$ is exactly t_G.
2) Otherwise, $t_{\mathcal{U}} > t_G$, and since $t_{\mathcal{L}} \leq t_G$, $t_{\mathcal{L}}$ and $t_{\mathcal{U}}$ still give an approximation (where the lower bound can be improved to $\max(t_{\mathcal{L}}, \tilde{t}_G)$).

More in general, the decomposition $truss[]$ given by EXTRUSS is correct for all values of trussness $\geq t_{\mathcal{U}}$, so when we find the correct value of t_G we also find the max k-truss. In practice, in Section VI HYBTRUSS is faster than EXTRUSS and always finds the exact trussness, even when $\mathcal{E}$ is significantly smaller than the initial number of edges. In Section VI-C we use HYBTRUSS to find the trussness of a graph with over 25 billion edges.

V. EXPERIMENTS

This describes our experimental settings and choices. We compare our exact algorithm with the fastest known exact algorithms for k-truss decomposition, and the python baseline provided for the challenge. We also compare our approximated and hybrid approaches with the exact method. The performance evaluation uses the metrics used in the past edition of the GRAPHCHALLENGE (e.g., see [9]).

a) Competitors:

- EXTRUSS: Our exact algorithm (see Algorihtm 1).
- MSP: the parallel algorithm in a shared-memory setting proposed in [12]. It has been one of the finalists of the 2017 GRAPHCHALLENGE [7]. The code has been downloaded from https://github.com/KarypisLab/K-Truss.
- PKT: the approach proposed in [6]. It won a student innovation award in the 2017 GRAPHCHALLENGE [7]. The code has been downloaded from https://github.com/humayunk1/PKT.
- BASELINE: this is the python sequential baseline provided by the organisers of the GRAPHCHALLENGE.

PKT and MSP are the state-of-the-art algorithms for k-trusses. Since there is no direct comparison between PKT and MSP as this comparison is left open in [12], we consider both of them as our direct competitors.

We will also analyze the performance of our approximated algorithms.

- APXTRUSS: Our approximated algorithm for k-trusses and trussness.
- HYBTRUSS: Our combination of EXTRUSS and APXTRUSS, which approximates the k-trusses but computes exactly the trussness of the graph. For less edges than this threshold the algorithm is equivalent to EXTRUSS.

For all the algorithms (except the baseline which is sequential), we used the maximum degree of parallelism allowed by the single-machine architecture (24 threads).[4]

b) Dataset: Our dataset is taken from LAW (law.di.unimi.it/) and from [7]. Some of them are also distributed by SNAP (snap.stanford.edu/). Our sample of networks cover collaboration, autonomous systems, social, collaboration, random, and web networks. We report in Table I the number of nodes, edges and the maximum k for each graph, which has been made undirected.

c) Software and architecture: The evaluations have been performed on a 12-core (24-thread) machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz, with 128GB of shared memory. The operating system is a Ubuntu 14.04.2 LTS, with Linux kernel version 3.16.0-30.

Our algorithms have been implemented in `C++11`, compiled with `gcc-5.5.0`, using the `-O3` optimization flag. The source code of our algorithm is available upon request from the

[4]Varying the number of used threads, the scalability of all the approaches is quite similar, with PKT scaling slightly better than EXTRUSS and MSP.

NETWORK	TYPE	NODES	EDGES	k_{max}
soc-Epinions1 (epi)	soc	75 888	405 739	33
as-Skitter (skit)	as	1 696 415	11 095 298	68
eu-2005 (eu05)	web	862 664	16 138 468	387
imdb (imdb)	coll	913 201	37 588 613	1 298
LiveJournal1 (jour)	soc	4 847 571	42 851 236	362
hollywood-2009 (ho09)	coll	1 139 905	56 375 711	2 209
enwiki-2013 (wiki)	soc	4 206 785	91 939 728	53
hollywood-2011 (ho11)	coll	2 180 759	114 492 816	1 298
graph500-ef16 (g500)	rand	17 043 781	523 467 448	996
arabic-2005 (arab)	web	22 744 080	553 903 073	3 248
it-2004 (it04)	web	41 291 594	1 027 474 947	3 222
twitter-2010 (twit)	soc	41 652 230	1 202 513 046	1 998
gsh-2015-host (gshh)	web	68 660 142	1 502 666 069	9 923
com-Friendster (frie)	soc	65 608 366	1 806 067 135	129
gsh-2015	web	988 490 691	25 690 705 118	5 204

TABLE I
GRAPHS CONSIDERED IN OUR EXPERIMENTS. EACH GRAPH HAS BEEN MADE UNDIRECTED. THE LAST GRAPH HAS BEEN CONSIDERED JUST FOR HYBTRUSS (SEE SECTION VI-C).

