UNDER REVIEW

RESCIENCE C

Replication / Algorithmics

[Re] Speedup Graph Processing by Graph Ordering

Fabrice Lécuyer[1], Maximilien Danisch[1], Lionel Tabourier[1]
[1]Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Edited by
(Editor)

Reviewed by
(Reviewer 1)
(Reviewer 2)

Received
–

Published
–

DOI
–

Abstract Cache systems keep data close to the processor to access it faster than main memory would. Graph algorithms benefit from this when a cache line contains highly related nodes. Hao Wei et al. propose to reorder the nodes of a graph to optimise the proximity of nodes on a cache line. Their contribution, Gorder, creates such an ordering with a greedy procedure. In this replication, we implement ten different orderings and measure the execution time of nine standard graph algorithms on nine real-world datasets. We monitor cache performances to show that runtime variations are caused by cache management. We confirm that Gorder leads to the fastest execution in most cases due to cache-miss reductions. Our results show that simpler procedures are yet almost as efficient and much quicker to compute. This replication validates the initial results but highlights that generating a complex ordering like Gorder is time-consuming.

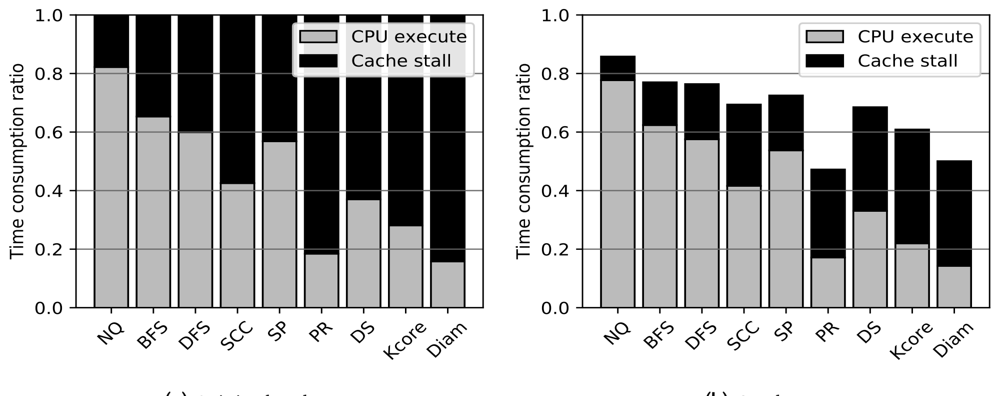
A replication of [1] by Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin, in Proceedings of SIGMOD 2016.

1 Introduction

In graph algorithmics, various procedures use the same few atomic operations. For instance, accessing the neighbours of a given node is key to a wide range of problems such as computing shortest paths, finding connected components, detecting communities etc. Making this type of elementary operation faster would improve such algorithms without having to modify their implementation [1]. Cache optimisation can have that effect: if two variables are often accessed together by algorithms, they should be stored side-by-side in memory so that they are copied together on a cache line. In a graph, it means reordering the nodes so that neighbours have close-enough indices. A cache-miss happens when data is not available in cache. The processor then has to fetch it in main memory, which is up to twenty times slower, depending on the machine architecture. As this cache stall is known to represent a significant share of the computation time [2, 3], reducing it can lead to important speedups.
This work replicates [1] by Hao Wei *et al.* which introduces Gorder, a new procedure to order nodes in a graph, and compares it to other standard orderings using typical algorithms and datasets as benchmarks. Because of the variety of graph algorithms, it is impossible to find an ideal ordering, which makes it interesting to propose and compare different strategies. The authors of the original paper claim an improvement of 10 to 50% in runtime, due to lower cache-miss rate.
We were able to replicate most of the experiments and confirm that ordering nodes according to Gorder makes the implementations 10 to 50% faster than without ordering. Section 2 presents the algorithms, orderings and datasets as well as issues faced during the replication. Our results are presented in Section 3 and are compared to the original results. Finally, Section 4 discusses the relevance of such an ordering compared to simpler ordering methods that offer satisfactory performances.

2 Method

The original study [1] was motivated by the observation that cache stall can take up to 70% of the whole computation time, which is supported by the observations reported



(a) Original order (b) Gorder

Figure 1. CPU execution and cache stall. Original order and Gorder are compared for all algorithms on *sdarc* dataset. Grey bars are time spent on CPU operations, black bars represent time spent waiting for data retrieval. Figure (a) shows the normalised runtimes with the original order, figure (b) shows the runtimes when the network has been reordered following the Gorder procedure. While both need about the same CPU time, the latter is significantly faster due to cache stall reduction. Compare to Figure 1 in [1].

in Figure 1. This issue has been addressed for specific algorithms such as breadth-first search [4]. In [1], the authors use a more general method: they reorganise the data to draw more benefit from the cache system, regardless of the algorithm or of the exact hardware specifications.
Gorder clusters nodes that are likely to be accessed simultaneously by any graph algorithm. More precisely, let us consider a graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. The proximity of two nodes u and v is measured by a score $S(u, v)$ which increases if they are neighbours and if they share many common in-neighbours. The total score F is the sum of $S(u, v)$ for all nodes u and v that have close indices. Window size w is the parameter that defines this closeness. Gorder creates an arrangement π of the indices to maximise F. We note π_u the index of a node u in such an arrangement. We give a more formal definition in Section 2.3.
The authors prove that finding the optimal ordering π is a NP-hard problem and propose a heuristic method with a theoretical approximation bound. They also present practical optimisations to reduce its time complexity. Finally, they run extensive experiments to compare their order to other standard orders.
Although the theoretical results of the original paper are important to explain the efficiency of Gorder, we only focus here on the algorithms and experiments. They provide an extensive analysis by comparing the runtimes for nine typical algorithms on eight large datasets with nine possible orderings. The current section describes them all and presents the replication issues that they imply. It also details the data structures that we used in this project. All the codes and instructions for this purpose can be found in our repository[1].

2.1 Algorithms

The original paper selects typical graph algorithms to test the different orderings. As implementation details are not fully documented and as its authors were not able to provide answers to some of our questions on this topic, we list below the details of our implementations.

Neighbour query (NQ) – Listing the neighbours of a given node is a standard elementary operation in graph algorithmics. As defined in [1], this operation must *access the out-neighbours* of each node. To ensure that neighbours are put in cache, thus benefiting

[1] https://github.com/lecfab/rescience-gorder

from a wise node ordering, an arbitrary operation is made over the set of neighbours. We compute for each node u the sum of degrees of its neighbours: $q_u = \sum_{v \in N_u} d_v$.

