Universiteit Leiden
The Netherlands

Opleiding Informatica

A Comparison of Breadth-First Search Implementations for Real-World Networks on a Modern GPU

Cem Sevingil

Supervisors: Dr. F.W. Takes & Dr. K.F.D. Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

22/06/2021

Abstract

The Breadth-First Search (BFS) algorithm is often considered as the foundation for traversing and processing graph data structures, such as in network analysis. There are many different BFS implementations which are able to exploit the parallel architecture of a graphics processing unit (GPU). In this thesis, we consider three state-of-the-art BFS implementations in which each implementation is able to exploit the parallel architecture of a modern GPU in a different manner. We consider a range of real-world networks to investigate two research questions: *which of the three state-of-the-art BFS implementations has the best overall performance* and *do particular characteristic network properties of real-world networks have an impact on the overall performance of the BFS implementation*? Our evaluation on a modern GPU reveals that one of the three state-of-the-art BFS implementations has a tipping point in terms of time taken to complete one BFS (running time) and Traversed Edges Per Second (TEPS) when the diameter of the networks that were used in this thesis surpasses a certain threshold. This allows us to predict to a certain extent, based on network properties, which BFS algorithm we should use to get the best overall performance.

Contents

1 Introduction

Graphs are widely used in computer science, mathematics and network analysis to model connectivity, and to understand or solve certain problems, such as path and matching problems. When graphs represent real-world data [19] or a real system, such as connections between users in social media, they are often called (real-world) networks. Our work focuses on real-world networks in which we mainly consider social networks. With the processing or analysing of social networks, we are able to gather more information about relationships amongst groups of people in the real world [23]. Moreover, it allows us to discover new relationships which may help to understand certain patterns in a social network. Graph traversal algorithms, such as Breadth-First Search (BFS), plays an important role in gathering such information.

In the last decade, there has been a lot of development in the usage of Graphics Processing Unit (GPU). Nowadays, GPUs are widely used for their computational power. The unique architecture of a GPU allows it to perform the same instruction on multiple data, also called Single Instruction, Multiple Data (SIMD), which is very pleasing for parallel computing. BFS has the potential to benefit from the SIMD architecture when taking into account how BFS actually works, which is explained in Section 2.4. Graph traversal algorithms are often very computationally expensive (i.e., it takes a great amount of time to complete it). Nowadays, real-world networks can contain up to a few hundred million nodes, which for large inputs may take a considerable amount of time to completely analyse. An increase in overall performance of BFS, for example by exploiting the highly parallel architecture of a GPU, means that we can process or analyse graph data structures, especially very large ones, much faster. To analyse a network completely it is often required to run BFS more than once. For example, to compute a ranking of nodes based on a node centrality measure such as closeness centrality, a BFS has to run n times for a network that has n nodes.

In this thesis, we study three state-of-the-art BFS implementations for the GPU [14, 16, 18]. The goal is to investigate the following two research questions.

1. *Which of the three state-of-the-art BFS implementations has the best overall performance?*
2. *Do particular characteristic network properties of real-world networks have an impact on the overall performance of the BFS implementation?*

Our work approaches these two problems by presenting a systematic comparison between the three state-of-the-art BFS implementations on empirical networks and artificially generated networks. Furthermore, we manipulate particular characteristic real-world network properties, such as global clustering coefficient or the number of triangles in a network, to observe if it has an effect on the overall performance of the BFS implementation. BFS performance was measured by observing and comparing *the running time* of one BFS and the number of *Traversed Edges Per Second* (TEPS) of the BFS implementations on these networks. Finally, the NVIDIA System Management Interface (`nvidia-smi`) [6] was used to monitor the GPU its activity and memory usage. The hypothesis is that the performance of BFS tends to be dependent on the network it has to traverse. Adjusting these characteristic real-world network properties potentially allows us to understand the performance of the three state-of-the-art BFS implementations.

The remainder of this thesis is organised as follows; Section 2 contains the background information that is necessary to understand this thesis; Section 3 describes previous work that are related to the subject of the thesis. Section 4 explains how the three state-of-the-art BFS implementations work. Section 5 describes the experiments and their outcome; Section 6 discusses the limitations of the thesis; Section 7 concludes this thesis.

2 Background

This section contains the background information to understand this thesis; Section 2.1 describes networks; Section 2.2 explores the differences between real-world networks and random networks. Section 2.3 explains network generators, and the benchmarks of generating and acquiring certain network properties; Section 2.4 reveals how the BFS algorithm works. Section 2.5 discusses parallel programming on a GPU.

2.1 Networks

A network consists of nodes and edges. Networks can either be *undirected*, the edges indicate a two-way relationship (the edges are bidirectional), or *directed*, the edges indicate a one-way relationship (the edges are directional). In this thesis, only undirected networks were used. A path is a sequence of edges which connects a sequence of nodes. The shortest path is the path in which a sequence of nodes is connected by the smallest number of edges. The distance between two nodes in a network is equal to the number of edges in the shortest path that connects them. The degree of a node in a network is the number of edges that are connected to it. Regular networks are networks in which the degree of every node is the same. The size of a network is often measured in terms of the number of nodes and edges. Networks can have clusters, which are groups of densely connected nodes. A network is said to be complete if every node is connected to all the other nodes. Finally, networks that are dense refer to the fact that there are relatively many connections between a set of nodes.

2.2 Real-World Networks vs. Random Networks

To really distinguish a network from another network one has to take a look at its characteristics. These network properties describe a network in a more detailed manner. There are many network properties. In this thesis, we consider four important network properties:

- *Diameter (d)*: maximum distance between any two nodes.
- *Global clustering coefficient (c)*: the level of clustering in a network, also called transitivity. The global clustering coefficient can be calculated by dividing the number of actually present triangles by the number of triplets [17, 29]. A triplet consists of three nodes that are connected by either two (open triplet) or three (closed triplet) edges. A triangle consists of three triplets. For example, the network (a) in Figure 3 contains 8 triplets and 1 triangle.
- *Network density (n)*: the number of edges that exist in the network divided by the number of possible edges that would exist if the network was complete.

- *Average path length (ℓ)*: the average distance between all node pairs.

Random networks, such as the Erdős-Rényi model [9] have a uniformly random degree distribution, whereas real-world networks are often considered scale-free (and often follow a power-law distribution). This often results in very different network properties. Figure 1 describes the degree distribution of real-world networks and random networks. Real-world networks are known for their high clustering (they have a substantial number of clusters), short average path length (the so-called small-world phenomenon), sparsity (low network density), and often but not always their short diameter. Random networks can be useful for more theoretical research. For example, random networks can be used to answer or prove if there are networks that exist with certain properties.

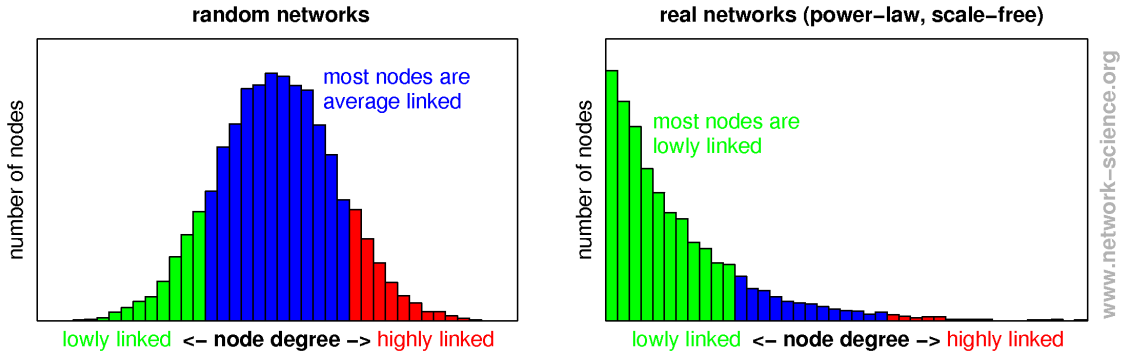
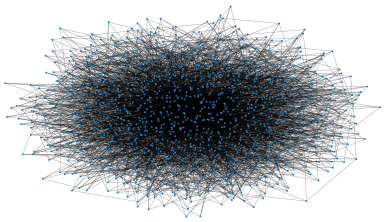


Figure 1: Degree distribution of random networks (left) vs. real-world networks (right) [21].

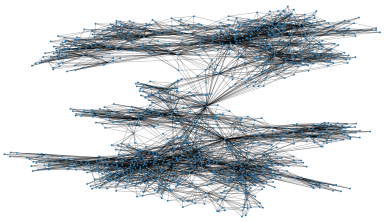
2.3 Network Generators

Random network models can be used to model networks with certain properties. Network generators use these type of network models to generate random networks. We have to take certain network properties, such as the ones mentioned in Section 2.2, into account when we are trying to generate networks that to a certain extent resemble real-world networks. There are many different network generators available that can do this [4]. The Barabási-Albert model [1], which was published in 1999, is a well-known network model which is able to model scale-free networks using preferential attachment. Preferential attachment is a probability mechanism in which a node with a high number of neighbours has a higher chance to get more neighbours attached to it. This is also referred to the 'rich get richer' phenomenon. The network density and diameter of the networks generated with this model are primarily determined by the parameter m, which affects the number of edges to add from a new node to existing nodes while generating a network. Holme and Kim published [13] an improved version of the Barabási-Albert model. They concluded that the Barabási-Albert model was adequate for modelling scale-free networks, but that it failed in describing networks with high clustering. They introduced a new parameter into the Barabási-Albert model which controls the number of triangles in a network and therefore also the clustering coefficient. This parameter (p) regulates the probability of adding a triangle after having added a random edge. In general, a higher p value results in more triangles in a network, and thus a higher global clustering coefficient. In this thesis, we use Holme and Kim's algorithm to generate networks. This allows us to control the number of triangles in a network. Figure 2 shows two networks that are generated with the

Barabási-Albert model and Holme and Kim's algorithm.



