Optimized Parallel Breadth-First Search with Adaptive Strategies

Chaoqun Li
University of Texas at Arlington

Runbang Hu
University of Texas at Arlington

Xiaojiang Du
Stevens Institute of Technology

Yuede Ji
University of Texas at Arlington

Abstract

Breadth-First Search (BFS) is a fundamental graph traversal algorithm in a level-by-level pattern. It has been widely used in real-world applications, such as social network analysis, scientific computing, and web crawling. However, achieving high performance for BFS on large-scale graphs remains a challenging task due to irregular memory access patterns, diverse graph structures, and the necessity for efficient parallelization. This paper addresses these challenges by designing a highly optimized parallel BFS implementation based on the top-down and bottom-up traversal strategies. It further integrates several key innovations, including graph type-aware computation strategy selection, graph pruning, two-level bottom-up, and efficient parallel implementation. We evaluate our method on 11 diverse graphs in terms of size, diameter, and density. On a CPU server with 48 threads, our method achieves an average speedup of 9.5× over the serial BFS implementation. Also, on a synthetic dense graph, our method processes 9.3 billion edges per second, showing its efficiency in dense graph traversal.

ACM Reference Format:
Chaoqun Li, Runbang Hu, Xiaojiang Du, and Yuede Ji. 2025. Optimized Parallel Breadth-First Search with Adaptive Strategies. In *FastCode Programming Challenge (FCPC '25), March 1–5, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3711708.3723449

1 Introduction

Breadth-First Search (BFS) is a fundamental graph traversal algorithm in a level-by-level pattern [26]. As it can be easily parallelized, it has been widely used in real world applications, such as social network analysis [29], scientific computing [32], web crawling [25], and cybersecurity applications [5, 9, 13, 14, 17–19, 21]. However, achieving high performance for BFS on large-scale graphs remains a challenging task due to irregular memory access patterns, diverse graph structures, and the necessity for efficient parallelization [24].

This paper addresses these challenges by presenting a highly optimized parallel BFS implementation that dynamically adapts to graph characteristics while leveraging modern parallel processing techniques. Our method is based on the top-down and bottom-up traversal strategies [7], and further integrates several key innovations.

(i) Graph type-aware computation strategy selection. To enhance efficiency, we introduce a lightweight graph initialization technique that assigns a 4-bit identifier to each graph, enabling rapid classification and adaptive strategy selection. This preprocessing step significantly reduces overhead by tailoring traversal methods to specific graph structures.

(ii) Graph pruning. We observe a special type of vertices along with their edges that can be quickly pruned without affecting the correctness of BFS traversal, which are the size-1 and size-2 connected components (CCs) [16, 20].

(iii) Two-level bottom-up optimization. For graphs that fit within cache, we employ a two-level bottom-up traversal strategy [12] that accelerates computation. This approach optimally balances coarse- and fine-grained parallelism, substantially improving performance for small graphs.

(iv) Efficient parallel implementation. We leverage both OpenMP [4] and OpenCilk [28] to achieve scalable parallel execution. Our implementation incorporates atomic operations and efficient memory management to handle concurrent updates to the frontier and distance arrays, ensuring robustness across multi-core architectures.

We evaluate our method on 11 graphs, including 6 real-world graphs and 5 synthetic graphs. These graphs are diverse in terms of size, diameter, and density, providing a comprehensive testbed for assessing scalability and performance. On a CPU server with 48 threads, our method achieves an average speedup of 9.5× over the serial BFS implementation. Notably, our approach demonstrates significant performance gains, achieving up to 155× speedup for a dense and large graph. Also, on a synthetic dense graph, our method processes 9.3 billion edges per second, showing its efficiency in dense graph traversal.

FCPC '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1446-7/2025/03
https://doi.org/10.1145/3711708.3723449

2 Background

Graph representation. A graph is defined as $G(V, E)$, where V denotes the set of vertices (nodes), and E denotes the set of edges. Each edge $(u, v) \in E$ connects vertices u and v. We use N, M to denote the number of vertices and edges in the graph, respectively. We store the graph in compressed sparse row (CSR) format [6], which is widely used in graph analytics [11, 20, 22, 23, 27, 30]. The CSR includes two arrays: the adjacent edges with the length of M and the begin positions for each vertex at the adjacent edge array with the length of $N + 1$. This format offers efficient storage and fast access to adjacency information, making it particularly well-suited for graph traversal algorithms.

Breadth-First Search (BFS) is a fundamental graph traversal algorithm that explores all the neighbors of a vertex before moving to the next level of neighbors, i.e., level-by-level traversal [26]. It operates in $O(N + M)$ time complexity, making it efficient for large graphs.