Breadth and Depth-first search (BFS, DFS) – BFS and DFS are standard graph traversal algorithms [5]. We adapted them to the data structure detailed in Section 2.2. Note that neighbours are selected in lexicographic order.

Strongly connected components (SCC) – To cluster nodes of the graph that can be accessed from one another, we use Tarjan's algorithm [6], which is a based on DFS.

Shortest paths (SP) – As in the original paper, we use Bellman-Ford algorithm [5] to compute the minimum distance from a source node to any other node. The time complexity after simple optimisations is in $O(\Delta m)$ where Δ is the diameter of the graph and m is the number of edges. As real-world networks are known to have relatively small diameters ($\Delta \ll n$), this algorithm works on massive datasets (see section 2.2). Note that for unweighted graphs, shortest paths can be computed in linear time and space using a BFS, but we keep the algorithm suggested in [1] for comparison purposes.

Page rank (PR) – This is the algorithm presented in [7] to rank webpages. It gives a score to each node according to its importance in the network structure. The original paper hints at an approximation based on the power iteration method with 100 iterations. We implement it with a damping factor set to $\alpha = 0.85$ which is a usual configuration.

Dominating set (DS) – A dominating set is a subset of nodes such that every node of the graph either belongs to the subset or has a neighbour in it. The implementation is not described in the original paper so we use a greedy approximation [5]: first, we select the node with the most uncovered neighbours and add it to the dominating set. Second, this node and all its neighbours are removed from the graph because they are now covered. The two steps are then repeated among the remaining nodes.

Core decomposition (Kcore) – This graph pealing algorithm [8] recursively removes the node of smallest degree until only a core of well-connected nodes remains. We use a binary heap structure to keep track of the degrees, leading to a quasi-linear time complexity.

Diameter (Diam) – Efficient approximations with theoretical bounds exist [9] to compute the aforementioned diameter Δ. In [1], the authors run 5000 times the shortest paths algorithm SP from a random node, and output the highest distance obtained. Note that the accuracy and efficiency of the algorithm are not key here, as the aim is to compare the performances in terms of computation time of different orderings.

2.2 Datasets and data structure

Size – Eight real-world datasets are used as benchmarks in the original work [1]. Their basic features are reported in Table 1. As per usual with real-world graphs [10], these graphs are sparse ($m \ll n^2$) and have small diameter and a skewed degree distribution, etc. As shown in Table 1, their sizes range from 1.6 million nodes and 30 million edges to almost 100 million nodes and two billion edges. In order to facilitate further experiments, we attach the *epinion* dataset to the repository, a smaller network on which our code can be tested quickly.

Sources – In the original paper, the sources are provided in the form of URLs where datasets can be downloaded. These data are available with the links given in Table 1.

Dataset	Size (Go)	Nodes (10^6)	Edges (10^6)	Source	Category
pokec	0.4	1.63	30.6	SNAP[1]	Social
flickr	0.4	2.30	33.1	Konect[2]	Social
livejournal	1.0	4.85	69.0	SNAP[1]	Social
wiki	6.7	13.6	437	Konect[2]	Web
gplus	7.3	28.9	463	Gong[3]	Social
pldarc	10	42.9	623	WDC[4]	Web
twitter	26	61.6	1470	Kaist[5]	Social
sdarc	34	94.9	1940	WDC[4]	Web
epinion (added)	0.005	0.0759	0.509	SNAP[1]	Social

Table 1. **General features of the datasets used in the experiments.** The data can be found in the following websites:
[1]Stanford Network Analysis Project: http://snap.stanford.edu/data/
[2]Koblenz Network Collection: http://konect.cc/networks/flickr-growth/ and http://konect.cc/networks/wikipedia_link_en/
[3]Gong Research Group: http://gonglab.pratt.duke.edu/google-dataset
[4]Web Data Commons: http://webdatacommons.org/hyperlinkgraph/2012-08/download.html
[5]Kaist Advanced Networking Laboratory: http://an.kaist.ac.kr/traces/WWW2010.html

Categories — The authors of [1] selected two main categories of real-world networks: online social platforms, where a node is a user and a directed edge represents a social interaction, and web graphs, where a node is a web page and an edge is a hyperlink.

Format — The datasets are directed graphs given as lists of edges. Most algorithms (*e.g.* computing shortest paths) have different results depending on whether edges are directed or not. In order to store large graphs in main memory, an efficient data structure is needed. Libraries exist for that purpose, but we develop our own light structure to have better control over the implementation of the algorithms. A list of edges does not provide quick access to the list of neighbours of a given node, which is the crucial operation for most of the above graph algorithms. The data is therefore converted into an adjacency list, where a node points to the list of its neighbours. To store it efficiently, we use a *Compressed Sparse Row* format, as described in Figure 2.

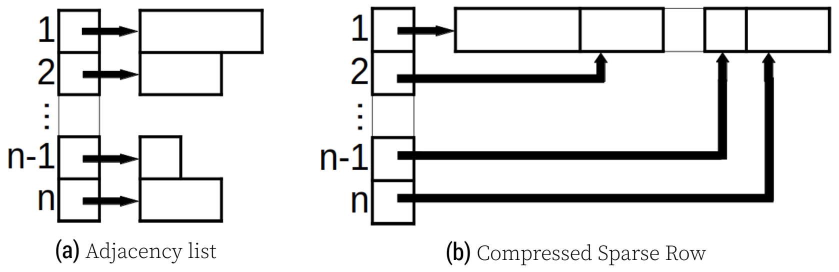


Figure 2. **Graph representations.** An adjacency list stores a list of neighbours for each node. In CSR, all the neighbours are stored in a shared array of size m, and each node has a pointer to its first neighbour. This is an equivalent but more compact format which allows for faster memory access.

2.3 Orderings

We list below the different ordering methods considered in the original study and used as a benchmark for comparison with Gorder.

Original — Datasets are collected in a way that is not random but is rarely reported. As shown in Section 3, the original orderings perform quite well regarding cache miss.