(a) Barabási-Albert model [1].



(b) Holme and Kim's algorithm [13].

Figure 2: Example networks that are generated with random network generators [1, 11, 13].

NetworkX [11] is a well-known Python [26] package which is very useful for generating and analysing networks. In this thesis, we use NetworkX to generate and analyse networks. Gathering certain properties of a network, such as the diameter and the average path length, can be very time consuming. Operations that are required to gather these properties are very computationally expensive, because they often require the network to be traversed from every node to all the other nodes. This is especially time consuming when traversing large networks (millions of nodes and edges). Sampling is often used to address this issue. Thus, instead of traversing the network from every node, we traverse a random small sample. This gives us an estimation of the diameter and the average path length instead of the exact value [27].

2.4 Breadth-First Search

BFS is a graph traversal algorithm for traversing graph data structures, such as networks. BFS is widely used to solve many graph problems. BFS is often implemented with one or two data structures: a queue and an array. BFS can be performed in many ways. The most basic variant of BFS uses a *frontier* queue, which contains nodes (frontiers) in which their adjacent nodes (neighbour nodes) need to be visited, and a status array, which keeps track of the nodes that have already been visited. This variant of BFS starts with a selected node v, marks it in the status array and adds it to the frontier queue. Node v is now the only member of the frontier. Thereafter, the adjacent nodes of v are being visited, marked and added to the frontier queue. Node v is removed from the frontier queue if all of its adjacent nodes are visited. The next node in the frontier queue will be the new node v. This whole process will be repeated until all the nodes have been visited or until the frontier queue is empty. BFS can either be done in a top-down (traditional) fashion in which BFS traverses the nodes from top downwards (from frontiers to neighbour nodes), or a bottom-up fashion in which BFS traverses the nodes from bottom upwards (from neighbour nodes to frontiers). There is also a hybrid variant of BFS which makes use of both the top-down and the bottom-up approach. Figure 3 shows an example on how top-down and bottom-up BFS with the use of a frontier queue and a status array works. Previous research [2] has shown that the bottom-up approach tends to traverse more efficiently compared to the top-down approach when the frontier queue contains a substantial number of nodes. This is because, in a top-down approach every node in the frontier

has to traverse all of its neighbours, whereas in a bottom-up approach every unvisited node (a node that is not in the frontier and not visited) is trying to find a parent node which is part of the frontier queue. This often results in checking less edges compared to the top-down approach [2].

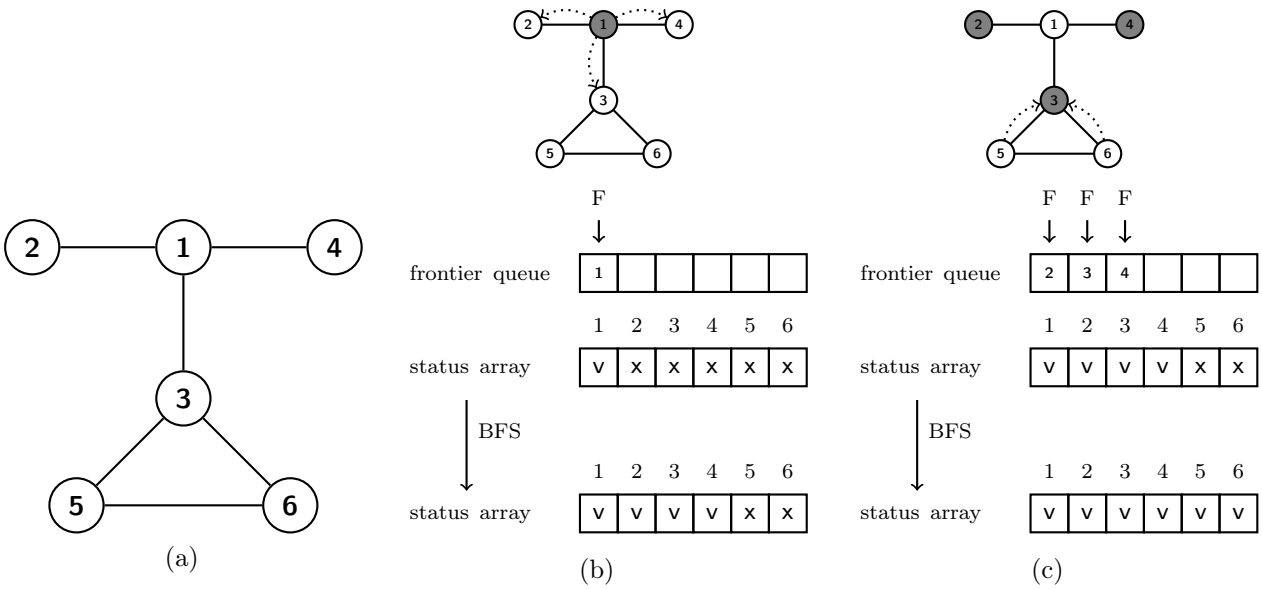


Figure 3: BFS examples on (a) undirected network consisting of 6 nodes and 6 edges with the use of a frontier queue and a status array. (b) Top-down BFS on the example network with node 1 as the active node in the frontier. (c) Bottom-up BFS on the example network with nodes 2, 3, and 4 as the active nodes in the frontier. Moreover, F, V and X indicate a node in the frontier, a visited node, and an unvisited node. In (b) and (c), gray nodes represent the current nodes in the frontier.

2.5 Parallel Programming on a GPU

GPUs are widely used in parallel programming. This is often done with the use of an Application Programming Interface (API), such as Compute Unified Device Architecture (CUDA). CUDA [5] is a well-known API for `C/C++`, which allows processes to be run on a GPU. Processes can contain mathematical operations, such as addition or multiplication. These kind of operations are performed by an Arithmetic Logic Unit (ALU). In parallel programming on a GPU, all threads execute the same process (kernel) in lock-step (at the same time), but on different data. The ID of a thread determines on which data it has to work. Threads are often grouped in blocks (in CUDA this is often referred to as CUDA blocks or CTA), of up to 1024 threads in CUDA, in which each block contains shared memory. Furthermore, a kernel is executed as a grid, which is a group of blocks. Shared memory can be accessed (read or write) by all threads in the same block. Shared memory has a higher memory bandwidth compared to global memory, but is much smaller in size. Memory bandwidth is the rate at which data can be accessed, which is often measured in gigabytes per second (GB/s). Streaming Multiprocessors (SMs) on a GPU schedule threads in groups of 32 called warps. The advantage of warps is that threads within a warp are often required to simultaneously access global memory. Memory accesses to global memory is very expensive. Therefore, with the use of warps it is possible to reduce multiple memory accesses to one memory access. This is also

referred to the coalescing of memory (memory coalescing). Figure 4 shows a simplified schematic view of the execution model in CUDA.

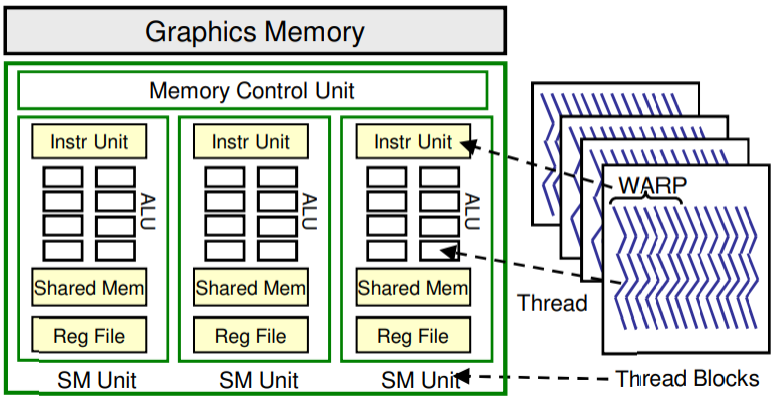


Figure 4: Thread execution model in CUDA [14].

3 Related Work

Over the past decade, there has been a lot of research done on executing BFS on GPUs. Many will consider the Harish and Narayanan BFS implementation [12] as the first one that is able to run on a GPU. This implementation executes BFS on a GPU with CUDA [5] in a top-down manner, which operates in a similar manner as the BFS procedure mentioned in Section 2.4. Throughout the years, the computational power and memory size of GPUs increased drastically. This resulted in GPUs becoming more popular for operations that are able to utilise the parallel architecture of a GPU and for operations which demand a high computational power or a considerable amount of memory. Several works [10, 14, 16, 18, 20, 28] were able to built upon the Harish and Narayanan BFS implementation. Their main goal was to improve the overall performance, such as running time and TEPS, of the BFS implementation. This introduced state-of-the-art BFS implementations with the potential to better utilise GPUs. Prior work [16, 18] has shown an immense increase in BFS performance on a GPU compared to BFS on a CPU. However, many of those GPUs are considered outdated nowadays. To fully utilise a GPU in processing graph data structures one has to take into account the fact that its architecture can have several major issues, such as memory management and workload imbalances [25]. Several papers have been published which address these issues by introducing different optimisations, such as thread classification [16, 20] or per-warp distribution [14]. These optimisations tend to rely on the hardware of the GPU. Prior work [14, 16, 18] has shown that the effect of optimisation techniques tend to fall off after a certain point. This is often caused by the GPU being the bottleneck and is especially noticeable when processing considerably large networks. Our work focuses on the systematic comparison of three state-of-the-art BFS implementations on generated networks and empirical networks to assess, based on certain network properties, which BFS algorithm has the best performance.

