An analysis of the scale-invariance of graph algorithms: A case study.

July 10, 2018

Tim van Zalingen
University of Amsterdam
tim.vanzalingen@os3.nl

Supervisors:
Merijn Verstraaten (University of Amsterdam)
Ana Lucia Varbanescu (University of Amsterdam)

***Abstract*—Research in graph processing algorithms resulted in many implementations that aim to make efficient use of parallelism. Especially on GPUs, many implementations exist. There is a lot of research into the relative performance of GPU-based graph algorithm implementations for graphs with differing topology. Knowledge on what algorithm implementation to use can significantly speed up graph processing. In order to benchmark real-world graphs with topological constraints, but for different scales, a scaling mechanism has recently been developed. This paper shows whether the relative performance of Breadth First Search implementations on the GPU is scale-invariant. First, we assess the scaling mechanism. Second, we examine the relative performance of the Breadth First Search implementations at different scales for a diverse set of graphs. We find that the relative performance is stable, but not scale-invariant and that one implementation shows better scalability that the other implementations. Third, we investigate when this implementation scales better than the others. This experiment hints that such a transition point might be predictable. Lastly, a comparison is drawn between the previous experiments, that make use of scaled real-world graphs, and generated Graph500 graphs.**

I. Introduction

The field of graph processing is ever expanding. From research and networking to social media giants, graphs are increasing in complexity. With the increase in data, the processing of large graphs is becoming more relevant than ever before. There are challenges in making efficient use of parallelism for graph processing [1].

In order to speed up large graph processing, algorithms have been developed that exploit Graphics Processing Units (GPUs). Because computation power is cheap on GPUs, graph processing can be more efficient. However, how much more efficient the processing becomes depends on the implementation and type of graph.

The performance of graph algorithms is usually tested by benchmarking public graph repositories, such as SNAP and KONECT [2, 3], but these repositories are limited in size and variety. When performing such benchmarks, we might be interested in graphs with specific topological properties. However, these graphs might not be available in the required size. Too large a graph might take too much time to benchmark. On the other hand, we might be interested in the scalability of an algorithm and would want to benchmark a graph on different sizes.

In order to meet the demand in research, a mechanism has been developed to scale graphs [4]. This allows graphs to be scaled to the required size. The scaling mechanism, developed by Musaafir, can create a family of graphs from an original graph that is used as seed. This scaling method takes samples from the original graph. This sampling preserves as many graph properties as possible. If the graph needs to be scaled up, these samples are combined to form the new, larger graph. Musaafir shows that tweaking different parameters for this combination effects the topology of the resulting graph.

It is also possible to generate graphs of a specific size [5–7]. However, generation with required topological constraints is a challenge. There is no single framework that can generate multiple classes of graphs and some classes of graphs can not be generated at all. In addition, it can be a challenge to get close to real life graphs.

The performance of different GPU-based graph algorithms implementations depends on structural properties of the graph. Examples of such properties are: Number of vertices, average degree, cluster coefficient. These graph properties can be used to predict the relative performance of graph algorithms [8–10]. Verstraaten *et al.* compared different implementations and variants of Breadth First Search (BFS) for different graphs from the KONECT repository. Similar research was performed by Merrill *et al.* Besides BFS, they also included the single-source shortest path (SSSP) algorithm. Both used their findings to predict what algorithm to use for a given graph, achieving a significant performance gain.

During runtime the most efficient implementation, what algorithm implementation performs best, changes. As such, performance can be improved if the algorithm implementation switches at runtime.

Knowing what algorithm implementation performs best is especially relevant for large graphs, where processing times are long. It is important to know what implementation performs best for certain classes of graphs. Once an implementation is chosen, will it perform well on similar graphs of other sizes?

This research focuses on *whether the relative performance of different graph algorithm implementations is scale-invariant*. In other words, if we scale a graph, we investigate whether the relative performance-based ranking of the different BFS algorithm implementations is preserved. While scaling up graphs, other properties except for the size are

preserved. The scaling mechanism allows for different parameters. Does the scaling method affect the performance at different scales? Different algorithm implementations are tested. Does the relative performance stay the same? Instead of using real-world graphs, generated graphs can also be used to measure the impact of size on algorithm implementation performance. Are there any significant differences between scaling and generating graphs when examining the relative performance of algorithm implementations over scale?

II. Background

A. Definitions and concepts

1) Scale-invariance: If the performance-based ranking of the algorithm implementations over different scales on the same graph stays unchanged, we observe scale-invariance. On the other hand, if the ranking of the algorithm implementations does change, the behaviour is scale-variant.

2) Scalability: What happens to the performance of an algorithm implementation once scaled is considered to be the scalability of the implementation. If an implementation shows to relatively perform better than another implementation on a larger scale graph, it shows better scalability.

When a graph is scaled, the input size to the GPU increases. As such, the number of threads in use rises. For these implementations, as explained in section II-D, each individual thread still handles the same amount of information. As such, because the problem size per processor is fixed, we talk about weak scaling.

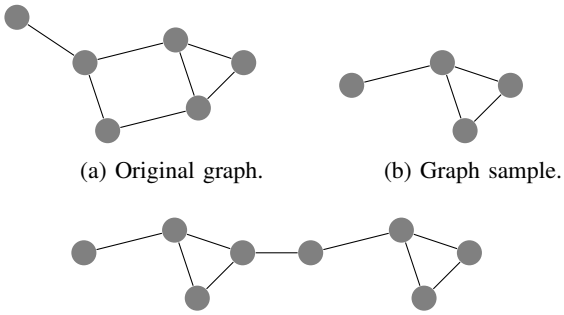
3) Transition point: In this paper, we compare the performance of graph algorithm implementations. It might be possible that one implementation starts to outperform another. Such an occurrence will be called a *transition point*.

B. GPU processing

The CUDA programming model allows for programmers to utilise the GPU [11, 12]. Code runs parallel on a collection of *threads*. These threads are grouped into *warps*. All the threads in a warp execute the same instruction. This allows for highly parallel processing of large regular data. While the representation of graphs is regular, the graph itself might not be. This can lead to workload imbalance. As explained in section II-D, the efficiency of an algorithm implementation depends on how well all threads in a warp are utilised. Not being able to execute the same instruction within a warp, leaves idle threads, resulting in a loss of performance.

C. Sample-based graph scaling

Sample-based graph scaling works by combining samples of an original graph into a new larger graph. An example of such a mechanism is shown in figure 1. First an input graph, as shown in figure 1a, is required. From this input graph, a sample is taken. Figure 1b shows a sample of $\frac{2}{3}$ the number of vertices from the original graph. Following the sampling, the samples can be combined to form a new larger graph. Figure 1c shows how two of the earlier created samples are combined to form a graph of $\frac{4}{3}$ the number of vertices of the original graph.



(a) Original graph. (b) Graph sample.

(c) Scaled graph.

Fig. 1: An example of sample-based graph scaling. The original graph is the input graph for the scaling mechanism.

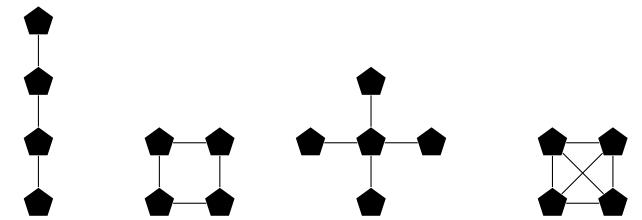


Fig. 2: Illustration of the different topologies. Each polygon represents a graph sample.

The graph scaling mechanism, developed by Musaafir, takes samples of the original graph and combines these into a larger graph. These samples should preserve topological features of the original graph or, in other words, the underlying structure should be similar. Note that, unlike the earlier example in figure 1, each sample can be different. One of the parameters, the sample size parameter, determines how large the samples are.

Several other parameters in the scaling mechanism can be tweaked: The topology, type of bridges between graphs and number of bridges between graphs. Because the scaling mechanism is sampling based, these sample can be connected in different ways. This is only relevant when a graph is scaled up. When scaling down, the single sample does not need to be connected to anything.

The topology determines how the different samples are connected. The topologies are: Chain, ring, star and fully connected. These topologies are illustrated in figure 2. In a chain, each sample is connected to the next, except for the last sample, forming a chain. For a ring topology, the samples form a ring, each connected to the next. In a star, one central sample is connected to all other samples. A fully connected

topology has each sample connected to every other sample.

Earlier results by Musaafir have shown that the topology mainly has an effect on the diameter. A chain, for example, tends to increase the diameter more than the other topologies. This is because the only way to reach a vertex in the last graph of the chain from the first, is by traversing all the other graphs. This is not true for the other topologies.

We can either create a bridge between samples by randomly selecting a vertex in each graph or by using vertices with a high degree. A consistent finding in earlier experiments is that picking high degree vertices decreases the average shortest path in the graph. When a path is searched between two samples, it is likely to be shorter if the samples are connected by vertices with a high degree.

The last parameter allows for specifying the amount of connections between samples. If we would connect samples with only a single edge for each connection, the average degree and density of the graph decreases. This problem is alleviated by creating multiple edges for each connection between two samples. Tweaking this setting has an effect on the average degree, diameter and density of the graph.