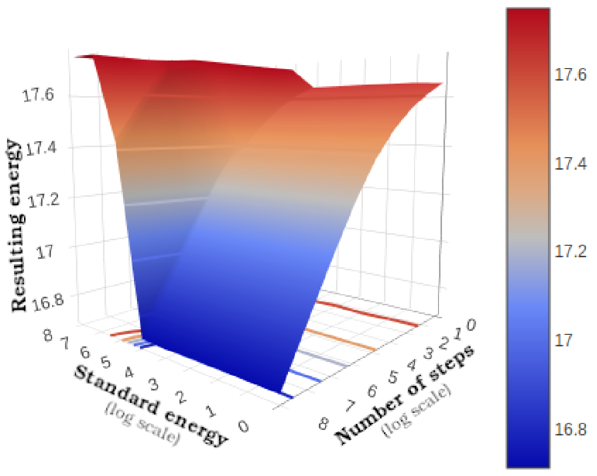


Figure 3. Tuning simulated annealing. The colour represents the energy of the best permutation π obtained for various parameters on *epinion*. Number of steps S ranges from n to $m \log n$ (logarithmic scale), which we consider to be the maximal acceptable time for the heuristic implementation. Standard energy k ranges from $1/(mn)$ to mn (logarithmic scale). We observe that a) the higher S, the lower the resulting energy, b) when k is high, the resulting energy is at a maximum: all swaps are accepted regardless of their quality which results in a random arrangement, c) any low value of k has the same result, in particular for $k = 0$ which corresponds to a local search.

Random (added) – In comparison to [1], we added a random ordering, obtained by shuffling the indices of nodes. We use it as a non-favourable benchmark for comparison to all other orderings.

MinLA and MinLogA – These acronyms stand for *minimum linear* (respectively *logarithmic*) *arrangement*. The goal is to find an arrangement π of the nodes that minimises a given "energy" function. The energy $\mathcal{E}$ is computed over the set E of edges in the following way:

$$\mathcal{E}_{MinLA} = \sum_{(u,v)\in E} |\pi_u - \pi_v| \qquad \text{and} \qquad \mathcal{E}_{MinLogA} = \sum_{(u,v)\in E} \log|\pi_u - \pi_v|$$

As both exact optimisations are NP-hard, a heuristic method is necessary. The authors of [1] use simulated annealing: random permutations are achieved to decrease the energy $\mathcal{E}$, while the temperature goes down which allows less and less modifications. Simulated annealing is known to be hard to tune. Our implementation has two parameters: the number of steps S and the standard energy k. The temperature T decreases linearly so that, at step s,

$$T(s) = 1 - s/S$$

At each step, two nodes are picked at random. Swapping their indices in π leads to a variation e of the total energy $\mathcal{E}$. If e is negative, the swap is registered. Otherwise, it is registered with a probability p, inspired by statistical physics:

$$p(e,T) = \exp\left(-\frac{e}{k\cdot T}\right)$$

In Figure 3 we test a wide range of values of S and k on *epinion*. While we cover a significant fraction of the parameter space, we are not able to find a combination of S and k that outperforms a simple local search ($k = 0, p = 0$), where only the favourable swaps are accepted. Below, we set $S = m$ and $k = m/n$.

RCM – The Reverse Cuthill–McKee ordering [11] is a Breadth-First Search where nodes of small degree are favoured. It is meant to find an arrangement π that reduces the bandwidth of a sparse graph, given by $\max_{(u,v)\in E} |\pi_u - \pi_v|$ with π_u the index of node u.

Degsort — As proposed in the original paper, nodes are sorted in descending order of in-going degree.

Chdfs — We assume that the *children-depth first search traversal* mentioned in [1] is a usual Depth-First Search algorithm. The first node is chosen at random, then the selection of children is made following the original order of node indices.

SlashBurn (simplified) — SlashBurn is an iterative process that separates hubs (high-degree nodes) from low-degree nodes connected to hubs. It creates an ordering by iterating over an array of size n, initially empty. Each iteration divides the array in parts A, B and C. Part A takes only one node, selected at random among those with highest degree. All isolated nodes go to part C. Then these nodes are removed from the graph which creates new isolated nodes, and degrees are updated. Part B is filled by the next iteration until no node remains.
The original SlashBurn algorithm [12] fills part C with disconnected components instead of isolated nodes and puts r hubs in part A, where r is a parameter. As no precise information was given in [1], we implement the simpler version described above instead.

LDG — Linear Deterministic Greedy partitioning [13] creates $\frac{n}{k}$ bins of size k and puts nodes in the bin where most of their neighbours belong. Larger bins are penalised: a node u with neighbours N_u is placed in a bin that achieves

$$\underset{\text{bin } B}{\arg\max} \Big(1 + |N_u \cap B|\Big) \times \Big(1 - \frac{|B|}{k}\Big)$$

At the end of the process, each bin contains about k nodes. In [1], the authors choose $k = 64$ so that a bin can fit on a cache line. Indeed, common contemporary processors have L1 caches of a few dozen kilobytes (32kB in our case) made of lines of 64 bytes each.

Metis (removed) — Metis is a powerful and extensive tool for graph partitioning. A C++ implementation is available[2] but it is not suitable for large graphs: the original paper could only test it on the three smallest datasets because of its excessive memory consumption. Since this ordering does not scale, we do not use it in our experiments.

Gorder — Gorder is the ordering method introduced in [1], where it is precisely described. A C++ implementation is available[3]. As mentioned at the beginning of Section 2, the authors define the quality function F of an arrangement π by:

$$F(\pi) = \sum_{0<\pi_u-\pi_v\leq w} S(u,v) = \sum_{0<\pi_u-\pi_v\leq w} \Big(S_s(u,v) + S_n(u,v)\Big)$$

where w is the window size; $S_s(u,v)$ is the number of times u and v coexist in sibling relationships or their number of common in-neighbours; $S_n(u,v)$ is the number of times they are in a neighbour relationship, which is either 0, 1 or 2 since both edges (u,v) and (v,u) may exist.
The greedy algorithm presented in [1] creates the ordering π by recursively inserting the node that has the highest proximity to nodes presently within the window. Storing the proximity scores S requires a complex structure called unit heap, made of a linked list and pointers to different positions. We took the functions provided in the original code and adapted them to our data structure.
In [1], Figure 8 shows how parameter w is selected. The authors create versions of Gorder for window sizes ranging from 1 to 8. For each version, they run the PR algorithm on *flickr* dataset. The fastest runtime is obtained with $w = 5$, so they use this value for subsequent experiments. However, the 8 versions only lead to a small relative variation of runtime (3%).