4 Methods

This section explores the optimisation techniques of the three state-of-the-art BFS implementations; Section 4.1 presents several common challenges of BFS on a GPU; Section 4.2 explains the optimisation techniques of UIUC BFS [18]. Section 4.3 describes the optimisation techniques of Warp BFS [14]; Section 4.4 reveals the optimisation techniques of Enterprise BFS [16].

4.1 GPU Challenges

To efficiently perform BFS on a GPU one has to take several challenges into account. In this thesis, we focus on two common challenges of running BFS on a GPU. First, GPU threads that need to access (read or write) the same data. In parallel computing it is often the case that several threads simultaneously access the same data (race conditions). The consequences of race conditions is that it often results in presenting the wrong outcome. Thread synchronisation techniques, mechanisms which ensure that only one thread can access a certain resource at a given point in time, can be used to prevent the occurrence of race conditions. In the case of BFS, threads are assigned to frontiers in which every thread has to access the same status array. This may lead to race conditions in the status array, such as writing to the status array or reading from the status array, if thread synchronisation mechanisms are not correctly implemented. However, such mechanisms are often very expensive which may lead to a significant overhead. Second, assigning threads to frontiers (workload imbalance). The degree distribution of real-world networks are often sparse. Therefore, nodes that have a high degree will most certainly need more threads compared to nodes that have a substantial smaller degree. As a result, threads assigned to frontiers with a small degree are expected to finish their task earlier than frontiers with a much higher degree. The incorrect assignment of threads may result in threads being wasted, they are idle threads (do nothing useful). Thereby, the workload imbalance increases. Table 1 gives an overview of the methods that the three BFS implementations use to address these two challenges.

4.2 UIUC BFS

UIUC BFS is a BFS implementation on the GPU which accelerates graph traversal for sparse and near-regular graphs on a GPU. It uses a hierarchical frontier queue management to prevent race conditions. Luo et al. [18] build the hierarchical queue structure from the lower-level queues to the higher-level queues. The hierarchical frontier queue consists of three levels. First, *W-Frontiers*, which are the lowest level queues that can only be accessed by threads from a single warp. Second, *B-Frontiers* that consists of eight W-Frontiers. Third, the *G-Frontier*, the highest level frontier queue that stores the complete new frontier queue for the next iteration. Figure 5 gives a schematic view of the hierarchical frontier queue management. The use of a hierarchical queue structure reduces thread synchronisation, because the queue has been partitioned such that different CUDA threads will not access the same element. However, thread synchronisation is still required at the end of each level. To achieve this, a global barrier synchronisation, a barrier in which threads wait (idle) until all the other threads have reached this point, needs to be implemented. This synchronisation technique inflicts a huge kernel-launch overhead. Every time the frontier queue gets too large, which depends on the number of multiprocessors of the GPU, this global barrier synchronisation has to be launched to correctly implement BFS.

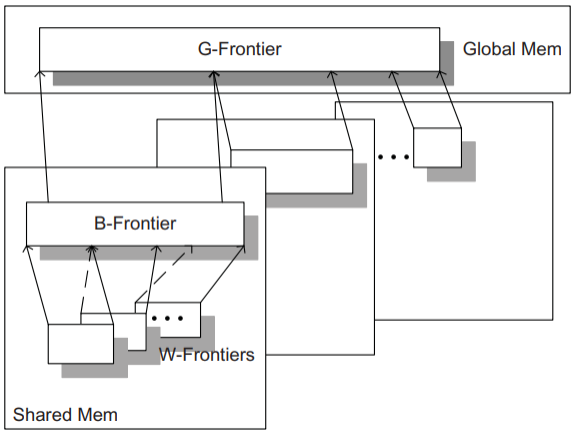


Figure 5: Hierarchical frontiers [18].

4.3 Warp BFS

Warp BFS is a warp-centric based BFS implementation that works on the GPU. This BFS implementation is able to considerably improve the performance of applications with heavily imbalanced workloads or heavy scattering memory accesses. Hong et al. [14] use a virtual warp-based top-down BFS approach of different sizes to execute distinct tasks in serial. The virtual warp-based approach enables threads to access the same shared memory within a warp, which reduces latency and has the potential to increase performance because of it having a higher memory bandwidth compared to global memory. Furthermore, the virtual-warp based approach divides the warp size of 32 in multiple virtual warps of different sizes. Multiple virtual warps are then located in one physical warp in which each virtual warp executes a different task instead of a (physical) warp executing one task. This approach may increase the utilisation of the ALUs and reduce the workload imbalance inside warps. However, the virtual-based approach does bring some difficulties with it. For example, the workload imbalance inside warps are to a greater or lesser extent resolved, but there is still the possibility of workload imbalance existing between warps. Warp BFS addresses this issue by introducing dynamic workload distribution in which each warp dynamically fetches a certain amount of work to process it. Unfortunately, dynamic workload distribution has the potential to cause a considerable amount of overhead.

4.4 Enterprise BFS

Enterprise BFS is a GPU-based BFS implementation that is able to reach very high TEPS, especially when run on large (real-world) networks. This implementation addresses three issues to remove potential bottlenecks. First, Liu et al. [16] believe that constructing an optimal frontier queue is key in gaining better performance. An optimal frontier queue prevents the traversal of duplicate nodes and the need of thread synchronisation. However, constructing optimal frontier queues does carry some overhead with it, because it takes time to identify the frontiers. Second, real-world networks are often very sparse (skewed degree distribution). Enterprise implements dynamic thread scheduling in which the frontiers are classified in four different frontier queues based on their degree. The frontier queues are classified as follows: SmallQueue, consists of frontiers with fewer than 32 edges, MiddleQueue, contains frontiers that have between 32 and 256 edges, LargeQueue, holds

frontiers that have between 256 and 65536 edges, and ExtremeQueue, contains frontiers with more than 65536 edges. Instead of assigning a fixed number of threads to frontiers, Enterprise assigns a certain number of threads to frontiers based on the specific frontier queue. Figure 6 gives a schematic view of the GPU workload balancing. Although dynamic thread scheduling further removes valuable GPU threads being wasted by not having idle threads, there is still some overhead caused by the classification of frontiers. Third, Enterprise uses a one-time switch from top-down BFS to bottom-up BFS to increase its performance even further. As mentioned in Section 2.4, bottom-up BFS is considered more efficient than top-down BFS when the frontier queue contains a substantial number of frontiers. Enterprise uses the ratio of hub nodes (nodes with high clustering) in the frontier queue as an indicator to switch from top-down BFS to bottom-up BFS. Furthermore, Enterprise tries to cache these hub nodes in GPU shared memory during and after direction switching to reduce the overhead of random global memory accesses. However, since GPU shared memory (L1 cache) is often small it is not always possible to cache all the hub nodes.

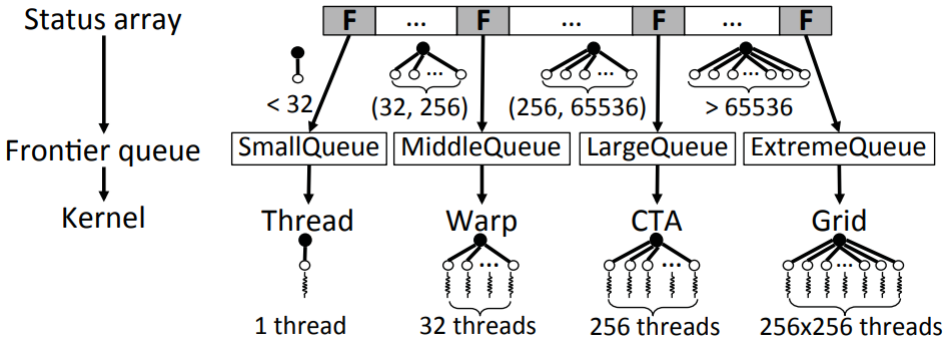


Figure 6: GPU workload balancing [16].

Name	Thread synchronisation	Workload imbalance
UIUC BFS	Only uses thread synchronisation at the highest level (G-Frontier).	Fixed number of threads for frontiers. Therefore, some threads are not being utilised.
WARP BFS	Removes thread synchronisation, because every warp works independently from one another.	Fixed number of threads for frontiers. Therefore, some threads are not being utilised.
Enter-prise BFS	Eliminates the need for thread synchronisation by scanning the status array first before the frontier queue gets generated.	Dynamic thread assignment based on degree of frontiers.

Table 1: Methods used by the three BFS implementations to address the common challenges of BFS on a GPU.

5 Experiments

This section describes the experiments and their outcome; Section 5.1 explains the data that was conducted for the experiments; Section 5.2 describes how the experiments were setup; Section 5.3 discusses the results that were acquired from the experiments. Section 5.4 explores the performance of Enterprise.

5.1 Data Properties

In our experiments, we observe the four network properties that are mentioned in Section 2.2. All the networks that were used in this thesis are represented by two different formats:

- Edge list: stores the data of a network as a list in which every line contains two nodes, a *from* node and a *to* node, which represent an edge.
- Compressed Sparse Row (CSR): stores the data of a network as a sparse matrix.

5.2 Experimental Setup

In this thesis, we use a machine from Leiden Institute of Advanced Computer Science (LIACS) Research and Education Laboratory (REL) Data Science lab that is intended for processes that work on the GPU. This machine contains 16 Intel Xeon E5-2650 2.00Ghz processors with two NVIDIA Geforce RTX 2080 Ti GPUs. This GPU was released late 2018 and is being powered by NVIDIA's Turing GPU-architecture [7]. All the experiments in this thesis were conducted on a single NVIDIA RTX GeForce 2080 Ti. This GPU has a base clock speed of 1350MHz up to 1545MHz, 11 GB (gigabyte) GDDR6 of memory with a memory bandwidth of 616.0 GB/s and 68 streaming multiprocessors (SM), which every SM has a L1 cache of 64kB (kilobyte), with 4352 NVIDIA CUDA cores. Every NVIDIA CUDA core has its own ALU. Moreover, the NVIDIA Geforce RTX 2080 Ti contains a single L2 Cache of 5.5MB (megabyte) which is shared by all SM's. In our work, all source code are compiled with NVIDIA `nvcc` 11.2 and GCC 4.8.5 [24].