D. BFS implementations

We use five main implementations of BFS: Edge-list, reverse-edge-list (rev-edge-list), vertex-pull, vertex-push and vertex-push-warp.

Both the edge-list and reverse-edge-list implementations are edge-centric implementations, because they launch one CUDA thread per edge. The edge-list uses the outgoing edges, whereas the reverse edge-list uses the incoming edges. Because every thread has the same amount of work, the edge-centric implementations do not suffer from workload imbalance. However, because a large amount of threads are spawned, they result in many parallel updates and contested atomic updates.

The vertex-centric implementations, vertex-pull and vertex-push, launch one CUDA thread per vertex. The pull implementation updates its own BFS level. The push implementation updates the BFS levels of its neighbours. Because the vertex degree in a graph varies, these vertex-centric implementations suffer from workload imbalance, especially if vertices with a highly varying degree are in the same warp. The pull implementation only touches a vertex once, therefore no atomic operations are performed. The frontier is the vertices that the algorithm is currently searching in. If no neighbours are in this frontier however, the neighbours are iterated for nothing, wasting time. The push implementation prevents processing irrelevant neighbours, but in turn requires more atomic operations.

The vertex-push-warp implementation is based on the vertex-push implementation. Instead of each vertex being process by one thread, a group of threads is assigned a number of vertices. Every edge in a vertex is processed by this group of threads. When completed, the group of threads moves on to the next vertex. This attempts to reduce the workload imbalance present in the vertex-centric implementations.

III. Related Work

A. Difference between algorithms

Foggia *et al.* have performed research into graph matching algorithms. Despite the fact that these algorithms do not run on a GPU, it does show some insight into the relative behaviour of different graph algorithms. This research shows that the relative performance is dependent on properties of the graph and that some algorithms perform better on large graphs than others [13]. When comparing two similar algorithms, Cordella *et al.* show that one algorithm only outperforms another when the graph is large enough [14]. Similar research in this field by Carletti *et al.* compares several graph matching algorithms [15]. The ranking of the algorithms stays the same for different graph sizes, however, the behaviour under scaling differs. Something similar is investigated by Voss and Subhlok, finding that how well an algorithm scales also depends on the topology [16]. One should note that the graphs used by these researchers are orders of magnitude smaller than the graphs from KONECT used in this paper.

Harish and Narayanan implemented several algorithms to process graphs on the CPU and the GPU using CUDA [12, 17]. Besides confirming that the efficiency of an algorithm depends on graph topology, they also compared the algorithms on similar graphs of different sizes. The relative performance appears to be quite stable or, in other words, the ranking stays the same. However it is unclear what graphs are used in the experiments. Once real world graphs are used, this stability seems to diminish.

B. Generation of graphs

As an alternative to graph scaling, the required graphs could be generated. We could attempt to control topological features in the graph generation. In doing so, it might be possible to generate similar graphs of different sizes. However, the current graph generators are not flexible enough.

Miller, Joel C and Hagberg, Aric created a method for random network generation. These networks can be created for any given degree [6]. Milo *et al.* combine two existing methods into a similar approach [18]. However, this approach is limited to directed graphs. Because there is more to graph topologies than just degree, utilising such an approach for performance benchmarking would limit the generality and applicability of these experiments.

Guo and Kraines have been able to generate graphs and tune the degree, clustering coefficient and power law [19]. While this approach is able to control multiple topological factors, it is limited to small graphs and doesn't cover all graph features of interest. The authors conclude that generators that control multiple topological constraints, in a similar manner, are rare.

There are also graph generation methods based on existing graphs. Ying, Xiaowei and Wu, Xintao propose such a method [5]. In order to create graphs, random edges of the original graph are selected and switched. The authors note that it is not possible to preserve multiple features at a time. Therefore, utilising this method can not guarantee all the topological

constraints to be maintained. Besides, the size of the new synthetic graph is limited to that of the original.

Using evolutionary computing, Verstraaten *et al.* have been able to create a synthetic graph generator [7]. This method allows for creation of graphs with topological constraints. Graphs of thousands of vertices were created with this method in minutes. The authors note that there are issues with the scale of the generated graphs. It was not possible to scale the graph generation to a sufficient size to do benchmarking [8].

In conclusion, graph generation shows significant limitations. The methods that scale well, do not preserve multiple topological constraints. The methods that do preserve such constraints, can not scale to a significant size. As such, graph generation does not suffice for benchmarking purposes when comparing algorithms, where the topology of a graph is of great importance. Compared to graph repositories, generated graphs can be less noisy. This would be an advantage when performing benchmarking [8].

IV. Method

In order to examine the scale-invariance of graph algorithms, we adopted an empirical research method, based on benchmarking. Specifically, we selected a set of graphs, scaled them to various sizes and measure the execution time. This set is comprised of graphs from the KONECT repository, used by Verstraaten *et al.* in earlier work. We first investigate the impact of using different scaling parameters.

The computation time of BFS depends on the vertex it starts from, the root vertex. Because the scaled graph is different, we can't use the same root in the scaled versions. During earlier experiments, this proved to make the performance under different scales highly unstable. To mitigate this effect, the experiment is repeated for a number of root vertices. Each implementation on one scale starts from the same set of root vertices. Additionally, the computation time can differ for each run, so we average each traversal over a number of runs. The exact number of iterations is explained in the results section, see V.

A. Scaling

In order to tweak the scaling parameters, we scale 2 graphs with 18 different scaling parameters and benchmark BFS computations for all these graphs. A comparison between the results, with differing scaling parameters, should show the effect of the parameters on the relative performance of algorithms on different scales. The parameters that best show the expected behaviour of scale-invariance, are used in further experiments. All four topologies, both random and high degree connections and 1, 5 and 10 edges per connection are considered.

B. Algorithm scalability comparison

Algorithm implementations may show different scalability. This behaviour could depend on graph properties as well. Therefore we benchmarked 5 different BFS implementations on a diverse set of graphs. This set is scaled down to 0.5 and scaled up to 2, 4, 8 and 16 times its original size with a sample size of 0.5. The time it takes to process a graph also depends on the starting root node. Because the root node changes when a graph is scaled, we take an average over 20 different root nodes for each individual scale. Depending on the results, we may perform additional experiments to investigate anomalies or scale-variant behaviour.

TABLE I: Topological details on the Graph500 graphs.

Level	Size (vertices)	Volume (edges)
12	3353	48358
13	6467	101959
14	12550	213088
15	24196	441406
16	46815	909601
17	90116	1864262
18	173692	3805027
19	335294	7740825
20	646127	15700394
21	2396657	64155735
22	4610222	129333677
23	8870942	260379520
24	17062472	523602831

C. Graph500

In order to see the difference between scaled and generated graphs, we run the same benchmarks as before on Graph500 graphs [20]. The Graph500 graphs are generated with a generator that is similar to the R-MAT generator [21]. Different levels of the Graph500 graphs are available, each increase in level doubles the size of the graph. Information on the graphs is provided in table I. All Graph500 graphs are undirected.

V. Results

A. Experimental setup

All experiments described in this paper we perform on the DAS-5 cluster [22]. We use an Nvidia GTX TitanX Maxwell generation cards with 12 GB of onboard memory [23].

We use the sample-based graph scaling mechanism developed by Musaafir, as mentioned in section II-C. We also use the BFS implementations, as mentioned in section II-D, and the GPU-based graph algorithm benchmarking framework that were developed by Verstraaten *et al.*

Throughout this paper we make references to the structure of graphs in order to interpret the results. Table II shows such information on all the graphs that we use. Note that the graphs have diverse properties.

B. The effects of scaling parameters

We used both the `DBpedia` and `actor-collaboration` graphs for the experiments investigating the effects of the scaling parameters. These graphs are some of the larger graphs in the repository, but not large enough to take extensive time to scale and benchmark. We scale the `actor-collaboration` and `DBpedia` graphs up to 4 times their original size.

TABLE II: Topological details on the graphs used in this research from the KONECT repository [24].

Code	Name	Type	Edge weights	Size (vertices)	Volume (edges)	Average degree	Cluster coefficient	Diameter
CL	actor-collaboration	Undirected	Multiple unweighted	382219	33115812	173.28	16.6%	13
TH	arXiv hep-th Coauthorship	Undirected	Multiple unwieghted	22908	2673133	233.38	16.9%	9
GC	Google.com internal	Directed	Unweighted	15763	171206	21.72	1.33%	7
DB	DBpedia	Directed	Multiple unweighted	3966924	13820853	6.97	0.014%	67
DI	Discogs	Bipartite	Multiple unweighted	3780417	14414659	-	-	22
PL	Prosper loans	Directed	Multiple unweighted	89269	3394979	76.06	0.31%	8
UC	UC Irvine messages	Directed	Multiple unweighted	1899	59835	63.02	5.68%	8
ND	Notre Dame	Directed	Unweighted	325729	1497134	9.19	8.77%	46
HUi	Hudong internal links	Directed	Unweighted	1984484	14869484	14.99	0.35%	16
AS	Route views	Undirected	Unweighted	6474	13895	4.29	0.96%	9
IN	CAIDA	Undirected	Unweighted	26475	53381	4.03	0.72%	17
TH	arXiv astro-ph	Undirected	Unweighted	18771	198050	21.10	31.8%	14
BK	Brightkite	Undirected	Unweighted	58228	214078	7.35	11.1%	18

The results are shown in figures 6, 7, 8, 9, 10 and 11. We see that the computation time does not scale linearly and that the relative performance for both graphs appears not to change ranking often. However, there are multiple points where two algorithm implementations do transition.