PC members. OpenMP has been used to implement the parallel version of our code, and we used AVX2 for instruction-level data parallelism.

d) Measures: The measures listed below are those suggested by GRAPHCHALLENGE organizers [9][5]. All our measurements for all the algorithms exclude the reading phase of the network file.

- EXECUTION TIME: The time (in seconds) needed to conclude the k-truss decomposition. The maximum time allowed for each competitor in each experiment has been set to 2 hours.
- RATE: The ratio between number of edges and the execution time.
- MEMORY: The *peak memory*, i.e. the maximum memory (in GigaBytes) used by a competitor during its execution.
- POWER CONSUMPTION: The average power consumption (in Watts), i.e. the energy consumption divided by the execution time. For measuring it we used the *Mammut*[6] [11] library which, on the architecture used for our experiments, relies on Intel's RAPL counters.
- RATE PER POWER. This measure is computed as the ratio between the rate and the average power consumption and can be interpreted as a measure of the efficiency (in terms of power consumption).

VI. RESULTS

This section is divided in three parts, Sections VI-A–VI-C. The first deals with the exact approaches and evaluates the different metrics presented in Section V for the graphs in our dataset. The second deals with approximated approaches, and the last one shows the result of one of our approaches with the largest graph in our dataset. As we will see, EXTRUSS outperforms its competitors, and further HYBTRUSS outperforms EXTRUSS for solving the simpler problem of finding the trussness.

[5]These are also listed at https://graphchallenge.mit.edu/sites/default/files/documents/SubGraphChallenge-2017-02-09.pdf

[6]https://github.com/DanieleDeSensi/mammut

A. Exact Algorithms

In this section we compare EXTRUSS with respect to the other exact methods, namely MSP, PKT, and BASELINE. In the first part we focus on the comparison between EXTRUSS and BASELINE, as this comparison involves just the smallest network, due to the limitations of BASELINE. In the second part we focus on the comparison with MSP and PKT. We remark that the results we obtained are consistent with those computed by MSP [12] and PKT [6]. In all the experiments, we write OOM to mean out-of-memory, when an algorithm exceeded the available memory of our machine (128GB), or OOT when the algorithm did not finish within the 2 hours time limit.

1) Comparing EXTRUSS *with the* BASELINE*:* We discuss the results of a comparison between BASELINE and EXTRUSS for the smallest graph of our dataset epi.

- Looking at the rate, EXTRUSS is 1577 times faster than BASELINE.
- The memory usage of EXTRUSS is 894 times lower than BASELINE.
- The rate per power of EXTRUSS is 906 times higher than the one of BASELINE.

2) Comparing EXTRUSS *with the state of the art:* In the following, we compare EXTRUSS with MSP and PKT observing the behaviour of the measures introduced in Section V.

a) Execution Time and Rate: In Figure 1 we show the execution time of the competitors. The missing points for MSP and PKT for the larger graphs are due to their OOT or OOM. Our algorithm EXTRUSS is almost always the fastest one and the improvement is much more evident in the larger graphs. On the graphs having more than 500 millions and less than 1.1 billions of edges, we are from four to seven times faster than PKT. For graphs with more than this number of edges, we had similar time performance in the case of friend, but for the other graphs PKT ran OOT. For some graphs MSP is faster than PKT, but it ran OOM for all the graphs having more edges than the ones of graph500, i.e. roughly 500 millions of edges.