The source code that was used in this thesis was acquired from the following GitHub repositories:

- Scaleable HeterOgeneous Computing (SHOC) benchmark suite [8]: benchmark suite which contains a collection of benchmark programs, including two BFS implementations [14, 18].
- Enterprise: version of the source code from the official authors [16].

SHOC uses the METIS format, which is an adaptation of the edge list, whereas Enterprise uses the CSR format. Previous research has shown that the Compressed Sparse Row (CSR) [3] is more efficient than the edge list.

In our work, we use $m \in \{2, 4, 8, 10\}$ and $p \in \{0.1, 0.2, 0.3, ..., 1.0\}$. These values allow us to generate networks that resemble the characteristic properties of real-world networks. If m gets too large then the network will get too dense, whereas real-world networks are often very sparse. In addition, if p gets too low then the network will to a greater or lesser extent represent a network that is

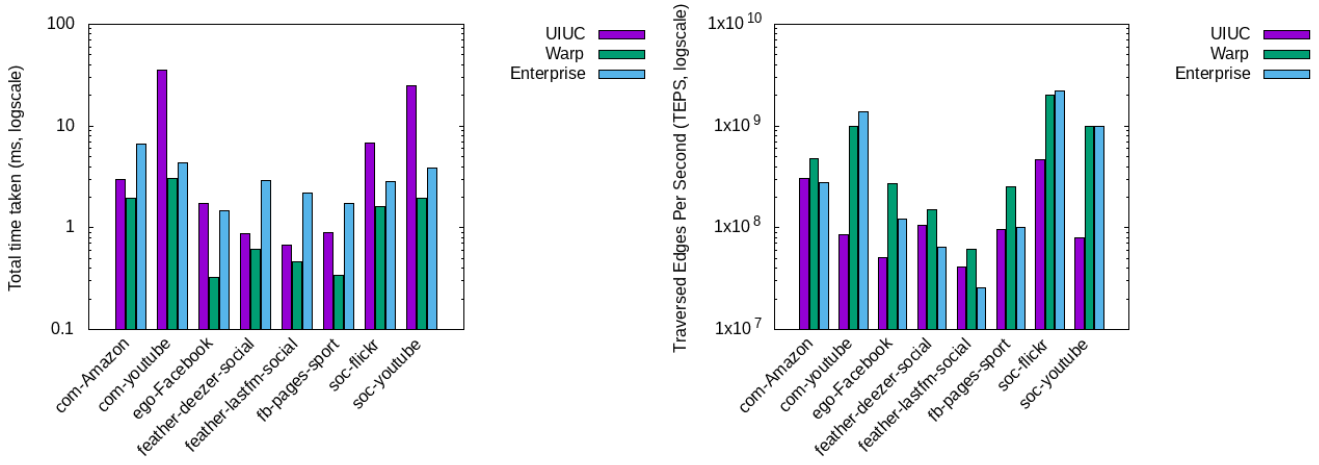
generated with the Barabási-Albert model, which fails in describing networks with high clustering. The experiments conducted in this thesis are repeated ten times to increase the accuracy of the results and reduce experimental bias.

5.3 Results

The first experiment involved both programs (SHOC and Enterprise) in which they were run on several different social network data sets. SHOC was run twice as much as Enterprise, because it includes two different implementations (one with the UIUC BFS implementation and the other with the Warp BFS implementation). Table 2 describes the properties of the social networks [15, 22] that were used for this experiment. Figure 7 shows the acquired results from the experiment. UIUC BFS tends to fall behind the other two implementations in overall performance, especially on the social network data sets *com-youtube* and *soc-youtube* in which both networks have a low global clustering coefficient and are generally larger in size compared to the other social networks. This difference in overall performance is up to 10 times slower in terms of time taken to complete one BFS and up to two times less in the number of TEPS. The results show that UIUC BFS cannot benefit as much from its hierarchical frontier queue management when the network gets too large, because of its huge kernel-launch overhead which occurs every time the frontier queue gets overcrowded. The results also show that Enterprise gets slightly worse overall performance compared to Warp BFS when a network is small to medium sized or when it has a considerable high diameter. Of all the three BFS implementations, note that Enterprise has the most optimisation techniques in which each leads to some overhead, whereas Warp BFS only has a considerable amount of overhead when each warp has to dynamically fetch a certain amount of work. This may be one of the main reasons why Enterprise tends to fall behind in overall performance on the social networks from Table 2 compared to Warp BFS.

Name	Nodes	Edges	Triangles	d	c	n	ℓ
com-Amazon	334863	925872	667129	44	0.3967	1.6514e-05	11.77
com-youtube	1134890	2987624	3056386	20	0.0808	4.6392e-06	4.24209
ego-Facebook	4039	88234	1612010	12	0.6055	1.0899e-02	3.65807
feather-deezer-social	28281	92752	45034	21	0.125	2.3194e-03	6.2528
feather-lastfm-social	7624	27806	40433	15	0.19	9.5688e-03	5.21808
fb-pages-sport	13865	86858	140023	11	0.276188	9.0358e-03	4.24062
soc-flickr	513969	3190452	58771288	18	0.1676	2.1455e-05	4.2044
soc-youtube	495957	1936748	2443886	21	0.1101	1.5747e-05	4.00465

Table 2: Properties of social networks.

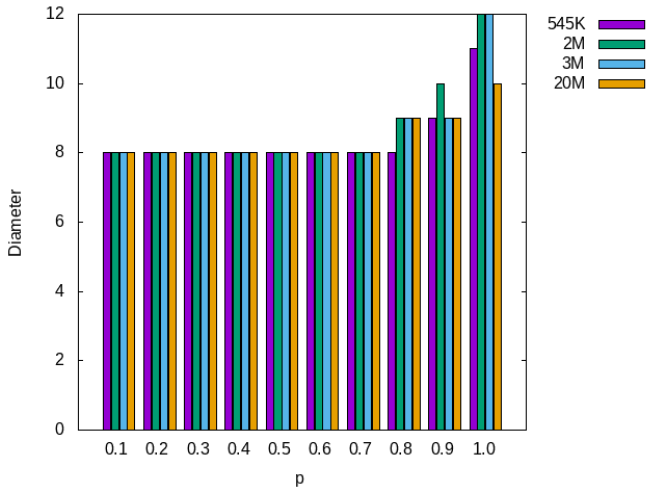


(a) Time taken to complete one BFS

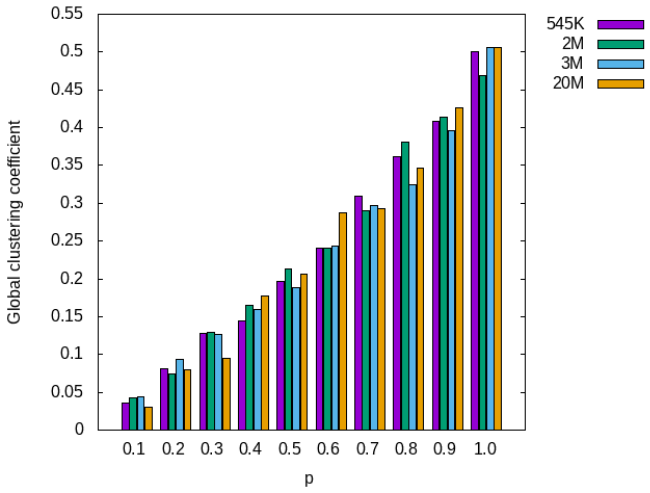
(b) Traversed Edges Per Second (TEPS)

Figure 7: Benchmark of the three BFS implementations on social networks [15, 22].

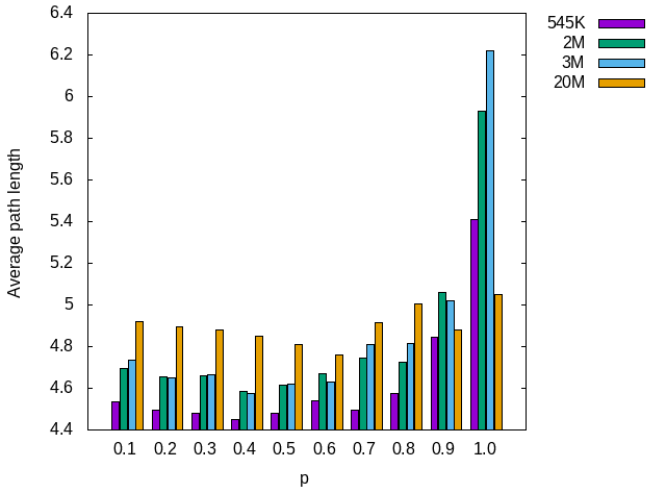
The second experiment involved generated networks that resemble the characteristic properties of real-world networks in which we control the parameter p. Figure 8 describes the important properties of the networks that were used for this experiment. Similar to the first experiment, SHOC and Enterprise were run on these networks. Figures 9, 10, 11, and 12 show similar to the previous experiment that UIUC falls drastically behind in overall performance compared to the other two BFS implementations. The results show that the performance of all the three BFS implementations are affected when manipulating the number of triangles in a network. This does not necessarily mean that we can conclude that the global clustering coefficient is the cause of this, because the results show that the diameter of the network is also changing. Although Enterprise is always able to reach the highest TEPS on these networks, its performance in terms of time taken to complete one BFS is still slightly worse than Warp BFS on the **545K** networks. We can see that the performance of Enterprise drastically increases when the networks get larger in size. This is where Warp BFS tends to lose from Enterprise in terms of overall performance, because large networks often require more distribution of work which leads to more overhead in Warp BFS. Enterprise achieves 1.2x to 3x speedup in terms of TEPS when compared to Warp BFS. Enterprise shows remarkable results when observing the Figures 9, 10, 11, and 12. However, these four figures also show that Enterprise has a tipping point in overall performance at $p = 1.0$. To understand why this is happening, we have conducted a third experiment on Enterprise.