For the `DBpedia` graph the fully connected topology, or "Full", experiences multiple such transition points. It does so more often than the star topology. On the `actor-collaboration` graph, the full topology shows heavy performance drops. Such drops can be seen around scale 1.5 in figure 6 and scale 1.7 in figure 7. Therefore we are reluctant to use the fully connected topology in further experiments.

In the `DBpedia` graph, implementations can be seen to compute faster for scale 4.0 when compared to smaller scales. We observe that this behaviour is almost absent for the star topology with random bridges in figure 9 and figure 7 for $n = 1$ and $n = 5$, respectively. This is also true for the fully connected topology in figure 11.

With the `actor-collaboration` graph, we observe that $n = 1$ in figure 6 shows more points of transition than $n = 5$ and $n = 10$ in figures 7 and 8, respectively. Especially the star with random bridging topology for $n = 5$ shows less transition points. Similarly $n = 5$ appears to have the least transition points for the `DBpedia` graph.

We also see that, for the `actor-collaboration` graph, the vertex-push implementation experiences the highest number of transition points. Comparable to that, the vertex-push

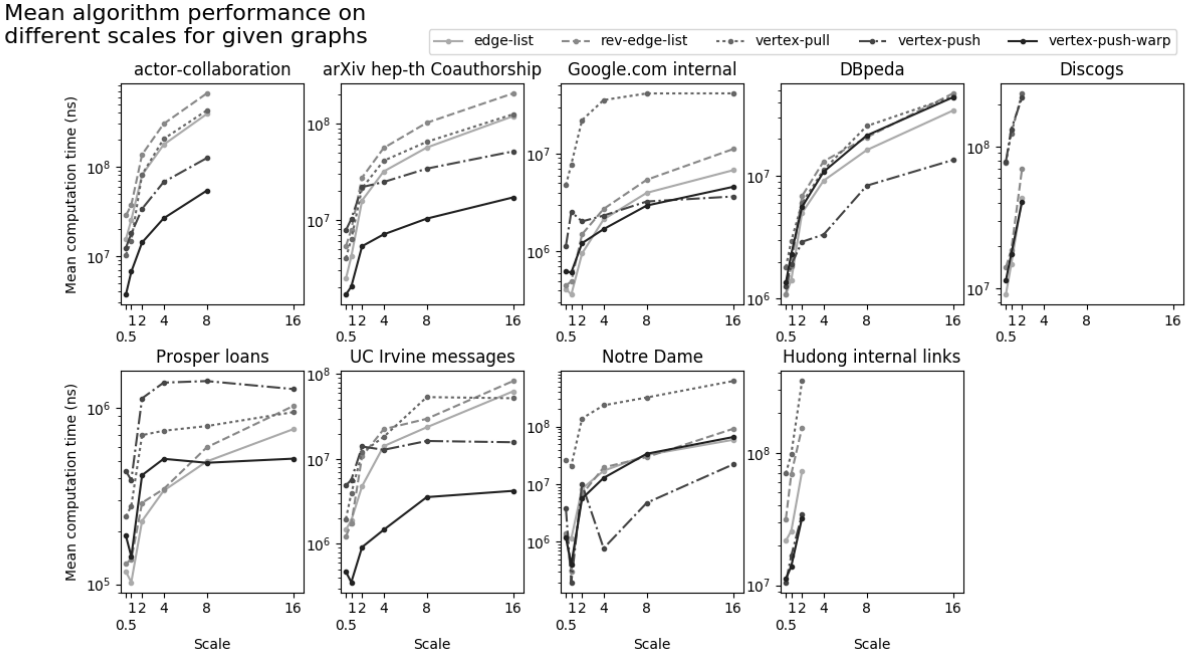


Fig. 3: A comparison of the relative algorithm performance under scaling for different graphs. Each experiment is repeated 5 times for 20 different root vertices. Vertical axis range zoomed to best fit the results. Vertical axis range from 10^0 shown in appendix A figure 12.

implementation switches ranking for all experiments for the DBpedia graph, except for the ring with random bridging topology in figure 11. Because these experiments solely focus on different scaling parameters, these observations are investigated in section V-D.

There appears to be no clear pattern in the differences between different scaling parameters from these experiments. Because the star topology with a random bridge and 5 interconnections shows the least number of transition points and appears most stable in terms of scalability, we chose these parameters for further experiments.

C. Algorithm comparison for different graphs

Figure 3 shows how the performance of different implementations scales for different graphs. We see that performance-based ranking is preserved over scales. However, there are a lot of points where the ranking switches.

The actor-collaboration graph only allowed to be processed up to and including scale 8. Similarly the Discogs and Hudong graphs could not be processed above scale 2. We observe that these three graphs are the largest we have taken from the repository. Therefore the inability to benchmark these graphs is likely size related. The GPU is not able to process graphs of such size.

We also observe that the performance difference between algorithm implementations for different graphs can be as high as an order of magnitude. This highlights the relevance of being able to employ the best implementation. The actor-collaboration and arXiv hep-th Coauthorship graphs show a ranking similar to each other. The graphs are similar: The cluster coefficients are 16.6% and 16.9%, the diameters 13 and 9 and the average degrees 173.28 and 233.38. Both graphs are undirected and unweighted. We expect that the similarity in ranking is due to these topological details.

Besides the ranking, the scalability of the algorithms is also similar for the actor-collaboration and arXiv hep-th graphs. The vertex-push implementation, in both graphs, starts in a worse ranking and, for larger scales, is only outperformed by the vertex-push-warp implementation. This phenomena occurs at a smaller scale for the actor-collaboration graph. This likely happens at a smaller scale because the actor-collaboration is the larger of the two and as such does not need to be scaled up as much before that effect to occur.

We also observe a difference in the number of transition points per graph. For example, the prosper loans and UC Irvine messages graphs experience more such transition points than others. The actor-collaboration, arXiv hep-th and Google.com graphs experience less of such transition points. Because this is different per graph and similar graphs, like the actor-collaboration and arXiv hep-th graphs, experience a similar number of transition points, we conclude that this behaviour has to do with the properties of the graph itself.

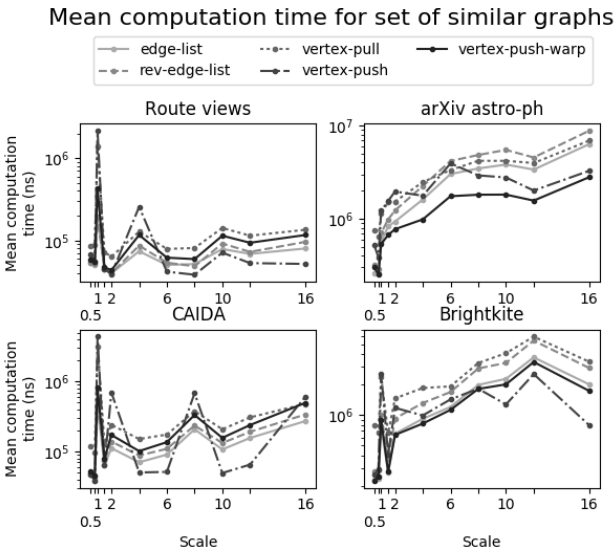


Fig. 4: Mean performance of algorithm implementations on topologically similar graphs over 20 different root vertices and 5 runs per root vertex. Vertical axis range zoomed to best fit the results. Vertical axis range from 10^0 shown in appendix A figure 13

.

Another recurrent phenomenon we observe is the better scalability of the vertex-push implementation. This basically means that the vertex-push implementation switches ranking or starts to relatively perform better. In figure 3 this happens for the actor-collaboration graph, arXiv hep-th, Google, DBpedia, UC Irvine and Notre Dame graphs. Only the Prosper loans graph definitely shows no such behaviour. This raises the question, does vertex-push scale better than other algorithms? If so, can a transition point be determined? We investigate this further in section V-D.

D. The transition point of vertex-push

We ran further experiments in order to investigate the better scalability of the vertex-push implementation. These experiments include graphs that are topologically similar or, in other words, have a similar underlying structure and attempt to find out whether such transition points lie close together for similar graphs. The results are shown in figure 4.

The route views and CAIDA graphs show that the vertex-push implementation heavily switches ranking at some scales. It is unclear to us why exactly this happens. The scalability of vertex-push appears to be unstable.

The route views and CAIDA graphs are slightly smaller and have less edges per vertex, whereas the arXiv astro-ph and Brightkite graphs are larger and have more edges per vertex. All graphs are undirected and unweighted. The arXiv astro-ph graph has an average degree of 21, whereas the other graphs have an average degree of around 5.