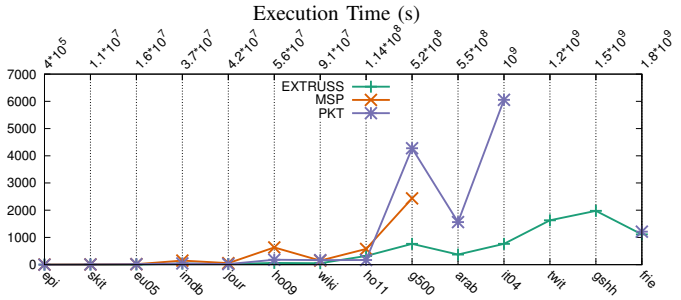

Fig. 1. Comparison of the execution time (y-axis) among EXTRUSS, MSP and PKT for our graphs (x-axis) with their corresponding number of edges (upper x-axis). Line interruptions corresponds to OOM experiments. Lower is better.

The improvement of EXTRUSS in the case of smaller graphs is more evident looking at Figure 2, where we show the rate, i.e. ratio between number of edges and time. EXTRUSS

(green line) performs almost always better than the others. The improvement with respect to MSP seems to be quite constant, while the improvement on PKT seems to more variable and more graph dependant.

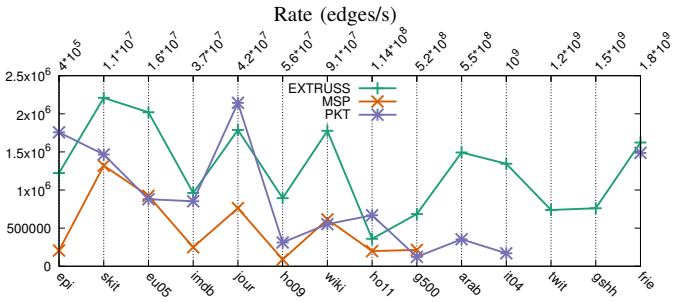

Fig. 2. Comparison of the rate (y-axis) among EXTRUSS, MSP and PKT for our graphs (x-axis) with their corresponding number of edges (upper x-axis). Line interruptions corresponds to OOM or OOT experiments. Higher is better.

b) Memory Usage: Concerning the memory usage, in Figure 3 we show our results for the competitors. MSP seems to use much more memory than the others and this is the reason why it ran OOM on the larger graphs. On the other hand, the memory performance for PKT seems to be slightly better than EXTRUSS, and this is at the price of a higher execution time since, for instance, on `it` and `gshh` it ran OOT (see Figure 1).

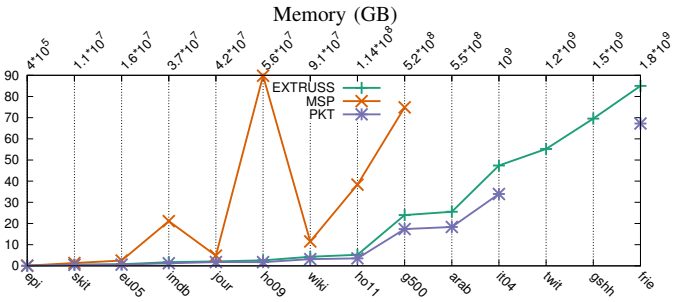

Fig. 3. Comparison of the memory usage (y-axis) among EXTRUSS, MSP and PKT for our graphs (x-axis) with their corresponding number of edges (upper x-axis). Line interruptions corresponds to OOM or OOT experiments. Lower is better.

c) Power Consumption and Rate per Power: The average power consumption of EXTRUSS and PKT is around 90 Watts and 100 Watts respectively, while MSP consumes around 45 Watts in average. We further investigated the execution profile of MSP and we found out that it is characterized by a long preprocessing sequential phases, which lowers the average power consumption but also increases the execution time.

Indeed, as shown in Figure 4, MSP is not as power efficient as EXTRUSS since it performs less work per unit of power. The rate per watts of EXTRUSS almost always dominate the other approaches, and for large graphs it is up to 9.5 times more power efficient than PKT and up to 6 times more power efficient than MSP.