[2]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
[3]https://github.com/datourat/Gorder

In Figure 4 we compare a wider range of window sizes, because w could in theory be anything between 1 and n. We find that setting w between 64 and 2048 gives a further 3% speedup compared to $w = 5$.

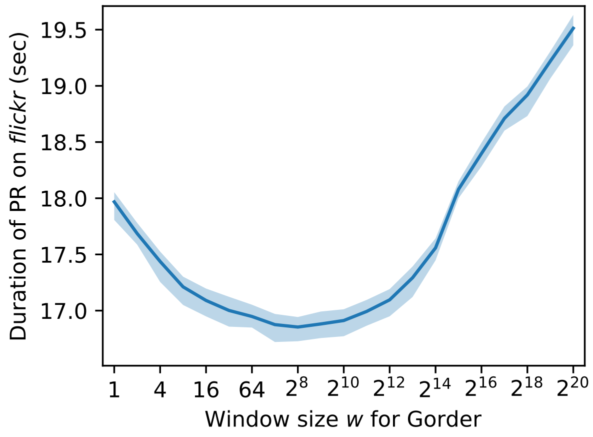


Figure 4. Tuning window size. Versions of Gorder obtained for window sizes ranging from $w = 1$ to $w = 2^{20} \simeq 10^6$ are tested in PageRank over *flickr* (with $n \simeq 2 \cdot 10^6$ nodes). Median and 90% confidence interval are shown for 100 repetitions. The plateau from $w = 64$ to $w = 2^{11} = 2048$ gives better results than $w = 5$. This figure can be compared to Figure 8 in [1]; note that the absolute runtimes are different because of hardware differences.

The choice of a small w is yet relevant because of two other factors: first, the computation of Gorder is faster when the window is narrow, because a candidate node has to compute its proximity score S with all the nodes of the window. Second, the authors of the original paper show that their heuristic is a $\frac{1}{2w}$-approximation of the optimal score: reducing w makes this bound tighter.
Considering all these remarks and for the purpose of replication, we also use $w = 5$ in the following experiments.

3 Results

3.1 Implementation hardware

To deal with bigger datasets and ensure stability, we run the experiments on an isolated cluster (SGI UV2000 Intel Xeon E5-4650L @2.6 GHz, 128GB RAM). Each processor has three levels of cache of respective size 32kB, 256kB and 20MB.
The hardware used in [1] has similar cache and RAM storage but higher clock frequency, which can explain the differences in runtime (in addition to programming techniques and optimisation). Note however that these differences should not modify the relative performance of different orderings.

3.2 Ordering time

Computing an ordering on a large network can be a long process, and some of the ordering methods have limited scalability. As mentioned above, Metis has been removed from the experiments for this reason. Table 2 reports the duration of the ordering processes. For datasets under a hundred million edges, they can all be computed in a couple of minutes at most, with DegSort and ChDFS orderings requiring less than a second.
When the number of edges rises however, the computation takes hours for MinLA, MinLogA, and Gorder. In the case of MinLA and MinLogA, the number of steps is chosen arbitrarily as described in Section 2.3. The process could thus be interrupted earlier, at the cost of a less efficient resulting ordering. As for Gorder, we can see that it does not scale linearly: the edges processed per second decrease from 380k for *pokec* to 60k for

sdarc, which requires an almost 9-hour-long computation. Using a smaller window size accelerates the process but slightly worsens the resulting ordering, as seen in Figure 4.

	pokec	*flickr*	*livejournal*	*wiki*	*gplus*	*pldarc*	*twitter*	*sdarc*
MinLA	28	27	92	441	539	579	**2956**	**4884**
MinLogA	89	64	217	**2169**	1662	**2258**	**10245**	**17168**
RCM	3	5	10	60	49	63	158	406
DegSort	0.8	0.4	1	5	9	14	30	85
ChDFS	1	0.8	1	3	8	10	54	76
SlashBurn	3	9	16	37	90	189	633	1066
LDG	6	7	13	68	101	144	673	798
Gorder w=5	79	110	118	988	**3324**	**8783**	**25475**	**32488**
Edges m	31M	33M	69M	437M	463M	623M	1.47G	1.94G

Table 2. Graph ordering time. We indicate the time to compute each ordering in seconds (in bold font when above 30 minutes). We also indicate the number of edges for each dataset to help evaluating the scalability of a method. Comparing to Table 9 in [1] is possible for RCM, DegSort, ChDFS and Gorder because the implementations are alike: it shows that our hardware is 2 to 5 times slower than in [1]. For the other orderings, the implementations are likely too different to be compared.

3.3 Running time

The main purpose of [1] is to measure if the node orderings listed above allow for faster execution of standard graph algorithms. We compare in Figure 5 the performances of all the orderings to Gorder. The results are reported for each dataset and algorithm in the same way as Figure 9 of the original paper. We also propose in supplementary figure S1 another visualisation of the same results but grouped by ordering instead of dataset, which emphasises the overall performance of an ordering method.
A first few observations can be made from the raw results of the experiments. As in [1], we observe that Gorder almost always leads to the fastest execution times. The speed-up factor reaches up to 2.5 compared to default for Diameter on *sdarc* and 3.7 compared to random for PageRank on *wiki*, but for the sake of clarity we limit the y-axis of Figure 5 to a factor 2.
To help making sense of the results, we propose an aggregated visualisation in Figure 6, where each ordering is ranked according to its performance in comparison to the other orderings through the 81 series of experiments reported in Figure 5. It shows in particular that Gorder is the best ordering method in half of the experiments, and second-best in most other cases.
Below, we comment on the performances of the different ordering methods under examination.

Original ordering — The experiments show that the default ordering performs better than more elaborate methods such as MinLA or MinLogA, which have a high computation overhead as shown in Table 2. It was also observed in [1]. This indicates that the way in which datasets are constructed tends to give close indices to nodes that are in the same neighbourhood. In a web graph for instance, if webpages are listed alphabetically by URL, it is likely that two consecutive nodes have a hyperlink between them since they belong to the same website.