We can see that the route views and CAIDA graphs show no lasting transition points for the vertex-push implementation. Instead, the implementation is most efficient from the start, except on 2 scales for the route views graph and 4 scales for the CAIDA graph from the 11 scales for each graph in total. The opposite is true for the arXiv astro-ph and Brightkite graphs, where vertex-push only starts to outperform other implementations on larger scales.

While similar in size, in terms of vertices, the arXiv astro-ph and Brightkite graphs have several times more edges when compared to the other two graphs. We expect that this has an effect on the relative performance of the algorithms and on the better scalability of vertex-push. We also observe that the Brightkite graph shows better scalability than the arXiv astro-ph graph. The ranking switches earlier for the Brightkite graph, despite the Brightkite graph having more edges and vertices. The arXiv astro-ph graph has more edges per vertex. We expect that the amount of edges per vertex plays a role in the scalability of vertex-push.

We also observe that the route views and CAIDA graphs show a similar ranking of the implementations. The same can be said for the arXiv astro-ph and Brightkite graphs. We expect that this is because of the topological similarities in the graphs. This confirms that the relative performance of the BFS implementations is indeed dependent on graph properties.

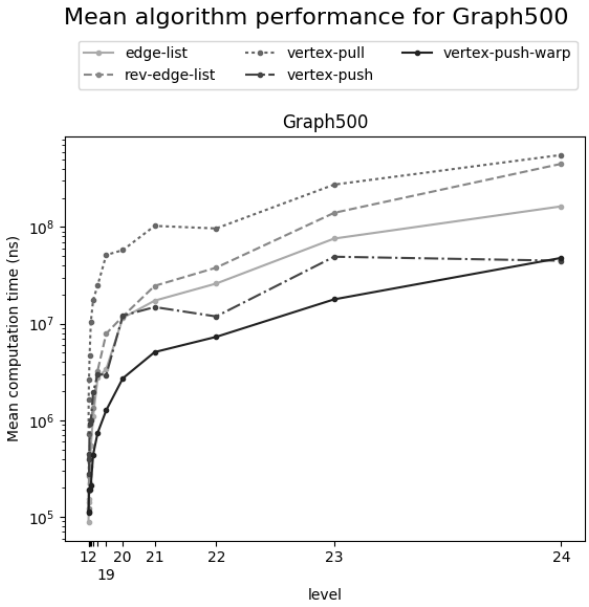


Fig. 5: Mean performance of algorithm implementations on the Graph500 graphs over 20 different root vertices and 5 runs per root vertex. Vertical axis range zoomed to best fit the results. Vertical axis range from 10^0 in figure 20 and both axes zoomed to highlight lower levels in figure 21 shown in the appendix A

E. Comparison to Graph500

The results of the benchmarks for the Graph500 graphs are shown in figure 5. We observe that the experiments on the Graph500 graphs show the same pattern of preservation of performance-based ranking over scale. The differences in computation time between the algorithms are, similarly to earlier experiments, multiples of each other.

Except for the vertex-push implementation, no other implementations have any transition points between them. On level 12, the vertex-push implementation starts as second to last. On level 24 however, vertex-push is fastest. This further confirms the better scalability of the vertex-push implementation.

Besides that, we observe that the scalability of the algorithm implementations appears smooth. There are no sudden drops in performance. This is similar to the actor-collaboration and arXiv hep-th Coauthorship graphs, but different to, for example, the Prosper loans and Notre Dame graphs in figure 3.

VI. Discussion

We show that the relative performance of BFS implementations is stable under scaling, not showing many transition points. However, the relative performance is not fully scale-invariant. Within a few multiples of scale, we expect the relative performance to stay unchanged. When a graph is scaled to multiple times its original size, one implementation can start to outperform another. This is similar to the results found in earlier research mentioned in section III-A that compares graph isomorphism algorithms.

The set of graphs used in this research however, is quite topologically diverse and limited by what KONECT has to offer.

We also show that tuning the scaling parameters has little effect on the scalability of algorithm implementation performance. However, these experiments are limited to 2 graphs and a single scaling mechanism.

We also note that the vertex-push implementation shows better scalability than others. Further experiments confirm this and show that the better scalability is indeed dependent on topology. The experiments also show that the transition point occurs around the same size for graphs that are topologically similar. This transition point also correlates with the number of edges per vertex.

Performing the same benchmarks on different levels of the Graph500 graphs shows the same general increase in computation time. We observe that only the vertex-push implementation switches in performance-based ranking.

The Graph500 results are similar to that of the scaled real-world graphs. However, the Graph500 levels do not show sudden drops and increases in performance that the scaled real-world graphs do. These anomalies might be caused by improper scaling of the graphs. If the scaled graph is topologically similar to the original, one would not expect such sudden performance drops or increases

Future Work

This work has focused on BFS, but there are many other graph algorithms where the same problem of relating implementation performance to topological properties matters. Future research into the scale-invariance of other graph algorithms might find different classifications.

Merrill *et al.* describe multi-GPU graph traversal [9]. It would be interesting to investigate the scale-invariance of such a mechanism when comparing algorithm implementations.

We have only used 5 BFS implementations. More implementations and variants exist. Naturally, the results should be extended if these are used in future research.

Instead of comparing diverse graphs, the relative performance under scaling for similar graphs should be examined. We briefly looked into this in section V-D. However the experiments are limited to several graphs of only one class. The field of graph generation might aid in creating repositories of similar graphs. These repositories would be constrained, as explained in section III-B.

We have shown that the transition points for topologically similar graphs occurs around the same size. Is it possible to determine, in detail, where transition points may occur for different graphs? If this is possible, one might be able to predict when one algorithm starts to outperform another.

References

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[2] J. Leskovec, "Stanford Network Analysis Platform (SNAP)," *Stanford University*, 2006.

[3] Kunegis, Jérôme, "Konect: the koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*, ACM, 2013, 1343–1350.

[4] A. Musaafir, "Shrinking and Expanding Graph Datasets," Master's thesis, University of Amsterdam, the Netherlands, 2017.

[5] Ying, Xiaowei and Wu, Xintao, "Graph generation with prescribed feature constraints," in *Proceedings of the 2009 SIAM International Conference on Data Mining*, SIAM, 2009, 966–977.

[6] Miller, Joel C and Hagberg, Aric, "Efficient generation of networks with given expected degrees," in *International Workshop on Algorithms and Models for the Web-Graph*, Springer, 2011, 115–126.

[7] M. Verstraaten, A. L. Varbanescu, and C. de Laat, "Synthetic graph generation for systematic exploration of graph structural properties," in *European Conference on Parallel Processing*, Springer, 2016, 557–570.

[8] ——, "Using Graph Properties to Speed-up GPU-based Graph Traversal: A Model-driven Approach," *arXiv preprint arXiv:1708.01159*, 2017.

[9] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN Notices*, ACM, vol. 47, 2012, pp. 117–128.

[10] D. Li and M. Becchi, "Deploying graph algorithms on gpus: An adaptive solution," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, IEEE, 2013, pp. 1013–1024.

[11] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *ACM SIGPLAN Notices*, ACM, vol. 46, 2011, pp. 267–276.

[12] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *International conference on high-performance computing*, Springer, 2007, pp. 197–208.

[13] P. Foggia, C. Sansone, and M. Vento, "A performance comparison of five algorithms for graph isomorphism," in *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001, pp. 188–199.

[14] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs," in *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, 2001, pp. 149–159.

[15] V. Carletti, P. Foggia, and M. Vento, "Performance comparison of five exact graph matching algorithms on biological databases," in *International Conference on Image Analysis and Processing*, Springer, 2013, pp. 409–417.

[16] S. Voss and J. Subhlok, "Performance of general graph isomorphism algorithms," PhD thesis, Citeseer, 2009.

[17] P. Harish, V. Vineet, and P. Narayanan, "Large graph algorithms for massively multithreaded architectures," *International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74*, 2009.

[18] R. Milo, N. Kashtan, S. Itzkovitz, M. E. Newman, and U. Alon, "On the uniform generation of random graphs with prescribed degree sequences," *arXiv preprint cond-mat/0312028*, 2003.

[19] W. Guo and S. B. Kraines, "A random network generator with finely tunable clustering coefficient for small-world social networks," in *Computational Aspects of Social Networks, 2009. CASON'09. International Conference on*, IEEE, 2009, pp. 10–17.

[20] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray User's Group (CUG)*, vol. 19, pp. 45–74, 2010.

[21] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, SIAM, 2004, pp. 442–446.

[22] Bal, Henri and Epema, Dick and de Laat, Cees and van Nieuwpoort, Rob and Romein, John and Seinstra, Frank and Snoek, Cees and Wijshoff, Harry, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 5, 54–63, 2016.

[23] DAS-5. (2012). Accelerators and special compute nodes, [Online]. Available: https://www.cs.vu.nl/das5/special.shtml (visited on 07/07/2018).

[24] KONECT. (2017). Networks, [Online]. Available: http://konect.uni-koblenz.de/networks/ (visited on 07/07/2018).