(a) Diameter (d)

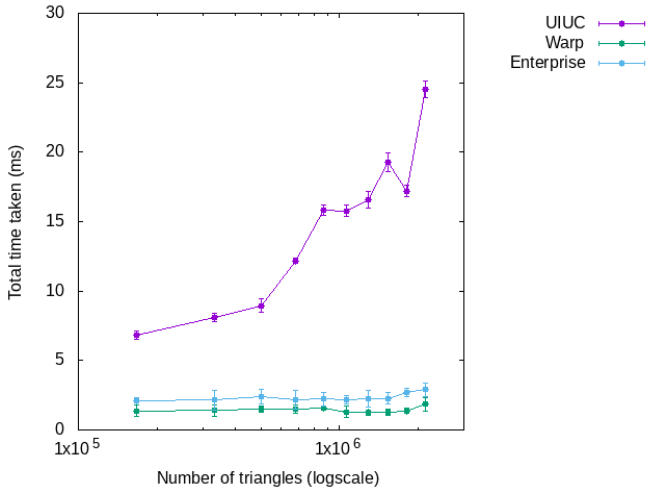


(b) Global clustering coefficient (c)

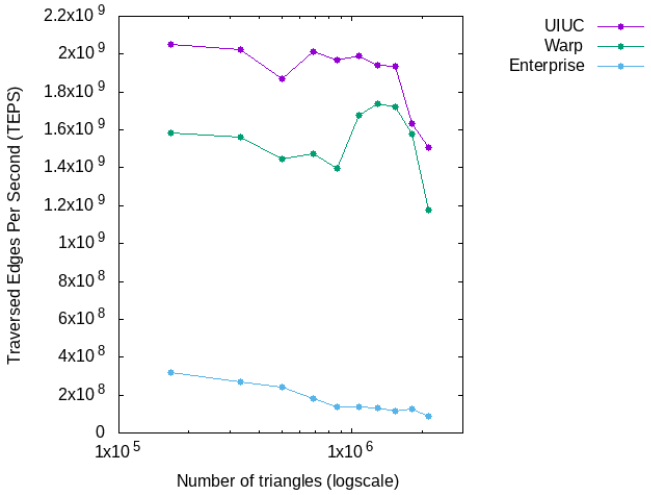


(c) Average path length (ℓ)

Figure 8: Several properties of the generated networks from Table 4.

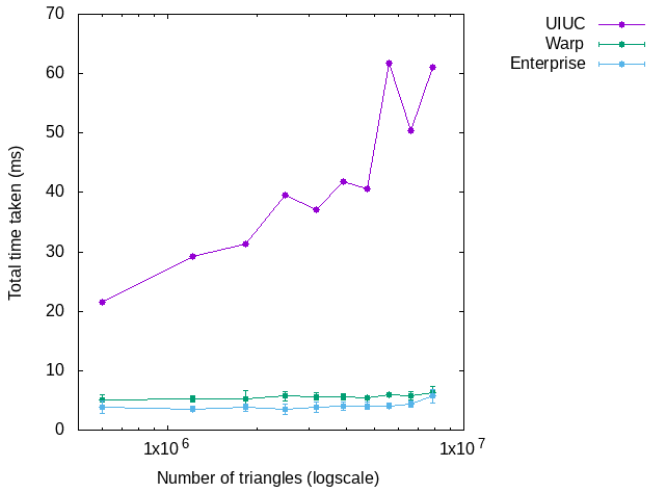


(a) Time taken to complete one BFS

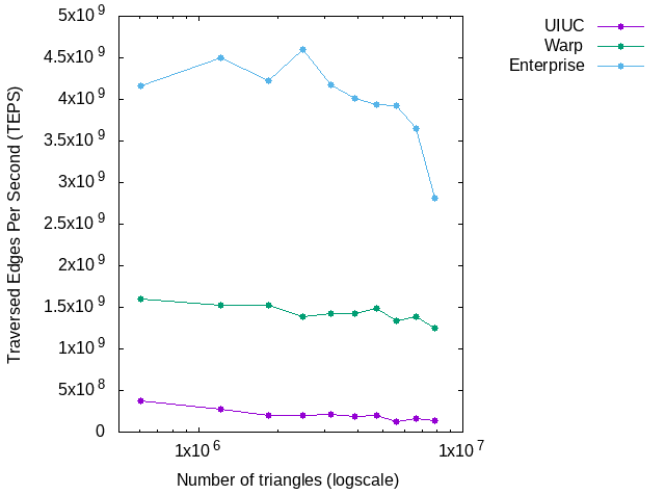


(b) Traversed Edges Per Second (TEPS)

Figure 9: Benchmark of the three BFS implementations on networks labelled as **545K** from Table 4.

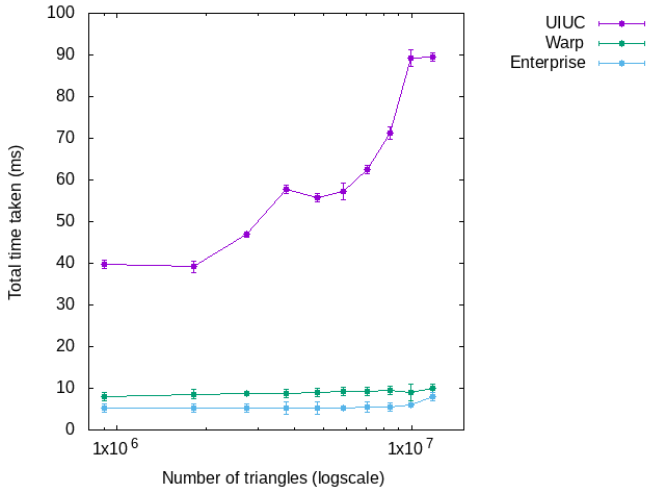


(a) Time taken to complete one BFS

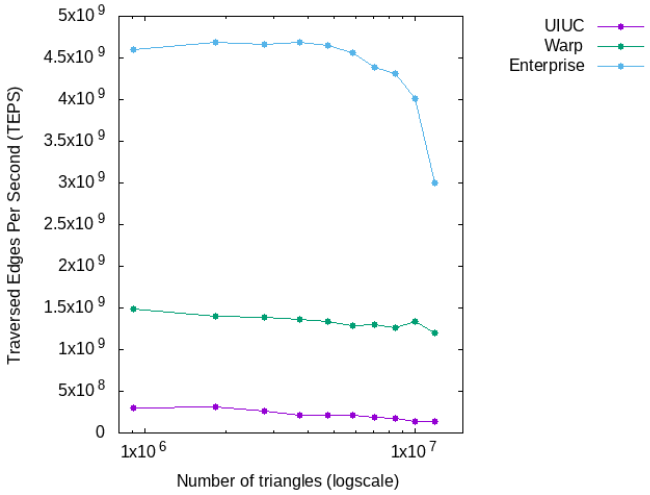


(b) Traversed Edges Per Second (TEPS)

Figure 10: Benchmark of the three BFS implementations on networks labelled as **2M** from Table 4.

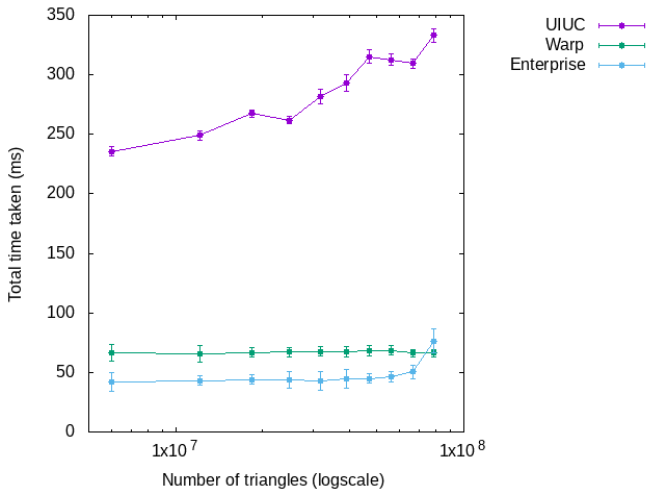


(a) Time taken to complete one BFS

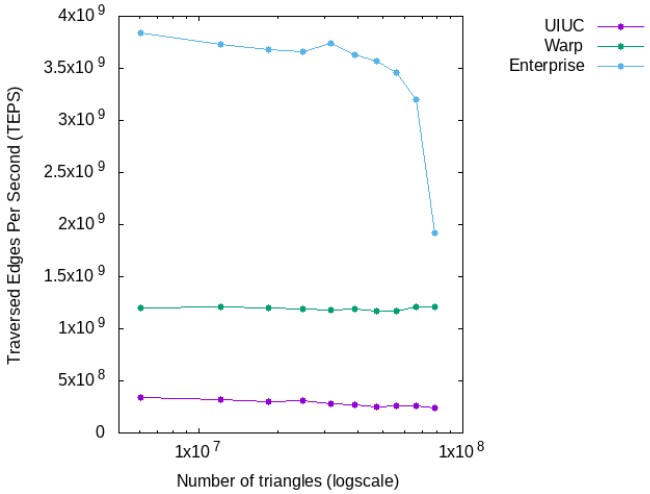


(b) Traversed Edges Per Second (TEPS)

Figure 11: Benchmark of the three BFS implementations on networks labelled as **3M** from Table 4.