Poorly performing orderings — The random ordering is always the worst performer except for 6 experiments where it is second-worst. It is not surprising as any other ordering tends to bring neighbouring nodes together, which should improve the algorithm runtime. Note however that LDG performs only slightly better than random, and that it is almost always the slowest in [1] too. In a quarter of the experiments, it is more than twice as slow as Gorder. These poor results lead to think that either its parameter (the size of bins $k = 64$) is not optimal, or that its quality function is not highly correlated to

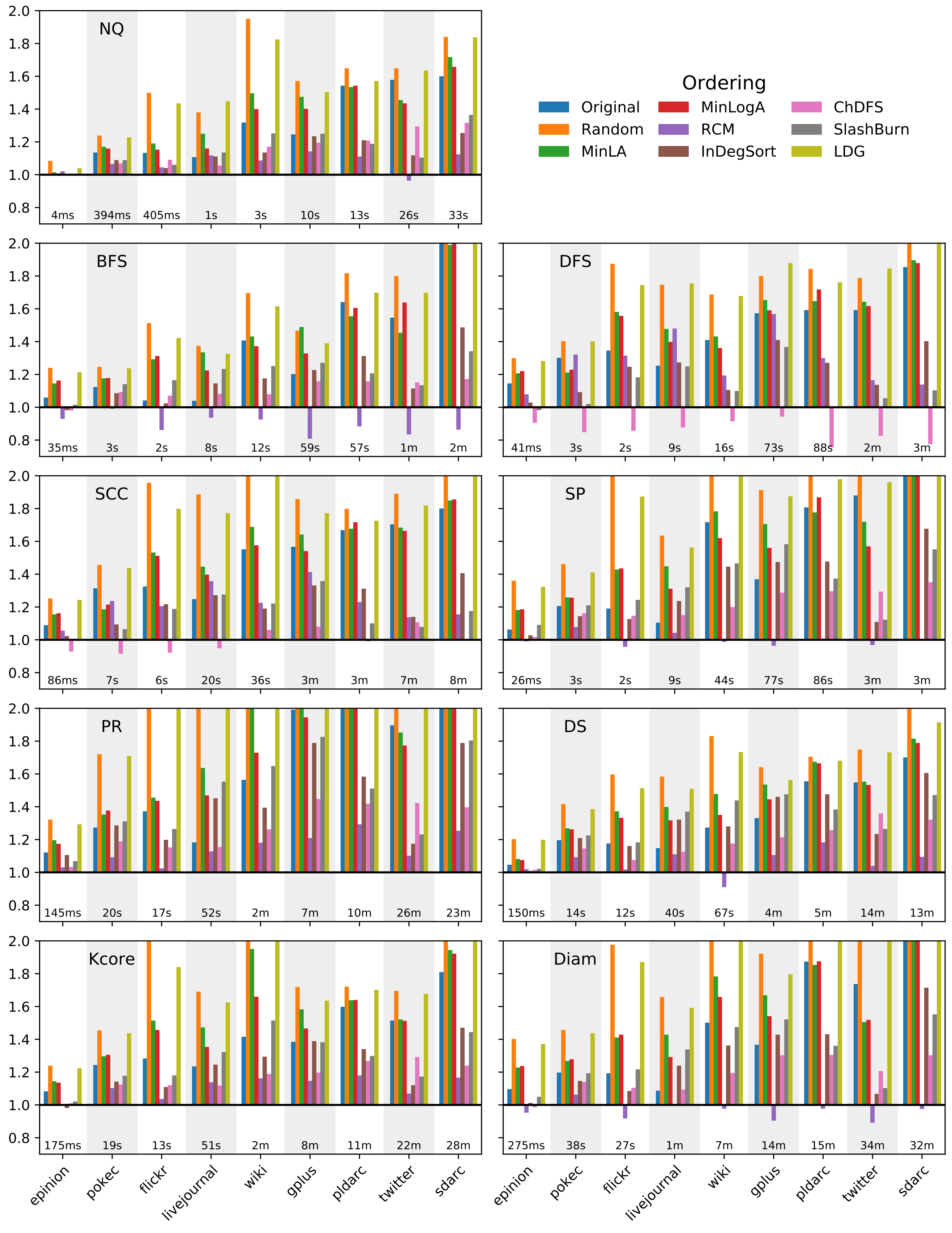


Figure 5. Speedup of Gorder. For each algorithm and each dataset, we display the absolute runtime for Gorder. Bars represent the relative time of all other orderings compared to this reference. For readability, the y-axis is cut above factor 2, but values go as high as 3.7. This figure can be compared to Figure 9 in [1]. Another visualisation is show in supplementary figure S1.

cache efficiency. MinLA and MinLogA are always faster than LDG but, except for *twitter* dataset, they are slower than the original ordering. As reported in Figure 3, we could not find any parameters with better results than local search, which is not ideal when the problem has local minima.

Degree-based orderings — Both InDegSort and SlashBurn use the degree of nodes as their main criterion. The experiments show that they outperform the default orderings, especially for larger datasets. For some algorithms such as BFS or NQ, they are less than 20% slower than Gorder. This indicates that cache misses are reduced when nodes of similar degree are copied together on a cache line. The original paper found similar results, though their implementation of SlashBurn did not perform as well; the different version that we use here (see Section 2.3) may be responsible for this discrepancy.

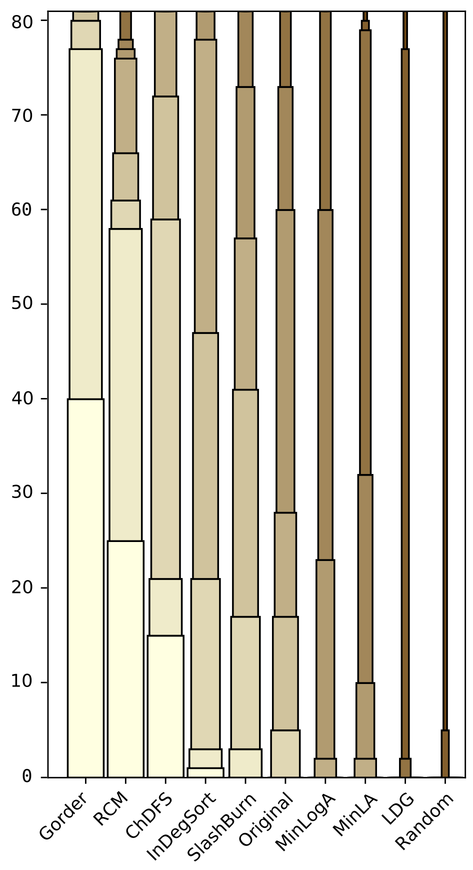
Orderings outperforming Gorder on specific algorithms — We also notice that some orderings perform particularly well on specific algorithms, even outperforming Gorder. The ChDFS ordering is the most efficient for DFS algorithm on all datasets. This is due to the close relation between these two processes: the algorithm explores the graph in the exact same way as the ordering is created. Likewise, RCM is a variation of a BFS that takes node degrees into account and it is the most efficient ordering for BFS algorithm.
Both also outperform Gorder for algorithms that are not as visibly related: ChDFS is up to 10% more efficient for SCC on smaller datasets, and RCM is the most efficient for Diameter and SP. More generally, Figure 6 shows that these two orders are among the three fastest ones in 75% of the experiments. The original paper has different results on that matter: RCM and ChDFS are the best alternatives as well, but they are always 10 to 20% slower than Gorder.