Direction-Optimizing BFS is an efficient parallel BFS algorithm, which switches between the conventional top-down BFS and a newly introduced bottom-up BFS [3]. **Top-down BFS** produces a tree in a top-down manner, i.e., traversing the graph from the root by increasing the depth one level at a time. A data structure called frontier queue (FQ) contains the vertices that are visited in the preceding level. For each frontier, top-down BFS expands its out-neighbors, inspects their statuses, and subsequently marks the unvisited neighbors as the new frontiers.

Bottom-up BFS reverses the expand direction, i.e., from unvisited to visited. For each *unvisited* vertex, bottom-up BFS expands its *in-neighbors*, inspects their statuses, and if any in-neighbor has been visited in the preceding level, it will mark the unvisited vertex as visited. The inspection can be terminated as soon as the first visited in-neighbor is found, which is called *early termination*. Benefited from the reverse direction and early termination, bottom-up BFS is more efficient during the middle traversal levels.

3 Methodology

In this section, we outline the methodologies used to process and traverse graphs efficiently. The approach is divided into four main steps: preprocessing the graph, performing a two-level bottom-up traversal for small graphs, switching between top-down and bottom-up traversal using atomic operations and efficient memory storage, and applying different parallelization strategies based on the graph type.

3.1 Graph Type-Aware Computation Strategy Selection

We identify four types of graphs (marked by a 4-bit mask), including the power-law graph, density graph, high-degree skewed graph, and cache fitting graph. Knowing their types will enable fast decision-making during the later stages of the BFS traversal.

(i) Power-law graph (0b0001) is a type of graph in which the degree distribution follows a power-law function, meaning that a small number of vertices have very high degrees, while most vertices have relatively few connections. To quantify this, we use the ratio of the maximum vertex degree over the average degree, $max/average > \alpha$ (e.g., $\alpha = 16$). These graphs often benefit from the direction-optimizing BFS.

(ii) High-degree skewed graph (0b0010) refers to a special power-law graph that includes some hub vertices with extremely high degrees, such as, greater than a threshold β (e.g., $\beta = 256$). This imbalance in degree distribution often results in workload imbalance in parallel BFS, requiring special computation strategies.

(iii) Dense graph (0b0100) refers to a graph that has a high average degree, such as $D_{avg} > \gamma$ (e.g., $\gamma = 10$). This indicates that most vertices have a relatively large number of neighbors. For such graphs, direction-optimizing BFS, especially bottom-up traversal, can significantly improve performance by reducing unnecessary edge traversals.

(iv) Cache-fitting graph (0b1000) refers to a small graph, whose CSR representation can be entirely placed within the L1 cache. When this condition holds, we observe that the cache locality can benefit the most for performance, allowing compact data structures and sequential memory access. For example, if one uses the unsigned integer data type for begin position and adjacent list arrays, and the L1 cache size is 32 KB, this condition can be expressed as $4N + 4M < 32 * 1024$.

In summary, by classifying graphs into these types, we can optimize graph traversal strategies accordingly. For example, the power-law graphs may require adaptive workload partitioning, while dense graphs benefit from bottom-up traversal to minimize redundant vertex explorations. High-degree skewed graphs require specialized load-balancing techniques to mitigate the impact of high-degree vertices. Lastly, for small graphs, we can tradeoff parallelism for better cache locality. Note that, a graph can combine multiple types, where we will also combine different computation strategies.

3.2 Graph Pruning and Reordering

Graph pruning. We observe a special type of vertices that can be quickly pruned without affecting the correctness of BFS traversal, which are the size-1 and size-2 connected components (CCs) [16, 20]. The size-1 CC is a standalone vertex without any neighbors, which can be quickly pruned by checking its degree. The size-2 CC only has two vertices with one edge. We can also quickly find them by checking whether the neighbor of a degree-1 vertex also has 1 degree.

This pruning strategy is particularly effective for sparse and power-law graphs, where redundant traversal of vertices and edges are significant, especially in bottom-up BFS. By reducing the overall graph size, we reduce the workload, memory access overhead, and also improve cache locality. Note that, even if a pruned vertex is identified as the source

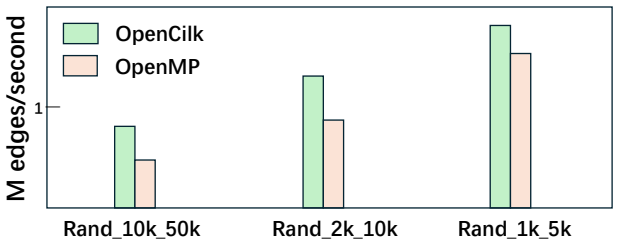

Figure 1. Performance comparison between OpenCilk and OpenMP for small graphs.