B. Approximated Algorithms

In this section, we analyze the behaviour of our algorithms APXTRUSS and HYBTRUSS, where the former approximates

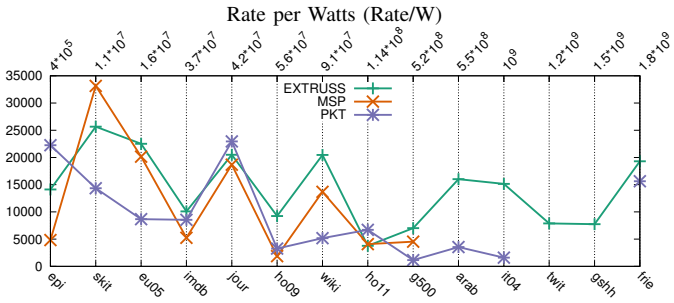

Fig. 4. Comparison of the rate per watt (y-axis) among EXTRUSS, MSP and PKT for our graphs (x-axis) with their corresponding number of edges (upper x-axis). Line interruptions corresponds to OOM or OOT experiments. Higher is better.

both trussness and k-trusses, while the latter computes exactly the trussness and approximates the k-trusses. We analyze their behaviour with respect to the exact algorithm EXTRUSS.

a) Execution Time and Memory Usage: We remark that for graphs having less than $100M$ edges the algorithms HYBTRUSS and EXTRUSS are the same. Above this threshold, in terms of execution time, the behaviour of APXTRUSS and HYBTRUSS is similar on big graphs and is strictly better than the one of EXTRUSS. In the great majority of the cases, the time needed by the APXTRUSS and HYBTRUSS is less than half the time needed by EXTRUSS. This behaviour is shown in Figure 5, which shows the rates: we can see that APXTRUSS often runs at least 3 times faster than EXTRUSS, and that APXTRUSS and HYBTRUSS behave similarly on big graphs.

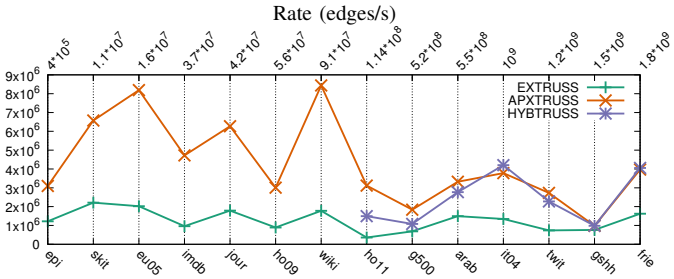

Fig. 5. Comparison of the rate (y-axis) among EXTRUSS, APXTRUSS and HYBTRUSS for our graphs (x-axis) with their corresponding number of edges (upper x-axis). For graphs having less than $\mathcal{E} = 100M$ edges the algorithms HYBTRUSS is exactly EXTRUSS. Higher is better.

In Figure 6, we report the memory usage, showing the great improvement of APXTRUSS and HYBTRUSS with respect to EXTRUSS. In the case of larger graphs, HYBTRUSS seems to use always less than one fifth of the memory used by EXTRUSS. Moreover, the improvement seems to increase when the number of edges increase, which will allow us to process graphs up to 25 billions edges, as we will show later. HYBTRUSS slightly improve also over APXTRUSS.

b) Power Consumption: Eventually, we have seen that the power consumption of EXTRUSS, APXTRUSS and HYBTRUSS is similar, while the rate per power of APXTRUSS and HYBTRUSS is on average twice that of EXTRUSS.

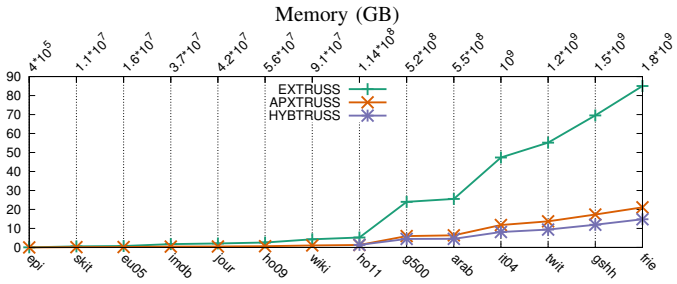

Fig. 6. Comparison of the memory usage (y-axis) among EXTRUSS, APXTRUSS and HYBTRUSS for our graphs (x-axis) with their corresponding number of edges (upper x-axis). Lower is better.

c) Rate varying approximation factor: For the sake of completeness, in the following we report the rate of APXTRUSS when varying the guaranteed approximation factor c. In particular, in Figure 7 we show for $c \in [2, 3.1]$ that clearly the rate of APXTRUSS increases when c grows. A very similar behaviour has been clearly observed also for HYBTRUSS.