3.4 Comparison to the original paper

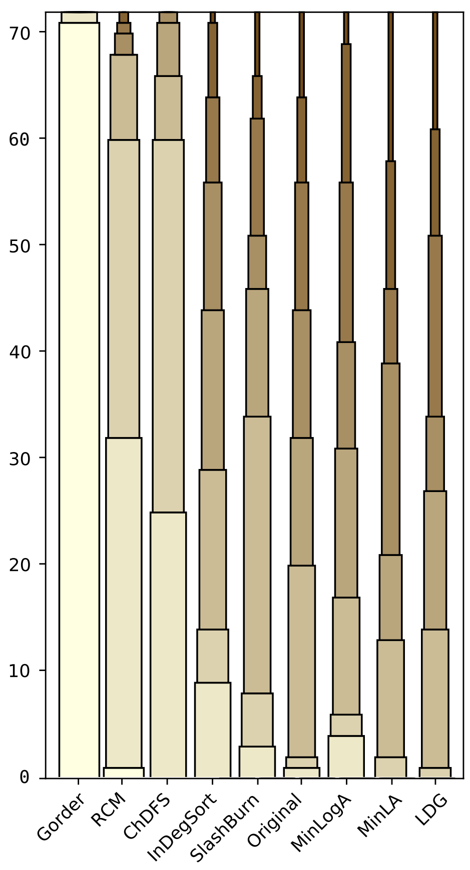
Our purpose here is to detect if there are significant discrepancies in performance between the original paper and our replication study.
Figure 6 presents an aggregate view of the results grouped by ordering method. For each series of experiments (*i.e.* a given algorithm applied to a given dataset), we rank ordering methods from best to worst performance. The figures report how many times each ordering has been ranked in each position. For instance Gorder is ranked first in 40 of our 81 replication series.

Results for Gorder — In both the original study and our replication, Gorder ranks first overall. This shows that this ordering is the best choice overall. However, our study shows that Gorder is outperformed by different orderings in half of the experiments, while it is only outranked once in the original paper. Figure 6b thus leads us to think of Gorder as the perfect choice whereas Figure 6a establishes RCM and ChDFS as relevant challengers, with 24 and 16 first places respectively. This difference between the two papers is probably due to implementations: in our replication, ChDFS uses exactly the DFS algorithm. In particular, the nodes are visited in the same order, which leads to a quick execution of DFS. The original paper likely prevents this mechanism, for instance by shuffling the nodes at each step of the search.

Ranking of other orders — The three best orderings are evidently the same in both studies, but there are some nuances for the other ones. First, this visualisation does not always allow for exact ranking: in Figure 6b, InDegSort has more second and third places than SlashBurn but fewer fourth and fifth places. There is no obvious way of deciding which is better, while Figure 6a clearly indicates InDegSort. The same issue happens between Original and MinLogA: we can say that their rank in [1] is equal.
The last nuance comes for the slowest orderings: in our study, LDG is only better than Random while MinLA competes with MinLogA and Original. In [1] on the other hand, LDG is better than MinLA overall. Still, the limit at factor 1.5 makes the ranking unreliable for slowest orderings which can explain this difference.



(a) **Rankings in our 81 experiments** (9 algorithms times 9 datasets). Random ordering was added and Metis was ignored.



(b) **Rankings in [1]** (72 experiments as *epinion* is omitted). Rankings are inferred from Figure 9 of [1]: values above the factor 1.5 limit are considered equal.

Figure 6. Rankings of ordering methods. For each series of experiments, we rank the runtime performance of the orderings. This figure shows how many times an ordering ranks best (thickest, lightest bar), second-best, ..., to worst (thinnest, darkest bar).

In the end, both studies rank the orderings in a very similar way.

Limits of visualisation – The aggregate view of Figure 6 induces several approximations. First of all, there are 72 experiments in the original study and 81 experiments in ours, which adds *epinion* dataset. This extra dataset is much smaller than the others and all its results in Figure 5 range in a 40% factor, to be compared with more than 200% for the biggest datasets. Yet, the ranking of ordering methods on *epinion* is consistent with other datasets.
The original study does not test random orderings. This does not disturb the results as this method ranks last in most experiments. Similarly, our study omits Metis so we ignore it in Figure 6b as well. Moreover, the original paper hides precise information when a runtime exceeds 1.5 times the runtime of Gorder. We consider that all orderings above this bound are equal.
The main issue is that this visualisation only shows the rank and hides the extent of runtime variations. This information is only visible in Figure 5, where a rift separates two categories: faster orders with Gorder, RCM, ChDFS, InDegSort and SlashBurn, from slower orders with the other ones. However, Figure 6 is useful to grade ordering methods. If original ordering is taken as a limit between faster and slower orders, we find the same gap again.

Order	L1-ref (10^9)	L1-mr	L3-ref (10^9)	L3-r	Cache-mr
Original	29	15.9 %	2.8	9.8 %	2.5 %
Random	30	20.2 %	4.1	13.6 %	3.6 %
MinLA	29	16.2 %	3.3	11.4 %	2.5 %
MinLogA	28	15.9 %	3.1	10.7 %	2.5 %
RCM	30	11.5 %	1.8	5.9 %	**1.6 %**
InDegSort	28	14.7 %	2.5	9.1 %	2.2 %
ChDFS	29	12.8 %	2.1	7.2 %	1.8 %
SlashBurn	28	14.8 %	2.6	9.3 %	2.2 %
LDG	30	19.2 %	3.7	12.4 %	3.2 %
Gorder	28	**10.3 %**	**1.4**	**5.0 %**	1.7 %

(a) On *flickr* dataset.