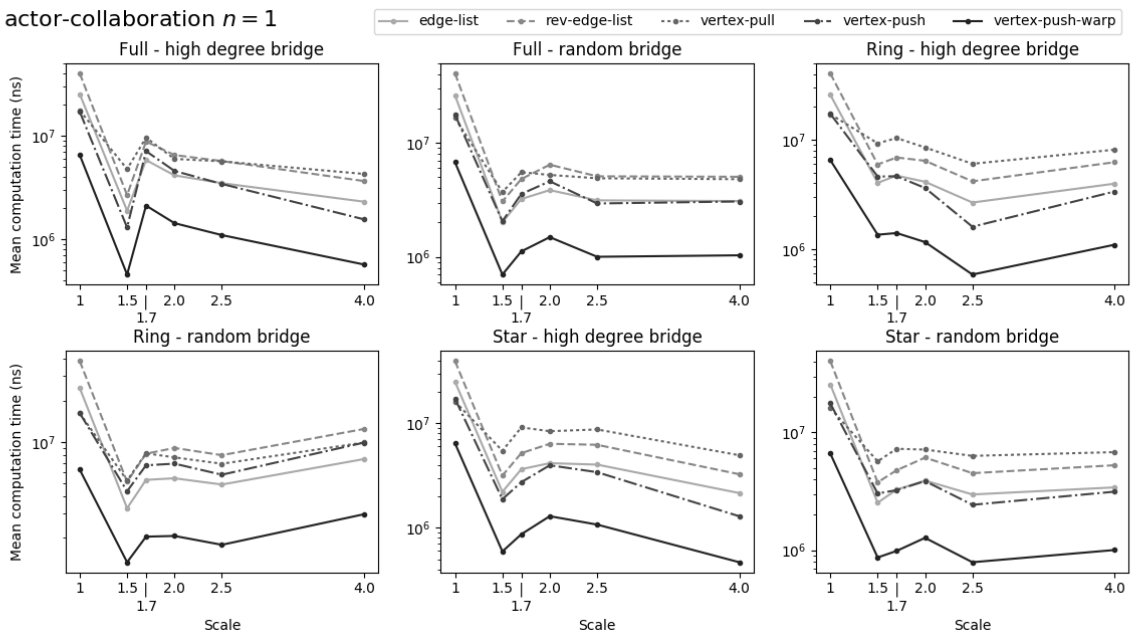


Fig. 6: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `actor-collaboration` graph. Number of interconnections is 1. Vertical axis range zoomed to best fit the results. Vertical axis range from 10^0 shown in appendix A figure 14.

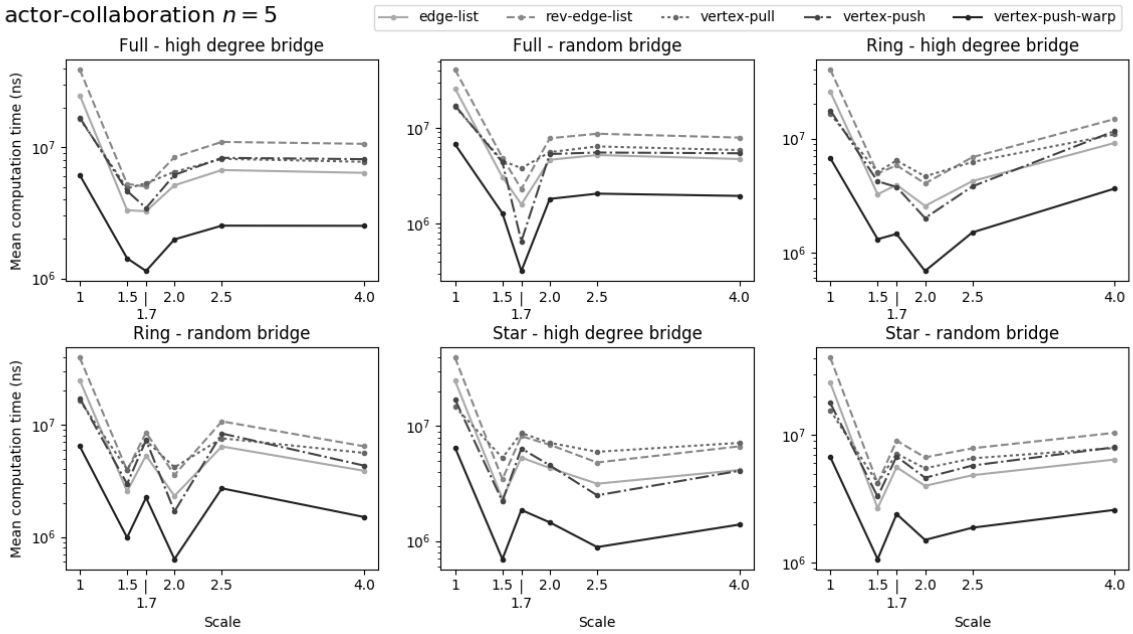


Fig. 7: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `actor-collaboration` graph. Number of interconnections is 5. Vertical axis range zoomed to best fit the results. Vertical axis range from 10^0 shown in appendix A figure 15.

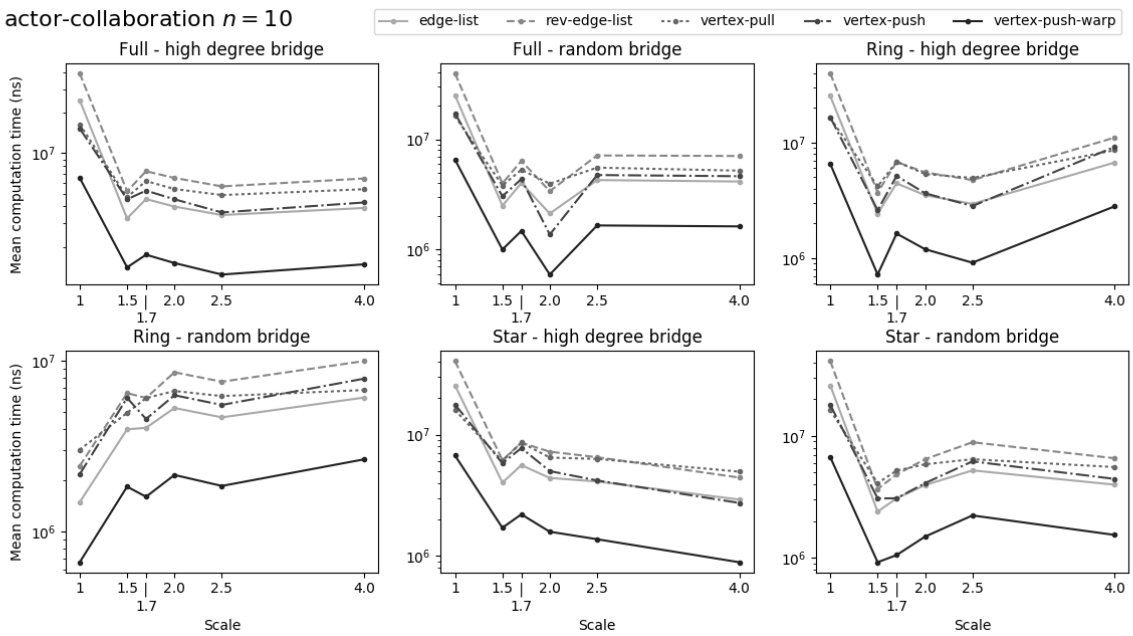


Fig. 8: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `actor-collaboration` graph. Number of interconnections is 10. Vertical axis range zoomed to best fit the results. Vertical axis range from 10^0 shown in appendix A figure 16.

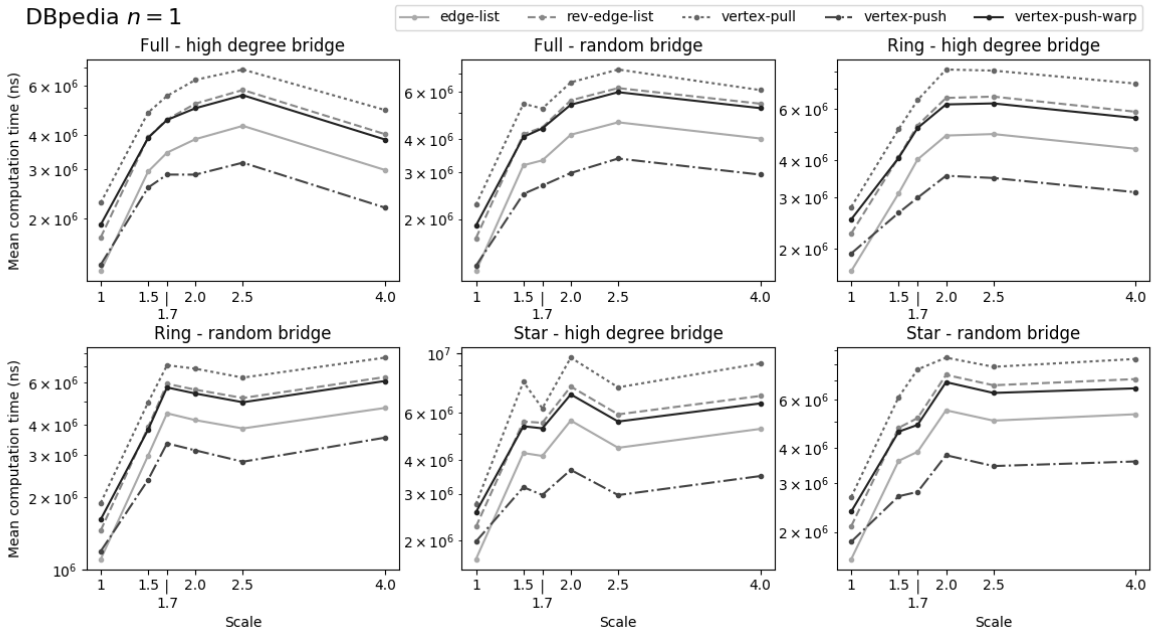


Fig. 9: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `DBpedia` graph. Number of interconnections is 1. Vertical axis range zoomed to best fit the results. Vertical axis range from 10^0 shown in appendix A figure 17.