(a) Time taken to complete one BFS



(b) Traversed Edges Per Second (TEPS)

Figure 12: Benchmark of the three BFS implementations on networks labelled as **20M** from Table 4.

Similar to the second experiment, the third experiment involved generated networks that resemble the characteristic properties of real-world networks, but instead of only regulating the parameter p we also regulated the parameter m. Figure 13 shows the results of Enterprise on these networks and Figure 14 describes several important properties of the networks that were used for this experiment. The results show us that Enterprise is able to reach a very high number of TEPS on the networks with a short diameter ($d < 8$). Furthermore, the results also show us that when $p = 1.0$ that the diameter of the network gets significantly higher than the ones in which $p < 1.0$. The clustering coefficient increases as well, but not as significantly as the increase in diameter. The decrease in overall performance between $p = 0.1$ and $p = 1.0$ is up to 80% in which most of the decrease in overall performance occurs between $p = 0.9$ and $p = 1.0$ (up to 60%). This might be the reason why Enterprise falls behind in overall performance when run on the social networks from Table 2. One of Enterprise its optimisation techniques is the caching of hub nodes. This technique is obsolete when there are hardly any hub nodes present in the network. The results of Figure 15 show us that the degree distribution of the networks for $p = 0.1$ up to $p = 1.0$ does not significantly change. It still follows a power-law distribution, and thus shows us that the clustering coefficient cannot be the main cause for this tipping point in overall performance. Figure 12 shows that after a certain point Warp BFS gets a better performance in terms of time taken to complete one BFS compared to Enterprise.

5.4 Exploring Enterprise

We have gathered several GPU metrics while running networks from Table 5 on Enterprise. See Table 3 for a summary of the gathered metrics. From this table we can conclude that the decrease in overall performance is mainly caused by the increase in invocations (calls) of the following kernels:

- THD_expand_sort
- WAP_expand_sort
- CTA_expand_sort

These three kernels expand the frontier queues (adding nodes to the frontier queue by examining the neighbours of the current frontiers). Nearly all the networks that were used for the third experiment have the same diameter for a particular value of m. However, when we observe the networks with $p = 1.0$ we see a significant rise in diameter and average path length compared to the other networks. The hypothesis is that Enterprise has to traverse more levels when the diameter, and thus average path length increases, which may explain the increase in invocations of the kernels. We have also noticed that in the case of a tipping point that the individual performance of these three kernels gets worse over time (i.e., later calls of the same kernel are significantly more expensive). We perceived this by observing the following metrics: the instruction count, the number of bytes that are being read from DRAM (global memory), the transaction count, and the branch efficiency. The metrics that increase drastically over time are the number of bytes that are being read from DRAM and the transaction count. To understand why especially these metrics are getting worse over time, it is helpful to explore the source code of Enterprise.

Kernel name	Invocations	Time (%)
THD_expand_sort	265	14.0
WAP_expand_sort	265	4.0
CTA_expand_sort	265	18.0
reaper	1524	9.0
__insp_pre_scan	1524	9.0
__insp_post_scan	1524	1.0

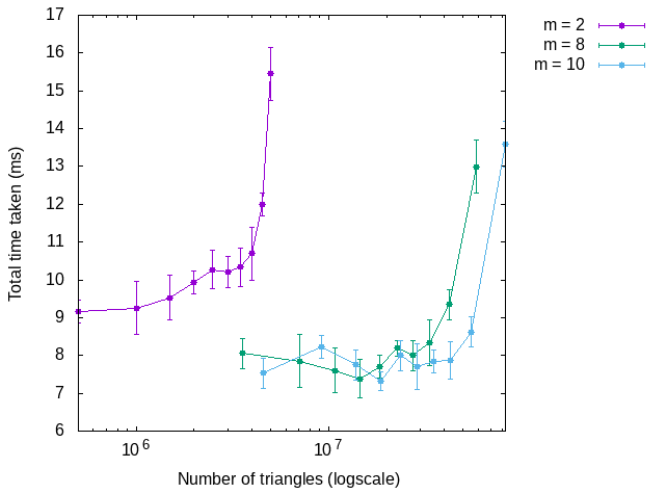
(a) On network with $m = 10$ and $p = 0.9$

Kernel name	Invocations	Time (%)
THD_expand_sort	368	26.0
WAP_expand_sort	368	10.0
CTA_expand_sort	368	21.0
reaper	2109	5.0
__insp_pre_scan	2109	4.0
__insp_post_scan	2109	< 1.0

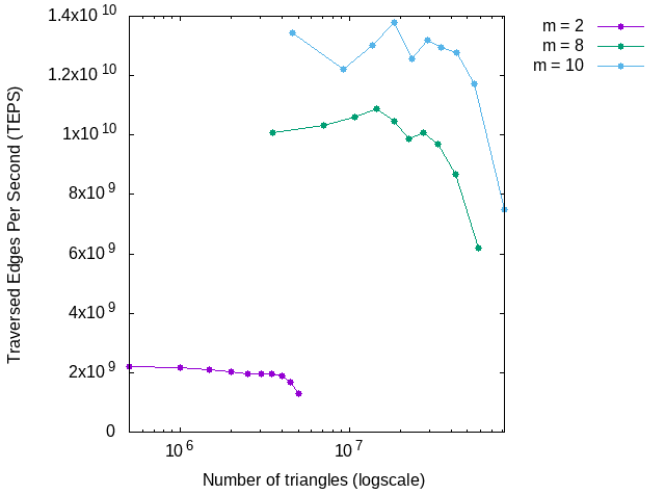
(b) On network with $m = 10$ and $p = 1$

Table 3: Summary of the gathered metrics while running Enterprise on networks from Table 5.

Enterprise has to traverse more levels when the diameter increases. As a result, the frontier queues are getting more crowded, because after every level (iteration) new frontiers are added to the queue. After a brief observation of Enterprise its source code, we observed that in the expand kernels data has to be prefetched from DRAM for every thread id (tid). More frontiers will result into more threads being assigned, which means that the number of tid's will increase. Therefore, more data is being prefetched. This may explain why there is an increase in the number of bytes being read from DRAM and in the transaction count in the later calls of the kernel. However, a further in-depth investigation of Enterprise its source code is required to accurately answer why the performance of the kernels gets worse over time, which we will leave for future work.

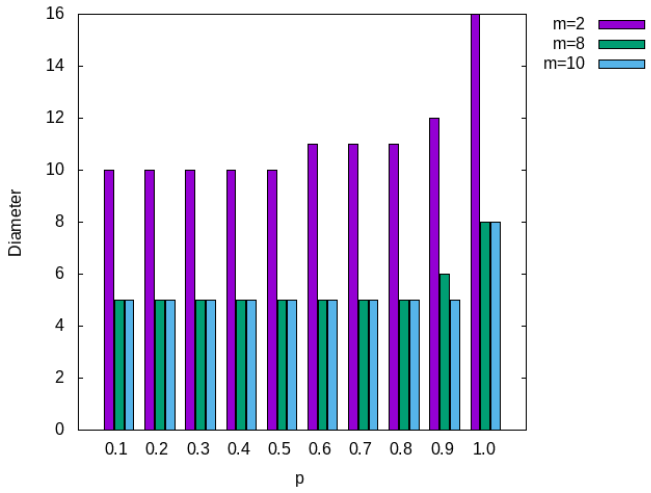


(a) Time taken to complete one BFS

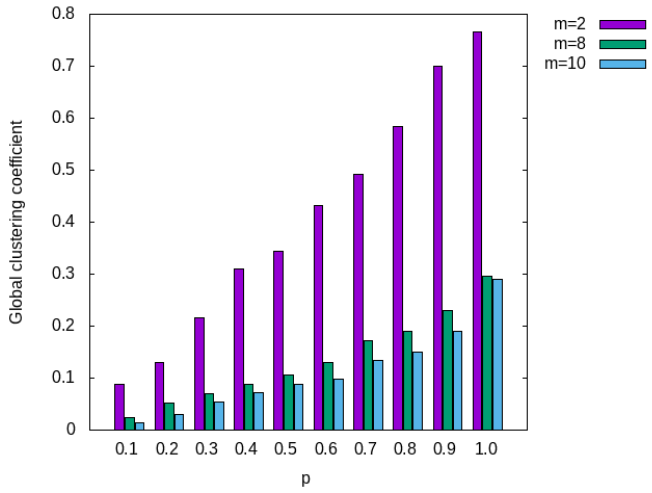


(b) Traversed Edges Per Second (TEPS)

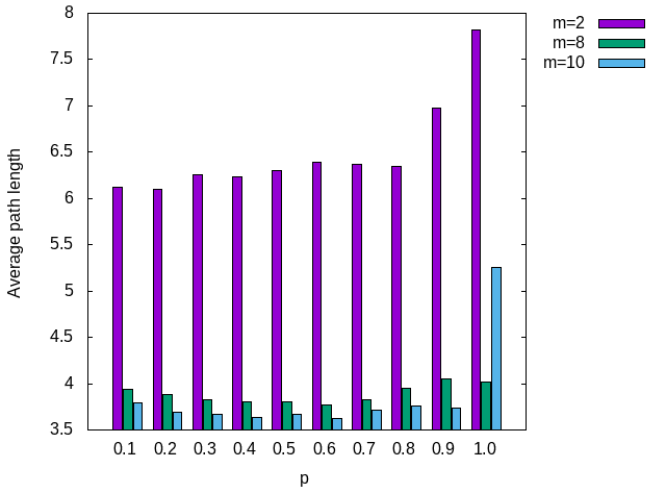
Figure 13: Benchmark of Enterprise on networks from Table 5.



