Fast Exact Diameter Computation of Sparse Graphs

Cameron Lloyd Bradley
clb420@txstate.edu
Texas State University
San Marcos, Texas, USA

Anju Mongandampulath Akathoott
anju.m.a@txstate.edu
Texas State University
San Marcos, Texas, USA

Martin Burtscher
burtscher@txstate.edu
Texas State University
San Marcos, Texas, USA

Abstract

The diameter of a graph is a fundamental topological parameter that provides valuable insight needed in multiple areas of graph analytics. The traditional approach to computing the diameter is solving the all-pairs shortest-paths problem (APSP). Since APSP has a time complexity that is at least quadratic in the size of the graph, it is impractical for large graphs. As a remedy, leading algorithms use Breadth-First Search (BFS) combined with various optimizations to limit the number of BFS calls required to find the diameter. We present a new algorithm called F-Diam for quickly computing the exact diameter of large graphs. It includes new techniques such as Winnowing to greatly reduce the number of BFS calls. Our parallel CPU implementation of F-Diam is faster than the state of the art on all tested inputs, often by orders of magnitude.

CCS Concepts

• **Computing methodologies** → **Shared memory algorithms**;
• **Theory of computation** → **Shortest paths**.

Keywords

Graph diameter, eccentricity, parallelism, sparse graphs

ACM Reference Format:
Cameron Lloyd Bradley, Anju Mongandampulath Akathoott, and Martin Burtscher. 2025. Fast Exact Diameter Computation of Sparse Graphs. In *54th International Conference on Parallel Processing (ICPP '25), September 08–11, 2025, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3754598.3754620

1 Introduction

The *diameter* of a graph describes the length of the longest of all shortest paths between any two vertices of the graph. In other words, no matter which vertex we start from, we can always reach all other vertices by traversing no more than *diameter* edges. For instance, the graph in Figure 1 has a diameter of 2 because no vertex is more than two edges away from any other vertex.

The *eccentricity* of a vertex is the length of the longest shortest path starting at that vertex. For example, vertex A in Figure 1 has an eccentricity of 1 because all other vertices are just one edge away. For the same reason, vertex D also has an eccentricity of 1. However, vertex B has an eccentricity of 2 because we must minimally traverse 2 edges to reach vertex C, which is the longest shortest path in this case. Symmetrically, vertex C also has an eccentricity of 2. As a consequence, this graph has a diameter of 2 because the diameter is always equal to the largest eccentricity.

In graph analytics, the diameter of a graph is a key property. For instance, in communication networks, it indicates the maximum message delay [12], in social networks, it shows how closely connected the individuals are [8], in protein-interaction networks, it provides insight into the efficiency of biological processes [13], and in computing, it determines the length of the critical path [20].

The diameter is typically computed using the *All-Pairs Shortest-Path* (APSP) algorithm [3], which can be broken down into many *Single-Source Shortest Path* (SSSP) problems [15], which amount to *Breadth-First Searches* (BFS) on unweighted graphs. These approaches run in $O(nm)$ time on graphs with n vertices and m edges. However, real-world graphs often contain millions of vertices and edges or more, making the APSP approach impractical. Some algorithms specialize in specific classes of graphs [21] or assume the graph to be directed or weighted [5]. Our work targets undirected, unweighted, sparse graphs as is done in the majority of the related work. Since the diameter of a disconnected graph is infinite, our implementation outputs infinity as well as the diameter of the largest connected component in case of a disconnected input.

Prior work includes various optimizations to minimize the number of BFS calls, that is, to improve performance. A common approach is to update lower and upper bounds of eccentricities across the graph as the computation progresses. When the upper and lower bounds become equal, the program terminates. However, these updates can be costly as each of them generally involves a traversal of the entire graph. Moreover, the worst case still requires $O(nm)$ work.