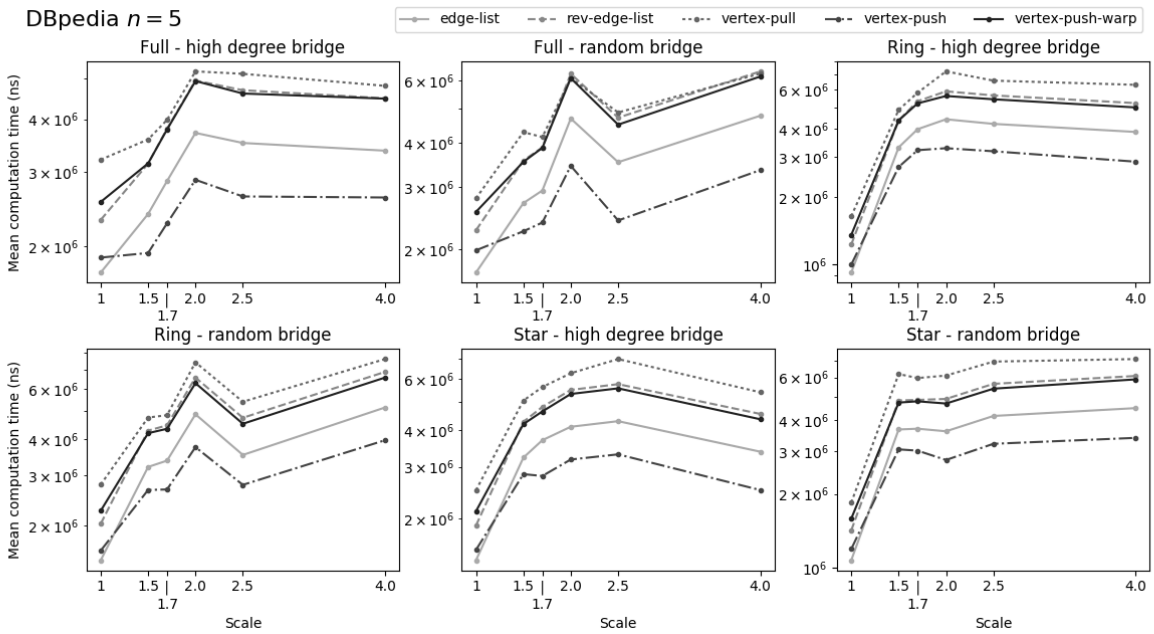


Fig. 10: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `DBpedia` graph. Number of interconnections is 5. Vertical axis range zoomed to best fit the results. Vertical axis range from 10^0 shown in appendix A figure 18.

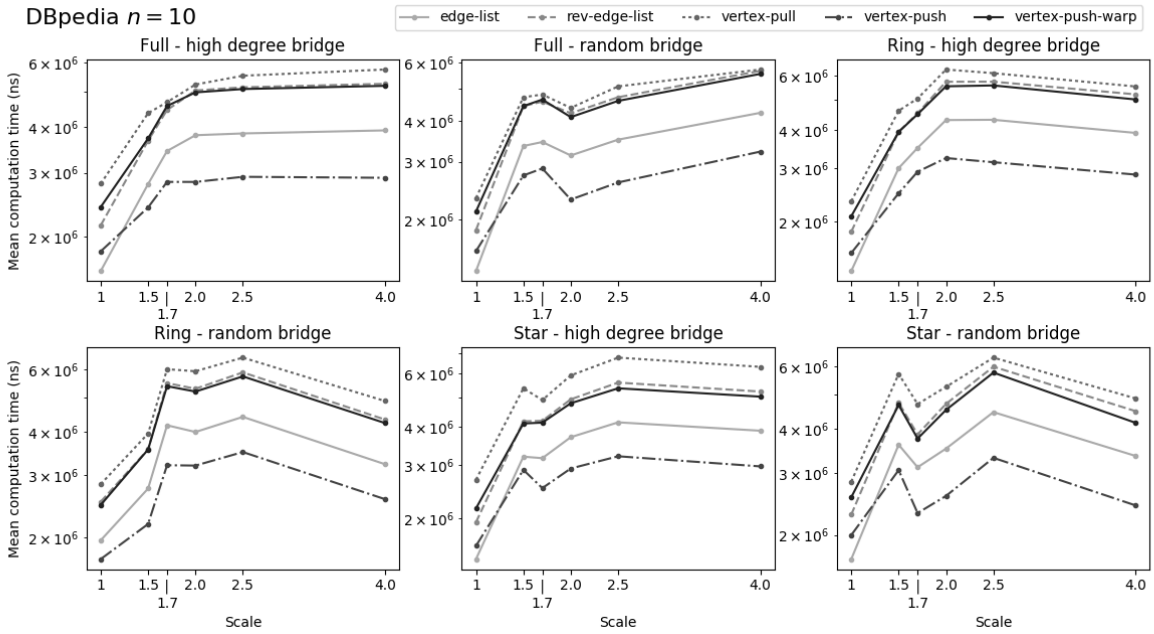


Fig. 11: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `DBpedia` graph. Number of interconnections is 10. Vertical axis range zoomed to best fit the results. Vertical axis range from 10^0 shown in appendix A figure 19.

APPENDIX A
PLOTS WITH VERTICAL AXIS RANGE STARTING FROM 10^0

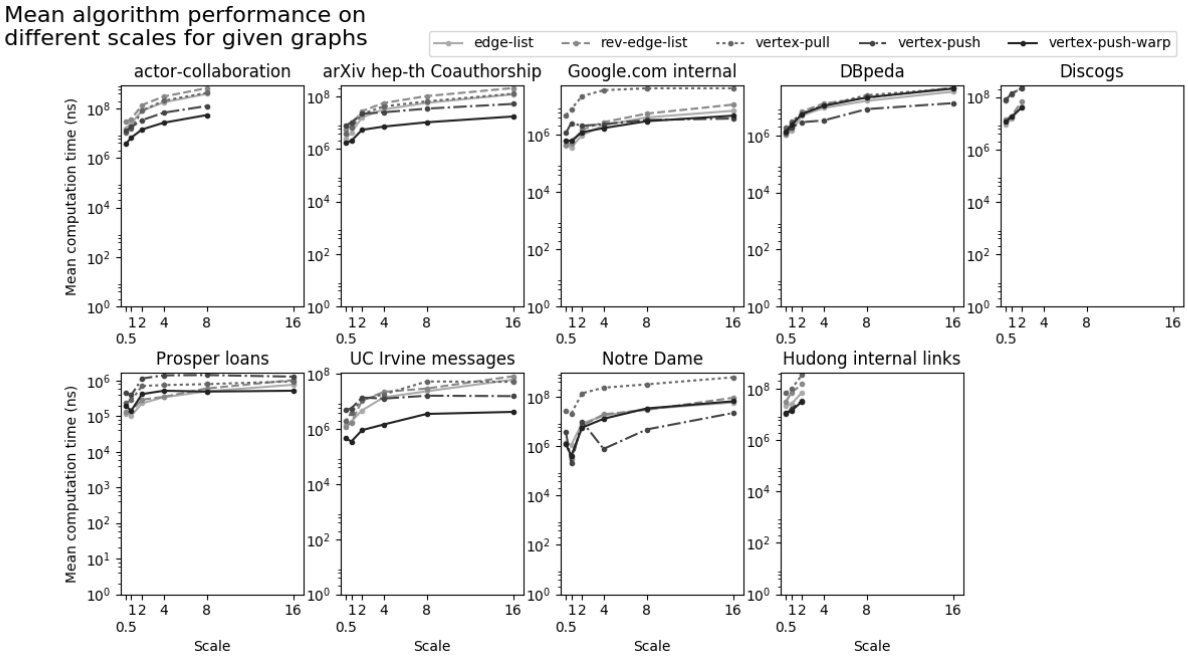


Fig. 12: A comparison of the relative algorithm performance under scaling for different graphs. Each experiment is repeated 5 times for 20 different root vertices.