(a) Diameter (d)

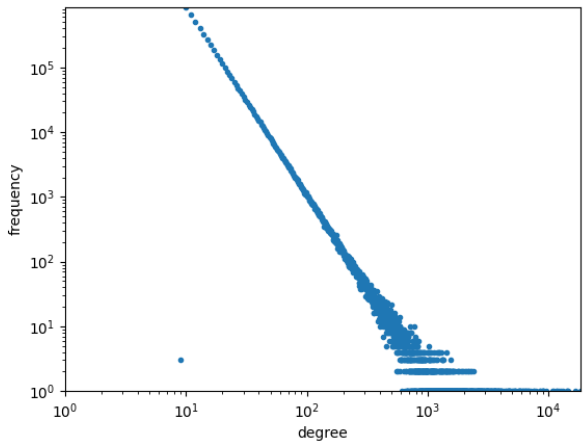


(b) Global clustering coefficient (c)

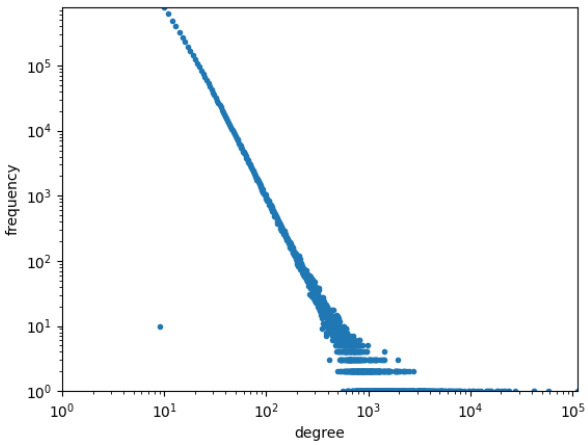


(c) Average path length (ℓ)

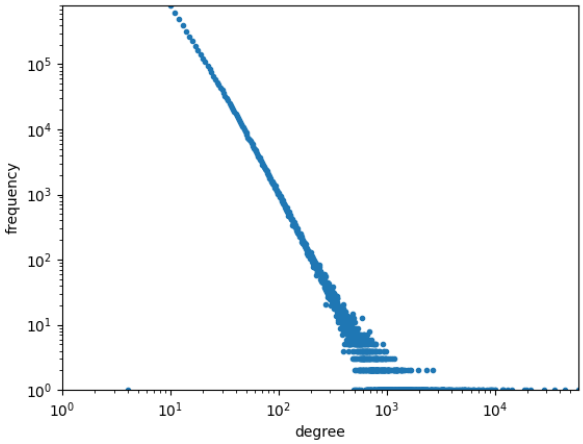
Figure 14: Several properties of the generated networks from Table 5.



(a) Degree distribution of a network labelled as **5M** with $m = 10$ and $p = 0.1$.



(b) Degree distribution of a network labelled as **5M** with $m = 10$ and $p = 0.9$.



(c) Degree distribution of a network labelled as **5M** with $m = 10$ and $p = 1.0$.

Figure 15: Degree distributions of networks from Table 5.

6 Limitations

Working on a GPU may cause several limitations, such as out of memory. For example, the machine on which the experiments were conducted is accessible by all students from the LIACS department at any moment in time. Thus, the hardware of the machine is shared among all students who access it. Therefore, it is possible that at a certain point in time while the tests were running that other students were also running their program on the GPU. Though very unlikely, this could lead to the GPU being the bottleneck and eventually have an effect on the results that were gathered from the experiments. However, we execute BFS multiple times from the same start node to prevent such exceptions (i.e., the experiments are repeated several times).

7 Conclusions and Further Research

In this work, we used three state-of-the-art BFS Implementations [14, 16, 18] on a modern GPU to investigate which implementation has the best overall performance. Moreover, we investigated if particular characteristic network properties of real-world networks have an impact on the overall performance of the BFS implementation. This was achieved by running several experiments with the three BFS implementations on various networks.

UIUC BFS falls behind in terms of overall performance compared to Warp BFS and Enterprise on all networks that were used in this thesis. Warp BFS tends to get the best overall performance out of the three BFS implementations on considerably small to medium sized social networks, because on these networks every warp only has to dynamically fetch work a few times. Therefore, Warp BFS is expected to have the least amount of overhead on these type of networks. Enterprise shows remarkable results in terms of overall performance and seems to be the best option when the size of the networks gets considerably large. On large networks, Enterprise is able to utilise its optimisation techniques more efficiently. Moreover, large networks require more distribution of work, which leads to a significant amount of overhead in Warp BFS.

Regulating the number of triangles in a network has shown to have an effect on the overall performance of all three BFS implementations. However, it is practically impossible to only change one property of a network. In our work, the parameter p does not only cause the global clustering coefficient to change, but it changes the diameter of the network as well. Enterprise show signs of a tipping point when the diameter of a network gets too high, it surpasses a certain threshold (approximately 8 on the networks run in this thesis). At this point it might be better to use Warp BFS instead of Enterprise to get a better overall performance. Figure 16 presents a flowchart which can be used to determine the best BFS implementation based on the networks that were used in this thesis.

As future research, there is the possibility to investigate the two research questions mentioned in this thesis, but then on BFS implementations that are able to run on multiple GPUs simultaneously and on even larger (real-world) networks. Our evaluation of the three BFS implementations on a modern GPU has shown that we can determine beforehand, based on the value of certain network properties, which of these implementations we should use to get the best overall performance.

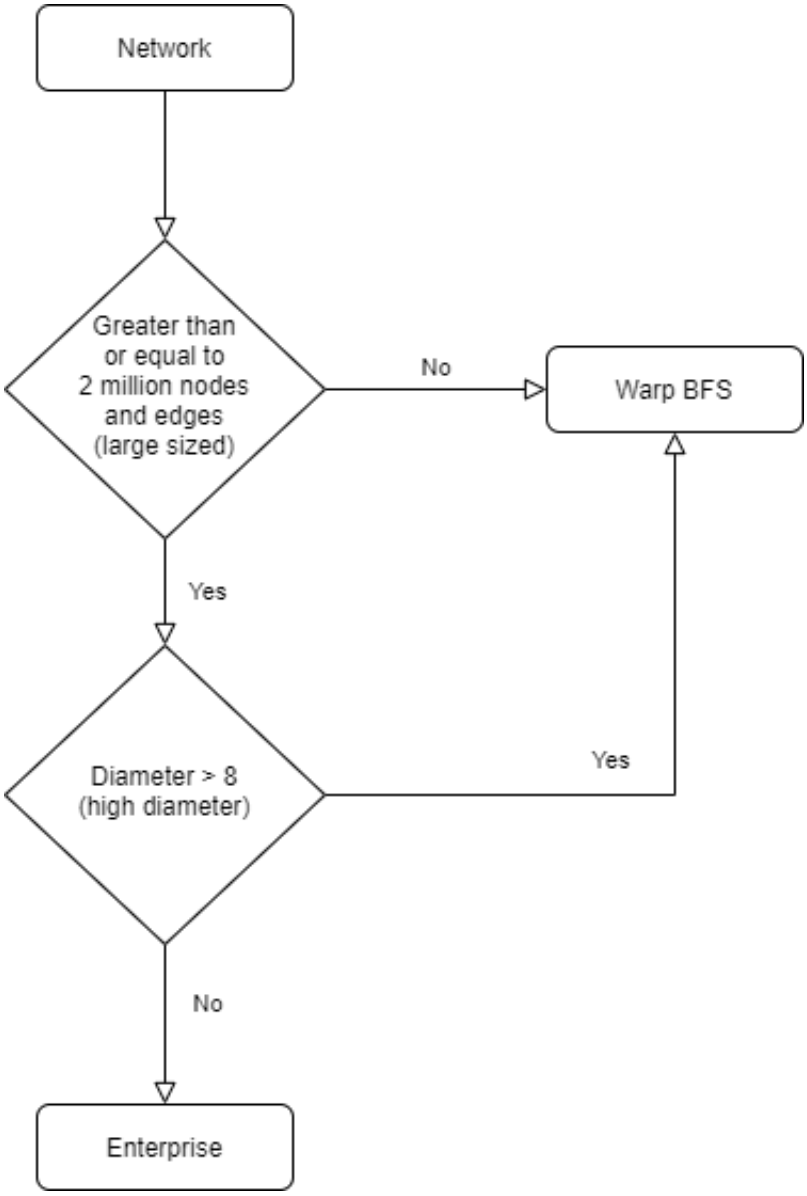


Figure 16: Flowchart on choosing BFS implementations based on the networks that were used in this thesis.

References

[1] Albert-László Barabási and Réka Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[2] Scott Beamer, Krste Asanovic, and David Patterson. Direction-Optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.

[3] Nathan Bell and Michael Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[4] Angela Bonifati, Irena Holubová, Arnau Prat-Pérez, and Sherif Sakr. Graph Generators: State of the Art and Open Challenges. *ACM Computing Surveys (CSUR)*, 53(2):1–30, 2020.

[5] NVIDIA Corporation. CUDA C++ Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Date accessed: 29-05-21.

[6] NVIDIA Corporation. NVIDIA System Management Interface (`nvidia-smi`) Documentation. https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf, Date accessed: 29-05-21.

[7] NVIDIA Corporation. NVIDIA Turing GPU Architecture, 2018. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, Date accessed: 29-05-21.

[8] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[9] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Public Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[10] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A High Level Api for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA USA, 2008.

[12] Pawan Harish and Petter J Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *International Conference on High-Performance Computing*, pages 197–208. Springer, 2007.

[13] Petter Holme and Beom Jun Kim. Growing Scale-Free Networks with Tunable Clustering. *Physical Review E*, 65(2), 2002.

[14] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. *ACM SIGPLAN Notices*, 46(8):267–276, 2011.

[15] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection, 2014. http://snap.stanford.edu/data, Date accessed: 18-05-21.

[16] Hang Liu and H Howie Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[17] R Duncan Luce and Albert D Perry. A Method of Matrix Analysis of Group Structure. *Psychometrika*, 14(2):95–116, 1949.