Table 1. Graph benchmarks (sorted by vertex count).

Graph	Avg. D	Max D	$\|V\|$	$\|E\|$
Road_Net_2	2	26	86.08M	216M
kNN_Graph_1	6	766	24.63M	154M
Road_Net_1	2	21	21.87M	58M
Synth_Dense_1	98	155	9.90M	980M
Synth_Sparse_1	3	4	9.90M	39M
Web_Graph_1	44	234,198	6.56M	294M
Social_Net_1	17	20,134	4.80M	84M
Collab_Net_1	104	11,358	1.06M	110M
Rand_2k_10k	9	22	2K	19.94K
Rand_1k_5k	9	22	1K	9.94K
Rand_10k_50k	9	23	10K	100K

vertex, we can immediately complete without going through the typical BFS computation.

Graph reordering. We also explored graph reordering as a preprocessing technique. The goal of graph reordering is to enhance cache locality by ensuring that frequently accessed vertices are placed close together in memory, reducing random memory accesses and improving traversal performance. However, the computation of graph reordering itself might be costly, e.g., sometimes more than the BFS traversal. Due to that, we only explore a lightweight degree-based reordering technique. That is, we reorder the vertices in descending order of degree, placing high-degree vertices first to improve memory access patterns.

While graph reordering led to performance gains for certain graph datasets, our evaluation revealed that the total benchmark scores were lower compared to our optimized implementation without reordering. This is likely due to additional overhead introduced by the reordering process, which offsets the benefits in some graphs. As the benefits are not generic, we decided to **not use graph reordering**.

3.3 Two-Level Bottom-Up BFS

The two-level bottom-up BFS extends the traditional bottom-up approach by exploring not only the current but also the next level in each iteration [12]. This strategy introduces an early termination flag to eliminate redundant computations and accelerate convergence. The idea is, if a vertex has a neighbor at the current level, its distance is updated immediately. Otherwise, if a neighbor is found at the next level, it will be assigned a distance two levels ahead, reducing the workload in next iteration.

The correctness of this two-level bottom-up BFS has been proved by a prior work, i.e., XBFS [12]. This work also demonstrated that it can significantly improve the performance over the conventional bottom-up BFS, particularly in graphs that allow aggressive level propagation. By reducing traversal overhead, this approach enhances efficiency and speeds up BFS computation.

3.4 Parallel Programming Library

To optimize BFS performance across different types of graphs, we utilize OpenMP [4] and OpenCilk [28] based on the graph's characteristics. We also adjust the threshold values and switching timing to further enhance performance.

From our observation, OpenMP is ideal for sparse graphs, providing efficient coarse-grained parallelism with minimal synchronization. It is well-suited for structured parallelization, where tasks can be distributed in larger chunks. OpenCilk, on the other hand, excels with fine-grained task parallelism and dynamic load balancing, making it particularly effective for dense graphs with high-degree vertices. Our BFS implementation switches between OpenMP and OpenCilk and adapts between top-down and bottom-up traversal based on graph density and the workload.

In particular, for large or sparse graphs, OpenMP is used with an initial top-down step. We then switch to bottom-up when the level exceeds the top threshold and return to top-down once the level drops below the bottom threshold. These adaptive thresholds ensure that we adjust the switching point dynamically to achieve the best performance based on the graph's size and structure.

For small graphs, we employ OpenCilk and transition from top-down to bottom-up when the active frontier surpasses a top threshold. Figure 1 compares the performance between OpenCilk and OpenMP for small graphs. If the frontier shrinks below a bottom threshold, we switch back to top-down for efficiency. We fine-tune the threshold values and switch conditions based on the graph's structure to maximize performance.

4 Experiment

4.1 Graph Benchmarks

Table 1 summarizes the details of the tested graph datasets, including both real-world and synthetic graphs. The datasets vary in terms of size, diameter, degree distribution, and sparsity, providing comprehensive evaluations for the algorithm's scalability and efficiency.

Table 2. Runtime performance (in ms).