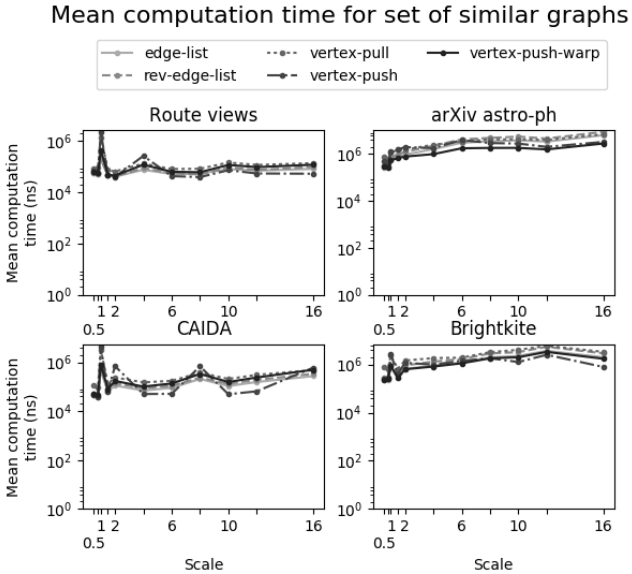


Fig. 13: A comparison of the relative algorithm performance under scaling for different graphs. Each experiment is repeated 5 times for 20 different root vertices.

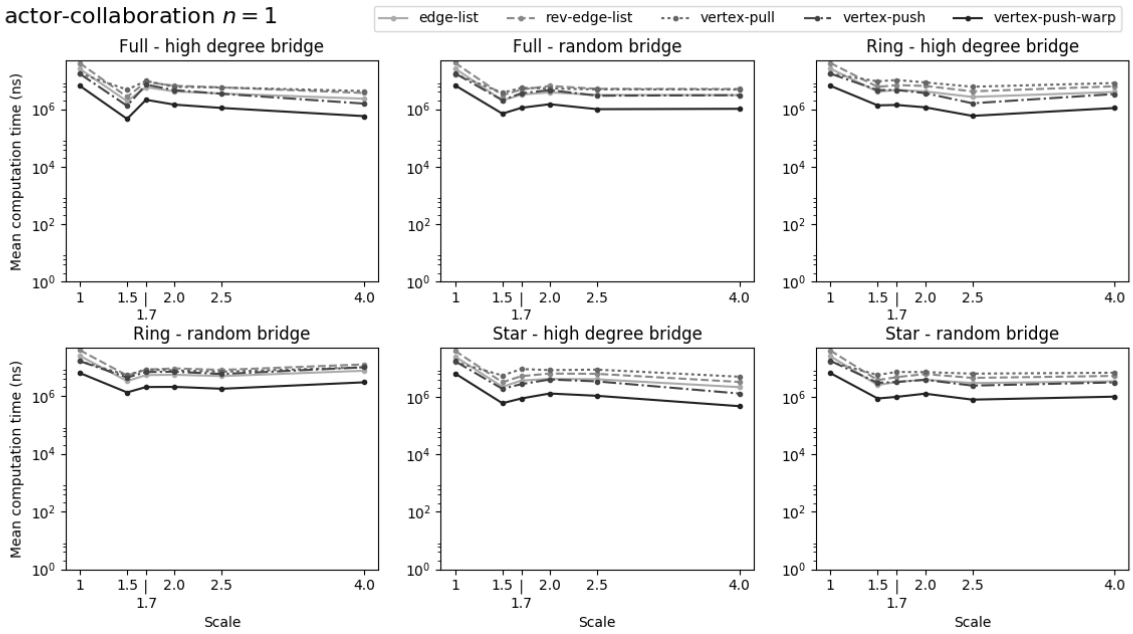


Fig. 14: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `actor-collaboration` graph. Number of interconnections is 1.

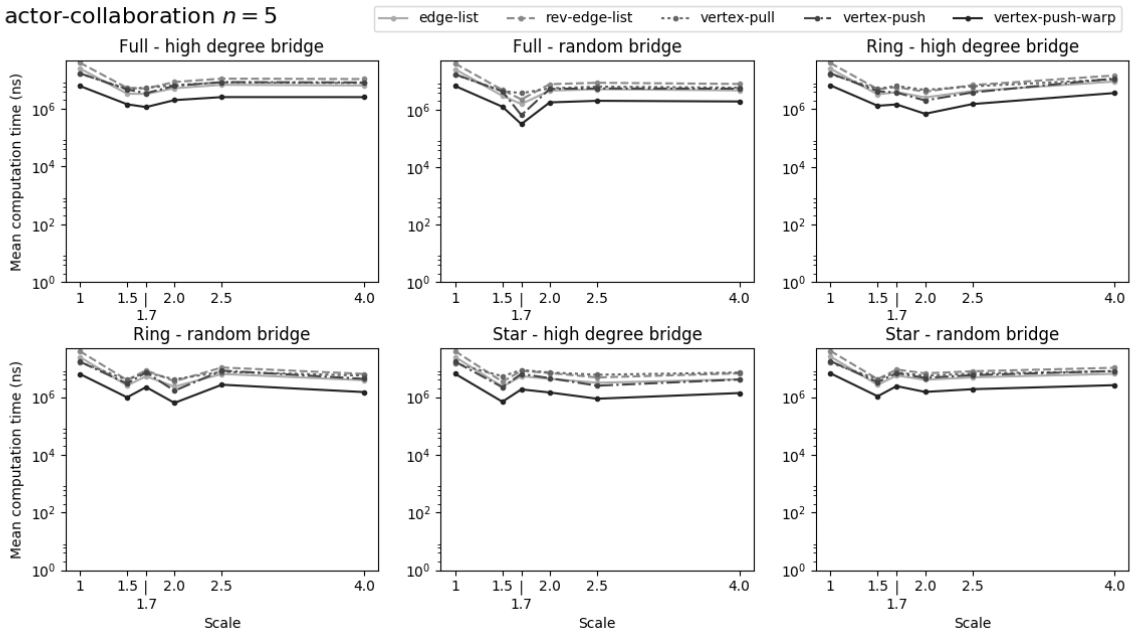


Fig. 15: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `actor-collaboration` graph. Number of interconnections is 5.

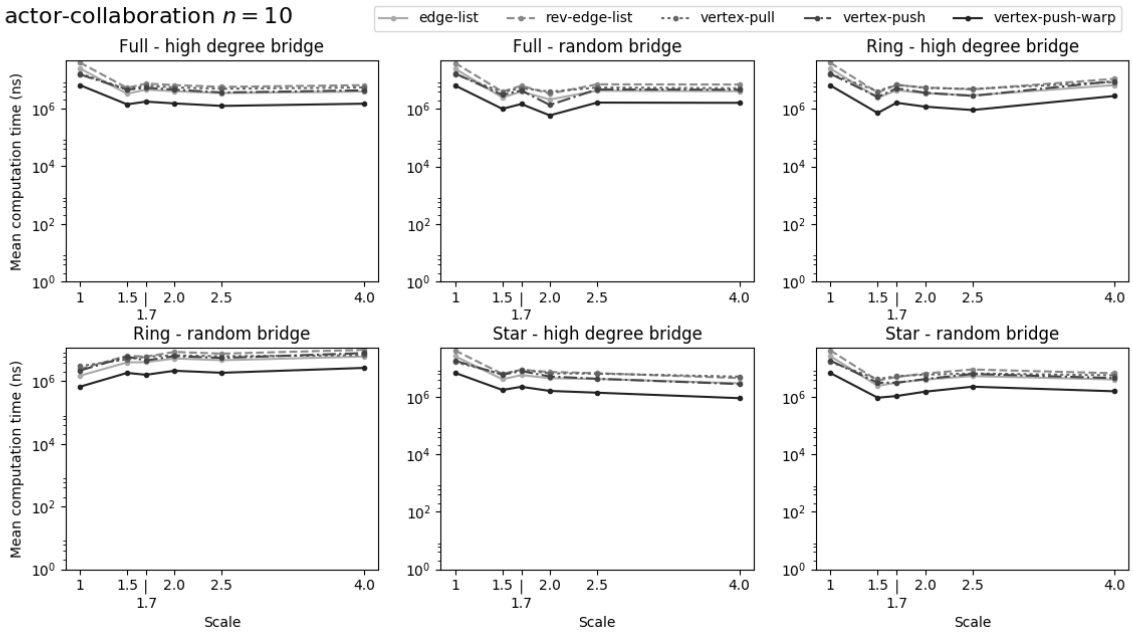


Fig. 16: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `actor-collaboration` graph. Number of interconnections is 10.