Order	L1-ref (10^9)	L1-mr	L3-ref (10^9)	L3-r	Cache-mr
Original	1885	19.0 %	303	16.0 %	6.8 %
Random	1886	23.4 %	397	21.0 %	9.0 %
MinLA	1893	21.2 %	341	18.0 %	7.1 %
MinLogA	1885	20.7 %	330	17.5 %	6.9 %
RCM	1885	11.0 %	139	7.4 %	3.7 %
InDegSort	1779	15.0 %	198	11.1 %	6.0 %
ChDFS	1863	11.8 %	153	8.2 %	4.3 %
SlashBurn	1784	15.3 %	203	11.4 %	6.0 %
LDG	1886	22.9 %	387	20.5 %	8.8 %
Gorder	1816	**9.3 %**	**104**	**5.7 %**	**3.1 %**

(b) On *sdarc* dataset.

Table 3. Cache statistics measured for PageRank algorithm. L1-ref (references): number of times a piece of data was required by the processor and searched in level 1 of cache. L1-mr (miss-rate): proportion of data that was not found in L1. L3-ref: number of references to the third (lowest) level of cache after data was not found in levels 1 and 2. L3-r (ratio): proportion of data that was not found in L1 nor L2, then searched in L3. Cache-mr: proportion of data that was not found in cache (L1, L2 or L3) and had to be retrieved in main memory. Compare to Tables 3 and 4 in [1].

3.5 Cache miss

A cache miss is a state of an execution when the data requested by the processor is not found in the cache memory. The program has to fetch the data in further cache levels or in main memory, which causes delays. Gorder capitalises on the intuition that if we cluster nodes that are frequently accessed together, higher levels of cache will hold more relevant data and thus make algorithms run faster.
To prove that Gorder speedup is due to cache optimisation, we compute the proportion of the total computation time spent in data retrieval. We use Unix *perftools* with the wrapper *ocperf*. It provides various hardware metrics such as the number of CPU cycles, branch predictions, cache misses...[4] Depending on the machine architecture, different metrics are available.
Table 3, just like the Tables 3 and 4 of the original paper, shows the cache-miss rates at different levels. The first column is the total number of L1-references which is the number of times a piece of data was required by the processor. A proportion of this data is not found (second column) and requested in intermediate levels of cache, until reaching L3 (third and fourth columns). The remaining data (last column) has to be retrieved in main memory. Note that each further level of cache roughly implies an additional factor 4 latency.
We observe that first-level cache references are similar for all orderings: the algorithms run in the exact same way so they need to access the same amount of data, regardless

[4]See https://perf.wiki.kernel.org. We select the following counters: the total time *task-clock*, *cpu-cycles*, *L1-dcache-loads* and *L1-load-misses* to measure the efficiency of the first layer of data cache (we are not interested in instruction cache here), *LLC-loads* and *LLC-load-misses* to measure the efficiency of the last cache layer, and metrics specifically designed to measure the impact of cache misses such as *cycles-l1d-pending* or *cycles-l3-miss* in the cycle-activity category.

of the arrangement of nodes. However, the miss-rate in L1 reveals important variations: with Gorder, only 10% of the data is not directly available in L1, while it reaches 20% with Random or LDG orders. The percentage of data requested in L3 is even more scattered, from 5% with Gorder to 20% with Random or LDG on *sdarc*. Finally, all the orderings have a low cache-miss rate (between 1.6 and 3.6%) on *flickr* and RCM has the smallest. The gap is more striking on *sdarc* where Random and LDG have 9% of cache-miss, three times as much as Gorder. This ratio is the proportion of data that had to be retrieved in main memory (RAM), which is about 60 times slower than the L1 cache[5].
In general, the ranking for cache-miss rates matches the ranking for runtime shown in Figure 6a. Gorder has the best results and RCM and ChDFS are close behind. MinLA, MinLogA and Original orders have high cache-miss rates for all levels. This shows that the speedup is indeed due to cache-miss reduction. When compared to Original order, Gorder reduces cache-miss rates to speed up the algorithms. Figure 1 shows that the total runtime is reduced by 15 to 50% on *sdarc*, but the CPU execution time is almost identical: it is the factor 3 reduction on cache stall that makes the algorithm faster.

4 Discussion

Our experiments replicate the ones proposed in [1] but some aspects are not discussed in sufficient detail in the original paper to allow for immediate replication. For instance, the algorithms NQ and DS only have a succinct description, which may explain why their performances reported in our Figure 1 do not align perfectly with Figure 1 of [1]. Similarly, we were not able to tune the simulated annealing procedure correctly, which questions the relevance of our experiments with MinLA and MinLogA. Nonetheless, most of these technical issues have been solved or circumvented thanks to the answers of Hao Wei to our questions.
Above all, this study replicates the main observation of the original paper: Gorder reduces cache latency significantly and is the best performer among all the ordering methods under study, as shown in Figure 6. Its consistent efficiency on all algorithms and datasets suggests that it could speed up other graph algorithms as well.
The only important difference with the original paper is that RCM and ChDFS follow closely behind Gorder and even outperform it in half of the experiments. They are based on straightforward graph searches, which are simple to program and very quick to execute, as reported in Table 2. On the other hand, computing Gorder requires a complex procedure and a lot of time. It has been pointed out in [14] that this high overhead time can only be amortised if algorithms are run thousands of times. In the case where networks evolve and require constant recomputation of the node ordering, Gorder needs to be adapted to integrate the modifications without running the whole process again. A parallel version of Gorder could reduce this problem.
Beyond algorithm speed-up, the contribution of Hao Wei *et al.* is an efficient framework that could be applied for other purposes. For example, graph compression also benefits from orderings that cluster nodes with high proximity [15]. Gorder could be an input for such existing methods. It would also be interesting to investigate how different types of real-world datasets [16] behave when a new ordering is applied.

5 Conclusion

The replication of paper [1] shows that Gorder is an efficient cache optimisation for various standard algorithms. We confirm its superiority for networks ranging from 30 million to 2 billion edges, and the hardware measurement tools prove that this is due to reduced cache stall. However, orders such as DFS are among the best performers, and they are much simpler to design. Indeed, the computation of Gorder does not scale linearly, which makes it very time-consuming for bigger graphs. It is then a matter of

[5]At 4GHz, a cycle is $1c = 1/(4 \cdot 10^9)s = 0.25ns$, so latency is $4c = 1ns$ for L1 and $42c + 51ns \simeq 62ns$ for RAM. Values are taken from https://www.7-cpu.com/cpu/Skylake.html.

balance between this long overhead time and the substantial speedup for subsequent graph algorithms.