[18] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An Effective GPU Implementation of Breadth-First Search. In *Design Automation Conference*, pages 52–55. IEEE, 2010.

[19] Amr Makady, Anthonius de Boer, Hans Hillege, Olaf Klungel, Wim Goettsch, et al. What Is Real-World Data? A Review of Definitions Based on Literature and Stakeholder Interviews. *Value in health*, 20(7):858–865, 2017.

[20] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. *ACM SIGPLAN Notices*, 47(8):117–128, 2012.

[21] Network-Science. Node Degree Distribution. http://www.network-science.org/powerlaw_scalefree_node_degree_distribution.html, Date accessed: 19-05-21.

[22] Ryan A. Rossi and Nesreen K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*, 2015.

[23] Olivier Serrat. Social Network Analysis. In *Knowledge Solutions*, pages 39–43. Springer, 2017.

[24] Richard M Stallman. *Using the GNU Compiler Collection: a GNU Manual for GCC Version 4.8.5*. GNU Press, 2015.

[25] Ha-Nguyen Tran and Erik Cambria. A Survey of Graph Processing on Graphics Processing Units. *The Journal of Supercomputing*, 74(5):2086–2115, 2018.

[26] Guido Van Rossum and Fred L Drake Jr. *Python Reference Manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[27] David Eppstein Joseph Wang. Fast Approximation of Centrality. *Graph Algorithms and Applications*, 5(5):39, 2006.

[28] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Stanley Wasserman, Katherine Faust, et al. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

A Properties of Generated Networks

Name	Nodes	Edges	m	p	Triangles	d	c	n	ℓ
545K	5.45×10^5	2.18×10^6	4	0.1	166680	8	0.036	1.4679e-05	4.5337
	5.45×10^5	2.18×10^6	4	0.2	332666	8	0.081	1.4679e-05	4.4971
	5.45×10^5	2.18×10^6	4	0.3	502491	8	0.128	1.4679e-05	4.4823
	5.45×10^5	2.18×10^6	4	0.4	680127	8	0.144	1.4679e-05	4.452
	5.45×10^5	2.18×10^6	4	0.5	866338	8	0.196	1.4679e-05	4.4789
	5.45×10^5	2.18×10^6	4	0.6	1068207	8	0.24	1.4679e-05	4.5398
	5.45×10^5	2.18×10^6	4	0.7	1286745	8	0.309	1.4679e-05	4.4964
	5.45×10^5	2.18×10^6	4	0.8	1532163	8	0.362	1.4679e-05	4.577
	5.45×10^5	2.18×10^6	4	0.9	1810394	9	0.408	1.4679e-05	4.8453
	5.45×10^5	2.18×10^6	4	1.0	2136024	11	0.501	1.4679e-05	5.4096
2M	2.0×10^6	8.0×10^6	4	0.1	603913	8	0.043	3.9999e-06	4.6963
	2.0×10^6	8.0×10^6	4	0.2	1213925	8	0.074	3.9999e-06	4.6538
	2.0×10^6	8.0×10^6	4	0.3	1838307	8	0.129	3.9999e-06	4.6615
	2.0×10^6	8.0×10^6	4	0.4	2490341	8	0.165	3.9999e-06	4.586
	2.0×10^6	8.0×10^6	4	0.5	3176017	8	0.213	3.9999e-06	4.617
	2.0×10^6	8.0×10^6	4	0.6	3910329	8	0.241	3.9999e-06	4.668
	2.0×10^6	8.0×10^6	4	0.7	4709561	8	0.29	3.9999e-06	4.7434
	2.0×10^6	8.0×10^6	4	0.8	5611226	9	0.381	3.9999e-06	4.7248
	2.0×10^6	8.0×10^6	4	0.9	6635009	10	0.414	3.9999e-06	5.0587
	2.0×10^6	8.0×10^6	4	1.0	7846321	12	0.469	3.9999e-06	5.9276
3M	3.0×10^6	1.2×10^7	4	0.1	905268	8	0.044	2.6666e-06	4.7343
	3.0×10^6	1.2×10^7	4	0.2	1818003	8	0.094	2.6666e-06	4.6523
	3.0×10^6	1.2×10^7	4	0.3	2758575	8	0.127	2.6666e-06	4.6664
	3.0×10^6	1.2×10^7	4	0.4	3731467	8	0.159	2.6666e-06	4.5745
	3.0×10^6	1.2×10^7	4	0.5	4756351	8	0.189	2.6666e-06	4.6181
	3.0×10^6	1.2×10^7	4	0.6	5861636	8	0.244	2.6666e-06	4.6295
	3.0×10^6	1.2×10^7	4	0.7	7062594	8	0.297	2.6666e-06	4.8093
	3.0×10^6	1.2×10^7	4	0.8	8407478	9	0.324	2.6666e-06	4.8138
	3.0×10^6	1.2×10^7	4	0.9	9946021	9	0.396	2.6666e-06	5.0187
	3.0×10^6	1.2×10^7	4	1.0	11776198	12	0.506	2.6666e-06	6.2198
20M	2.0×10^7	8.0×10^7	4	0.1	6024009	8	0.03	3.9999e-07	4.9223
	2.0×10^7	8.0×10^7	4	0.2	12112922	8	0.08	3.9999e-07	4.8934
	2.0×10^7	8.0×10^7	4	0.3	18356372	8	0.095	3.9999e-07	4.8783
	2.0×10^7	8.0×10^7	4	0.4	24842475	8	0.177	3.9999e-07	4.8503
	2.0×10^7	8.0×10^7	4	0.5	31693440	8	0.206	3.9999e-07	4.8096
	2.0×10^7	8.0×10^7	4	0.6	39034169	8	0.288	3.9999e-07	4.7596
	2.0×10^7	8.0×10^7	4	0.7	47072035	8	0.293	3.9999e-07	4.9128
	2.0×10^7	8.0×10^7	4	0.8	56040249	9	0.346	3.9999e-07	5.0055
	2.0×10^7	8.0×10^7	4	0.9	66325574	9	0.426	3.9999e-07	4.8804
	2.0×10^7	8.0×10^7	4	1.0	78421646	10	0.506	3.9999e-07	5.0525

Table 4: Properties of undirected networks.

Name	Nodes	Edges	m	p	Triangles	d	c	n	ℓ
5M	5.0×10^6	1.0×10^7	2	0.1	501115	10	0.087	7.9999e-07	6.122
	5.0×10^6	1.0×10^7	2	0.2	1000569	10	0.13	7.9999e-07	6.1018
	5.0×10^6	1.0×10^7	2	0.3	1497666	10	0.215	7.9999e-07	6.2544
	5.0×10^6	1.0×10^7	2	0.4	2002127	10	0.31	7.9999e-07	6.2358
	5.0×10^6	1.0×10^7	2	0.5	2502287	10	0.344	7.9999e-07	6.3049
	5.0×10^6	1.0×10^7	2	0.6	3002694	11	0.431	7.9999e-07	6.3881
	5.0×10^6	1.0×10^7	2	0.7	3502016	11	0.491	7.9999e-07	6.3648
	5.0×10^6	1.0×10^7	2	0.8	4000484	11	0.584	7.9999e-07	6.3469
	5.0×10^6	1.0×10^7	2	0.9	4500684	12	0.7	7.9999e-07	6.9728
	5.0×10^6	1.0×10^7	2	1.0	4999997	16	0.765	7.9999e-07	7.8186
	5.0×10^6	4.0×10^7	8	0.1	3553008	5	0.024	3.1999e-06	3.9417
	5.0×10^6	4.0×10^7	8	0.2	7106175	5	0.052	3.1999e-06	3.8849
	5.0×10^6	4.0×10^7	8	0.3	10728317	5	0.07	3.1999e-06	3.8318
	5.0×10^6	4.0×10^7	8	0.4	14468920	5	0.088	3.1999e-06	3.7993
	5.0×10^6	4.0×10^7	8	0.5	18388977	5	0.106	3.1999e-06	3.802
	5.0×10^6	4.0×10^7	8	0.6	22636241	5	0.129	3.1999e-06	3.7681
	5.0×10^6	4.0×10^7	8	0.7	27514852	5	0.171	3.1999e-06	3.824
	5.0×10^6	4.0×10^7	8	0.8	33561043	5	0.189	3.1999e-06	3.9547
	5.0×10^6	4.0×10^7	8	0.9	42366853	6	0.229	3.1999e-06	4.0498
	5.0×10^6	4.0×10^7	8	1.0	58515125	8	0.295	3.1999e-06	4.0195
	5.0×10^6	5.0×10^7	10	0.1	4600240	5	0.013	3.9999e-06	3.7897
	5.0×10^6	5.0×10^7	10	0.2	9177923	5	0.03	3.9999e-06	3.6864
	5.0×10^6	5.0×10^7	10	0.3	13812120	5	0.053	3.9999e-06	3.6637
	5.0×10^6	5.0×10^7	10	0.4	18591819	5	0.072	3.9999e-06	3.6309
	5.0×10^6	5.0×10^7	10	0.5	23560996	5	0.088	3.9999e-06	3.6699
	5.0×10^6	5.0×10^7	10	0.6	28930466	5	0.098	3.9999e-06	3.6247
	5.0×10^6	5.0×10^7	10	0.7	35084286	5	0.133	3.9999e-06	3.7089
	5.0×10^6	5.0×10^7	10	0.8	42934967	5	0.149	3.9999e-06	3.7577
	5.0×10^6	5.0×10^7	10	0.9	55256038	5	0.189	3.9999e-06	3.7363
	5.0×10^6	5.0×10^7	10	1.0	82813921	8	0.289	3.9999e-06	5.2532

Table 5: Properties of undirected networks.