Graph	Baseline	Ours	Speedup	B edges/s
Road_Net_2	14,670	740	19.82	0.29
kNN_Graph_1	2,100	220	9.55	0.69
Road_Net_1	3,310	310	10.68	0.19
Synth_Dense_1	11,870	110	108.82	9.30
Synth_Sparse_1	1,620	200	8.10	0.20
Web_Graph_1	2,860	18.40	155.43	15.99
Social_Net_1	1,060	16.23	65.31	5.17
Collab_Net_1	0.47	3.79	123.96	29.13
Rand_2k_10k	0.05	0.02	2.50	1.11
Rand_1k_5k	0.02	0.01	2.00	1.84
Rand_10k_50k	0.35	0.20	1.75	0.50

4.2 Runtime Performance

We test the methods on a CPU server with 48 threads and 96 GB memory on the Speedcode platform [31]. Table 2 shows the runtime performance. The speedup is calculated by the runtime of baseline (serial top-down BFS) over ours.

We can see that our optimized BFS algorithm demonstrates significant speedup over the baseline method across various graph benchmarks. On average, our method achieves an 9.5× speedup across all graphs. The highest speedup is observed in Web_Graph_1, reaching 155.43×, while the lowest improvement is in small random graphs, such as random_10k_50k, with only 1.75× speedup. Dense graphs, including Synth_Dense_1 and Collab_Net_1, benefit the most due to high connectivity, allowing better workload parallelization. Meanwhile, road networks (e.g., Road_Network_1 and Road_Network_2) show moderate acceleration, due to their sparse and structured nature limiting parallelism.

4.3 Performance Study of Direction Switching

By applying graph type-aware computation, two-level bottom-up BFS, and direction-optimizing BFS, along with customized strategies designed for different graph structures, we achieved a remarkable performance boost. Our optimizations led to a substantial increase in average processing speed, especially for dense graphs, reaching an impressive 137.25× improvement over the baseline.

Additionally, we study the impact of switching strategies on denser and more uniform graphs, as shown in Figure 2. For these graphs, we optimize the switching threshold between top-down and bottom-up approaches based on the current workload, enabling earlier switching to bottom-up traversal and earlier switching back to top-down, thereby enhancing performance.

This helps to improve the performance by at least 3× compared to the parallel direction-optimizing BFS implementation. By fine-tuning the switching threshold between the top-down and bottom-up phases, we identify a value that can greatly enhance the final performance.

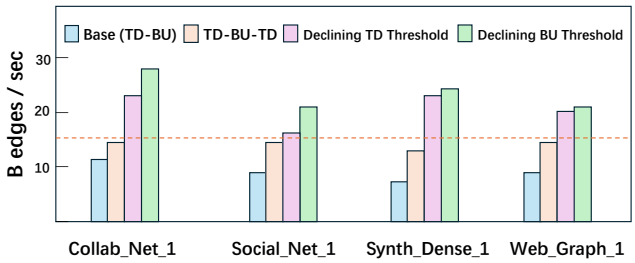

Figure 2. Performance of direction-optimizing switch on dense graphs.

5 Related Work

Parallel BFS has been widely studied, with optimizations focusing on workload balancing, memory efficiency, and traversal strategies. Traditional top-down and bottom-up approaches have been enhanced through direction-optimizing switch mechanisms [8], which dynamically alternate between strategies based on frontier size. However, determining optimal switching points remains a challenge due to graph structure variability and runtime unpredictability.

Graph preprocessing techniques [10] such as reordering [1] and pruning [2] improve cache locality and reduce the search space, leading to potential performance gains. Reordering strategies aim to enhance data locality by restructuring adjacency lists, while pruning techniques [15] eliminate unnecessary computations by removing redundant or low-impact edges. However, these preprocessing steps introduce additional overhead that may offset benefits, particularly for dynamic or frequently updated graphs [33].

6 Conclusion

This paper designs a highly optimized parallel BFS implementation based on the top-down and bottom-up traversal strategies. It further integrates several key innovations, including graph type-aware computation strategy selection, graph pruning, two-level bottom-up, and efficient parallel implementation. We evaluate our method on 11 diverse graphs in terms of size, diameter, and density. On a CPU server with 48 threads, our method achieves an average speedup of 9.5× over the serial BFS implementation. Also, on a synthetic dense graph, our method processes 9.3 billion edges per second, showing its efficiency.

Acknowledgment

This work was supported in part by National Science Foundation grants 2331301, 2508118, 2516003, 2419843. The views, opinions, and/or findings expressed in this material are those of the authors and should not be interpreted as representing the official views of the National Science Foundation, or the U.S. Government.

References

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–38.

[2] K Balasubramanian. 1990. Recent developments in tree-pruning methods and polynomials for cactus graphs and trees. *Journal of Mathematical Chemistry* 4, 1 (1990), 89–102.