Acknowledgements

We warmly thank Hao Wei for replying to our questions about the original paper, which settled several technical hesitations relative to the implementation.
This work is funded by the ANR (French National Agency of Research) partly by the LiMass JCJC project (under grant ANR-19-CE23-0010) and partly by the ANR FiT LabCom.

References

1. H. Wei, J. X. Yu, C. Lu, and X. Lin. "Speedup Graph Processing by Graph Ordering." In: SIGMOD. 2016.
2. A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. "DBMSs on a modern processor: where does time go?" In: VLDB. 1999.
3. J. Cieslewicz and K. Ross. "Database Optimizations for Modern Hardware." In: Proceedings of the IEEE (2008).
4. M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. "The More the Merrier: Efficient Multi-Source Graph Traversal." In: Proceedings of the VLDB Endowment (2014).
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. **Introduction to algorithms**. MIT press, 2009.
6. R. Tarjan. "Depth-First Search and Linear Graph Algorithms." In: SICOMP (1972).
7. L. Page, S. Brin, R. Motwani, and T. Winograd. "The PageRank citation ranking: Bringing order to the web." In: Stanford InfoLab (1999).
8. V. Batagelj and M. Zaversnik. "An O(m) algorithm for cores decomposition of networks." In: arXiv (2003).
9. D. G. Corneil, F. F. Dragan, and E. Köhler. "On the power of BFS to determine a graph's diameter." In: Networks (2003).
10. M. Latapy and C. Magnien. "Measuring Fundamental Properties of Real-World Complex Networks." In: arXiv (2006).
11. E. Cuthill and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices." In: ACM. 1969.
12. Y. Lim, U. Kang, and C. Faloutsos. "SlashBurn: Graph Compression and Mining beyond Caveman Communities." In: TKDE (2014).
13. I. Stanton and G. Kliot. "Streaming graph partitioning for large distributed graphs." In: ACM SIGKDD. 2012.
14. V. Balaji and B. Lucia. "When is Graph Reordering an Optimization?" In: IEEE IISWC. 2018.
15. P. Boldi and S. Vigna. "The webgraph framework I: compression techniques." In: WWW. 2004.
16. R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon. "Superfamilies of Evolved and Designed Networks." In: Science (2004).

Supplementary material

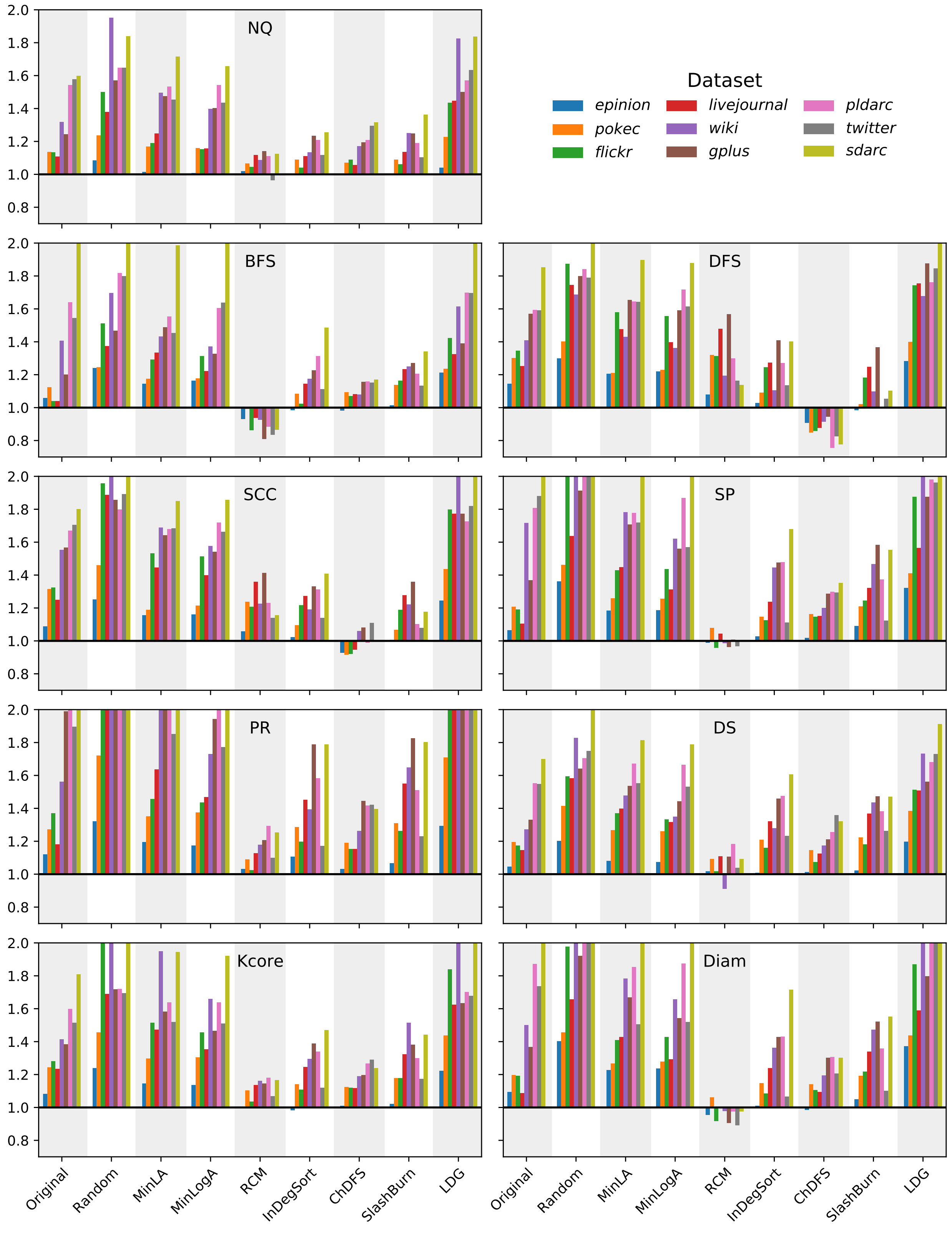


Figure S1. Speedup of Gorder grouped by ordering. For each algorithm and each ordering, bars represent the relative duration on each dataset compared to Gorder, taken as a reference. For readability, the y-axis is cut above factor 2. This figure displays the same information as Figure 5 but groups bars by ordering instead of dataset.