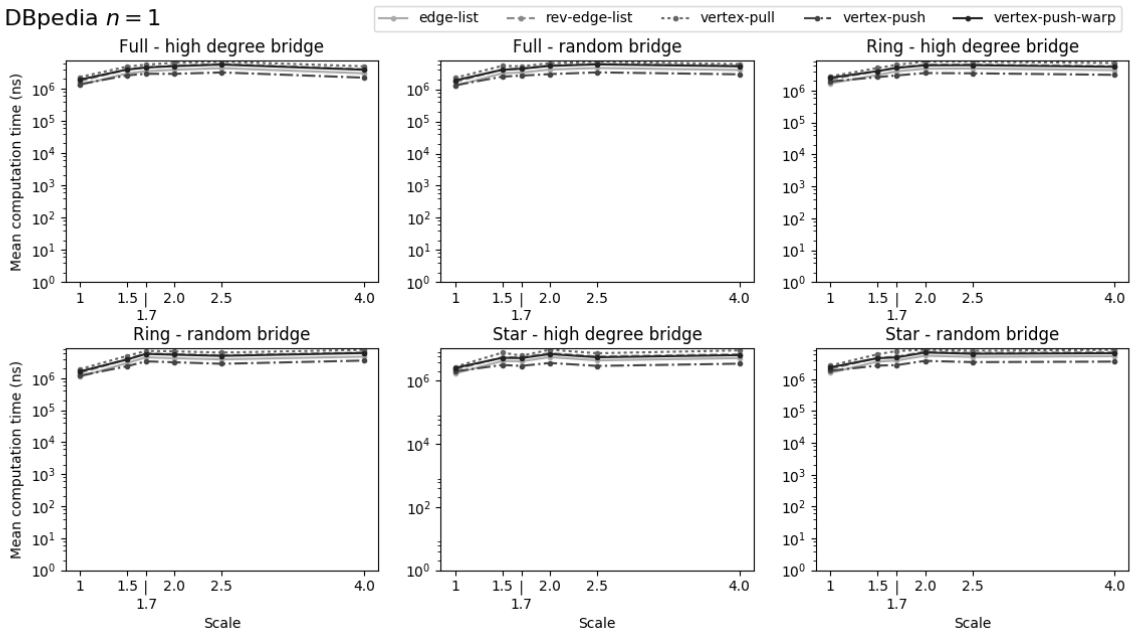


Fig. 17: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `DBpedia` graph. Number of interconnections is 1.

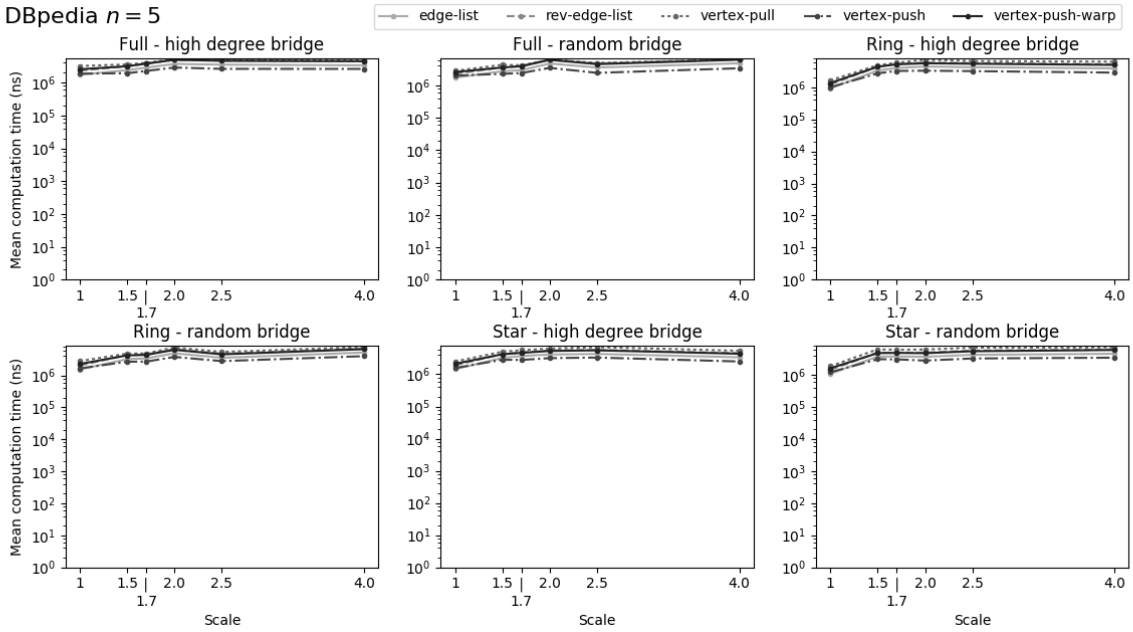


Fig. 18: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `DBpedia` graph. Number of interconnections is 5.

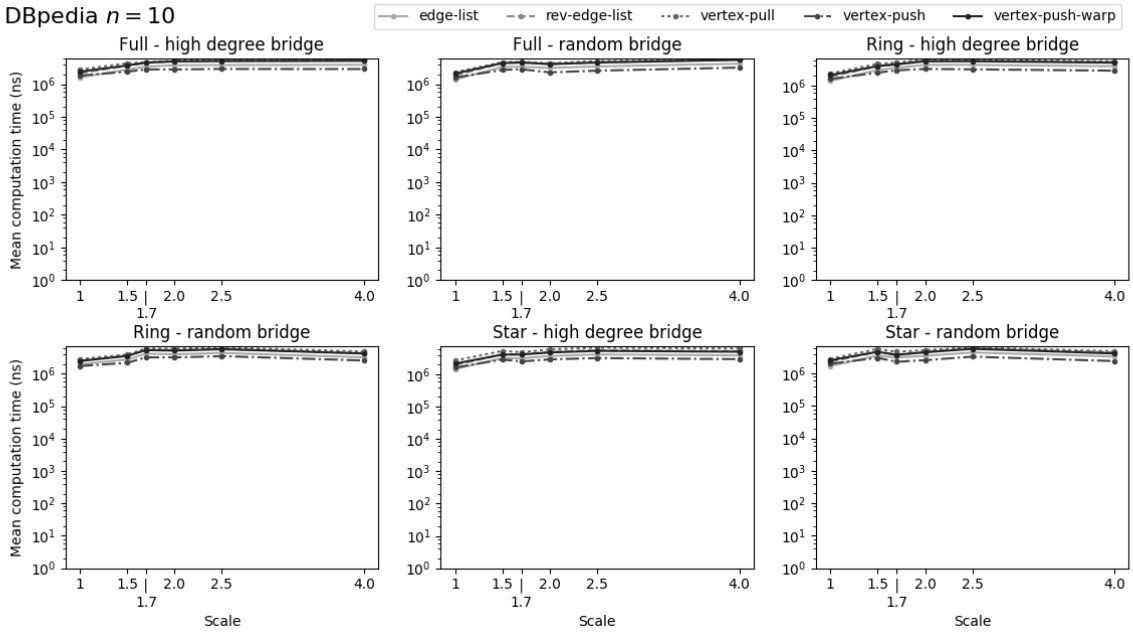


Fig. 19: Comparison of mean algorithm computation time over 20 different root vertices for differing topologies on the `DBpedia` graph. Number of interconnections is 10.

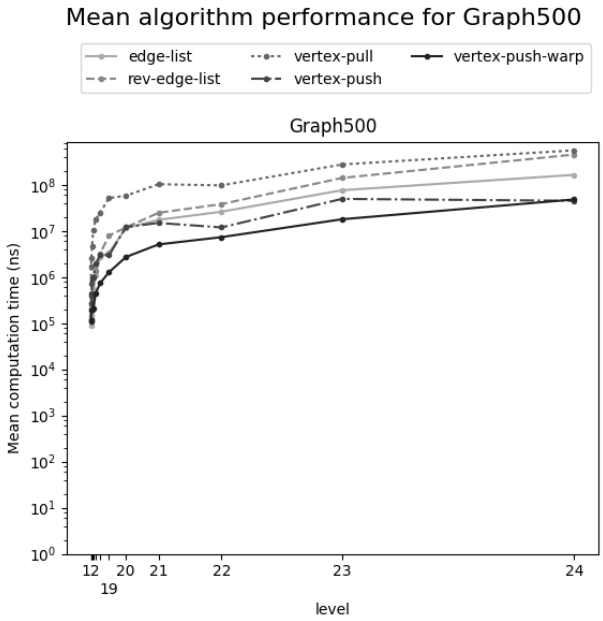


Fig. 20: Mean performance of algorithm implementations on the Graph500 graphs.

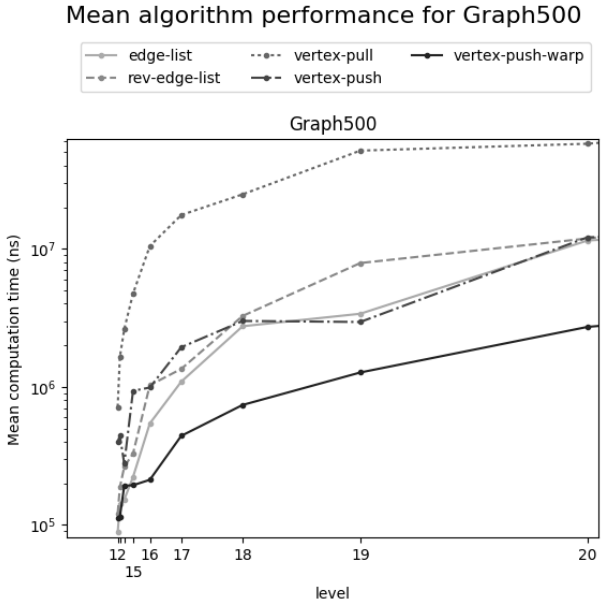


Fig. 21: Mean performance of algorithm implementations on the Graph500 graphs. Vertical and horizontal axes range zoomed to highlight the lower levels.