[3] Scott Beamer, Krste Asanovic, and David Patterson. 2013. Direction-optimizing breadth-first search. *Scientific Programming* 21, 3-4 (2013), 137–148.

[4] OpenMP Architecture Review Board. 2018. OpenMP API specification for parallel programming. https://www.openmp.org. Accessed: 2025-02-03.

[5] Benjamin Bowman, Craig Laprade, Yuede Ji, and H. Howie Huang. 2020. Detecting Lateral Movement in Enterprise Computer Networks with Unsupervised Graph AI. In *Proceedings of RAID*.

[6] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (SPAA)*. ACM, 233–244.

[7] Aydin Buluç and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[8] Lin Chen, Keisuke Asai, Taku Nonomura, Guannan Xi, and Tianshu Liu. 2018. A review of Backward-Facing Step (BFS) flow mechanisms, heat transfer and control. *Thermal Science and Engineering Progress* 6 (2018), 194–216.

[9] Lei Cui, Jiancong Cui, Yuede Ji, Zhiyu Hao, Lun Li, and Zhenquan Ding. 2023. API2Vec: Learning Representations of API Sequences for Malware Detection. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[10] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. 2006. High-performance multi-level graphs. *9th DIMACS Challenge on Shortest Paths* (2006), 1–13.

[11] Wang Feng, Shiyang Chen, Hang Liu, and Yuede Ji. 2023. Peek: A Prune-Centric Approach for K Shortest Path Computation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[12] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: eXploring runtime optimizations for breadth-first search on GPUs. In *Proceedings of the 28th International symposium on high-performance parallel and distributed computing*. 121–131.

[13] Haoyu He, Yuede Ji, and H. Howie Huang. 2022. Illuminati: Towards Explaining Graph Neural Networks for Cybersecurity Analysis. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.

[14] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024. Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection. In *The 33rd USENIX Security Symposium (USENIX Security)*.

[15] Yang He and Lingao Xiao. 2023. Structured pruning for deep convolutional neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence* (2023).

[16] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.

[17] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network. In *16th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*.

[18] Yuede Ji, Lei Cui, and H Howie Huang. 2021. Vestige: Identifying Binary Code Provenance for Vulnerability Detection. In *International Conference on Applied Cryptography and Network Security (ACNS)*. Springer, 287–310.

[19] Yuede Ji, Mohamed Elsabagh, Ryan Johnson, and Angelos Stavrou. 2021. DEFInit: An Analysis of Exposed Android Init Routines. In *30th USENIX Security Symposium (USENIX Security)*.

[20] Yuede Ji and H. Howie Huang. 2020. Aquila: Adaptive Parallel Computation of Graph Connectivity Queries. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[21] Yuede Ji and H. Howie Huang. 2022. NestedGNN: Detecting Malicious Network Activity with Nested Graph Neural Networks. In *IEEE International Conference on Communications (ICC)*.

[22] Yuede Ji, Hang Liu, and H. Howie Huang. 2018. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 731–742.

[23] Yuede Ji, Hang Liu, and H. Howie Huang. 2020. SwarmGraph: Analyzing Large-Scale In-Memory Graphs on GPUs. In *International Conference on High Performance Computing and Communications (HPCC)*. IEEE.

[24] Zite Jiang, Tao Liu, Shuai Zhang, Zhen Guan, Mengting Yuan, and Haihang You. 2020. Fast and efficient parallel breadth-first search with power-law graph transformation. *arXiv preprint arXiv:2012.10026* (2020).

[25] Nithin T K, Chandana S, Barani G, Chavva Dharani, and M S Karishma. 2023. Comparative analysis of various web crawler algorithms. *arXiv preprint arXiv:2306.12027* (2023).

[26] Donald E. Knuth. 1974. *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (2nd ed.). Addison-Wesley.

[27] Hang Liu, H Howie Huang, and Yang Hu. 2016. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. ACM, 403–416.

[28] J. S. Plank and N. Zeldovich. 2019. OpenCilk: A parallel programming model for efficient, scalable execution. https://www.opencilk.org. Accessed: 2025-02-03.

[29] John Scott. 2011. Social network analysis: developments, advances, and prospects. *Social network analysis and mining* 1 (2011), 21–26.

[30] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 550–559.

[31] Speedcode. [n. d.]. Speedcode. https://speedcode.org/

[32] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.

[33] Leshanshui Yang, Clément Chatelain, and Sébastien Adam. 2024. Dynamic graph representation learning with neural networks: A survey. *IEEE Access* 12 (2024), 43460–43484.