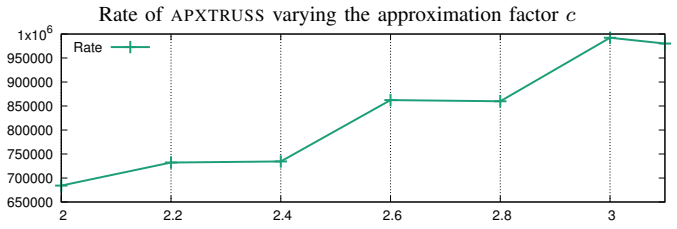

Fig. 7. Rate (y-axis) of APXTRUSS for `gshh` varying c (x-axis).

C. Experiments on a Huge Graph

As a final experiment, we have considered a large crawl of the web performed by BUbiNG [2] in 2015 starting from europa.eu, namely `gsh-2015`. We ran HYBTRUSS on this graph. We report in the following the results of our experiment.

	`gsh-2015`
NODES	988 490 691
EDGES	25 690 705 118
TIME (S)	42 890.44
RATE (EDGES/S)	598 984.43
POWER (W)	58.1328
RATE PER POWER (EDGES/S/W)	10 303.72
MEMORY (GB)	123.17

The trusness of `gsh-2015`, which is computed exactly by HYBTRUSS, is 5 204. Moreover, the trussness of edges whose value is in the interval $[523, 5\,204]$ is guaranteed to be exact.

VII. CONCLUSIONS

In this paper we presented two algorithms: EXTRUSS for computing the truss decomposition of a graph and HYBTRUSS for computing the trussness and the maximum k-truss.

We can summarize our results as follows.

- Our exact algorithm EXTRUSS improves memory, time, and power per rate with respect to the baseline. EXTRUSS processes graphs with more than a billion of edges, while competitors do not always terminate.
- HYBTRUSS pushes the bar further, reducing the memory usage by a factor of ten on the biggest graphs, for finding the trussness.
- As a result, we are able to perform the biggest known trussness computation, finding the exact trussness of a graph with over 25 billion edges.

Acknowledgements

We are grateful to Antonio Cisternino and Maurizio Davini for having provided facilities from the IT center of the University of Pisa (http://www.itc.unipi.it).

REFERENCES

[1] Jonathan W. Berry, Luke A. Fostvedt, Daniel J. Nordman, Cynthia A. Phillips, C. Seshadhri, and Alyson G. Wilson. Why do simple algorithms for triangle enumeration work in the real world? *Internet Mathematics*, 11(6):555–571, 2015.

[2] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. Bubing: massive crawling for the masses. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, pages 227–228, 2014.

[3] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report*, 16, 2008.

[4] Oded Green, James Fox, Euna Kim, Federico Busato, Nicola Bombieri, Kartik Lakhotia, Shijie Zhou, Shreyas Singapura, Hanqing Zeng, Rajgopal Kannan, et al. Quickly finding a truss in a haystack. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[5] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, 2014.

[6] Humayun Kabir and Kamesh Madduri. Parallel k-truss decomposition on multicore systems. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[7] MIT/Amazon/IEEE. GraphChallenge.org: Raising the bar on graph analytic performance. https://graphchallenge.mit.edu/, 2017. [Online; accessed 22/05/2018].

[8] Shahir Mowlaei. Triangle counting via vectorized set intersection. In *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*, pages 1–5, 2017.

[9] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, et al. Graphchallenge. org: Raising the bar on graph analytic performance. *arXiv preprint arXiv:1805.09675*, 2018.

[10] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms, 4th InternationalWorkshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005, Proceedings*, pages 606–609, 2005.

[11] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. Mammut: High-level management of system knobs and sensors. *SoftwareX*, 6:150 – 154, 2017.

[12] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis. Truss decomposition on shared-memory parallel systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2017.