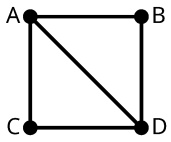

Figure 1: Example of a graph with a diameter of 2

In this paper, we present the F-Diam algorithm as well as our parallel CPU implementation thereof. F-Diam is a new algorithm for computing the exact diameter of large graphs. It incorporates novel optimizations, including Winnowing and Chain Processing, to minimize the amount of work performed. Based on the result of just 2 initial BFS calls to compute the eccentricity of a couple of vertices, Winnowing eliminates a set of vertices whose eccentricities do not need to be evaluated. This set often covers over 90% of the graph. Winnowing is accomplished using only a partial BFS

ICPP '25, San Diego, CA, USA

ACM ISBN 979-8-4007-2074-1/25/09
https://doi.org/10.1145/3754598.3754620

rather than traversing the entire graph. Next, F-Diam computes the eccentricities of some of the remaining vertices. With every new highest eccentricity that it finds, it incrementally winnows and eliminates even more vertices. It terminates when no vertices are left. Its worst-case time complexity is $O(nm)$, i.e., the same as that of the state of the art. However, in practice, F-Diam consistently provides a large speedup compared to prior work.

This paper makes the following main contributions.

- It introduces F-Diam, an exact diameter-finding algorithm that incorporates the novel Winnowing and Chain Processing techniques to minimize graph traversals.
- It describes key optimizations, including incremental extension of the set of vertices whose eccentricity does not need to be computed.
- It shows that our parallel CPU code outperforms the fastest prior diameter-finding codes on all tested inputs, in many cases by orders of magnitude.

Our OpenMP implementation is freely available in open source at https://github.com/burtscher/F-Diam.git

The rest of the paper is organized as follows. Section 2 summarizes related work. Section 3 describes background information. Section 4 details the implementation of our approach. Section 5 discusses our experimental methodology. Section 6 presents and analyzes the results. Section 7 concludes the paper with a summary.

2 Related Work

One of the first public implementations for computing the diameter of a graph is iFUB [4]. The algorithm starts with a random vertex or the vertex with the highest degree. From this starting vertex, 2 BFS calls are performed to find 2 vertices that are maximally far apart from each other. Along the path between the two, a third vertex is selected from where this process is started again. Totaling 4 BFS calls, a final "central" vertex is found whose eccentricity is close to the *radius* of the graph, where the radius refers to the smallest eccentricity. This 4-SWEEP algorithm is used to determine a lower bound for the diameter. Next, iFUB calculates the eccentricities of vertices in groups called fringe sets, which are vertices at a certain distance from the starting vertex. Based on the distances in the BFS tree, the algorithm can terminate early and set upper bounds for the eccentricity of vertices in fringe sets, limiting the number of BFS calls. We only use 2 BFS calls to determine the initial bound for the diameter and update it dynamically. Moreover, we introduce Winnowing to eliminate vertices, which we found to be more effective than early termination and fringe sets.

Pennycuff et al. [14] propose a new algorithm to measure the diameter of an unweighted graph using vertex-centric programming. Their parallel implementation computes the eccentricity of every vertex simultaneously by assigning a thread to each vertex and sending messages to the vertex's neighbors. Each message includes a history to prevent it from being re-propagated through the graph. When no further messages are propagated, the largest eccentricity has been computed. The authors mention that their approach runs out of memory for larger graphs and that, for smaller graphs, a non-parallelized APSP version is faster. Although their vertex-centric approach provides large amounts of parallelism, it is impractical for many real-world graphs due to memory constraints.

The Graph-Diameter work [1] presents a novel approach for computing the diameter of directed graphs. The algorithm uses a double sweep, starting from a random vertex, to find vertices on the opposite ends of a shortest path. This yields an initial lower bound of the diameter. An upper bound is computed using the triangle inequality in strongly connected components, which asserts that, for any pair of vertices x and y, $ecc(x) \leq d(x, y) + ecc(y)$, where $d(x, y)$ is the length of the shortest path from x to y and $ecc(v)$ is the eccentricity of vertex v. The algorithm maintains an upper bound on the eccentricity for each vertex and updates it with further BFS traversals of the graph, skipping vertices whose upper bounds are less than the lower bound of the diameter. F-Diam includes some of these ideas but eliminates vertices using partial BFS traversals.

Takafuji et al. [18] introduce a GPU version of the blocked Floyd-Warshall algorithm to compute the length of the shortest path between all pairs of vertices in a directed graph. They call this the "single kernel" implementation. They report a speedup of 1.02× to 1.09× on graphs with up to 32768 vertices. They accomplish the single-kernel implementation by partitioning the adjacency matrix used to represent the graph into tiles. A CUDA block is thus able to execute the parallel Floyd-Warshall algorithm for a number of tiles in stages without synchronization. They show higher speedups for smaller graphs. In contrast, F-Diam uses the compressed-sparse-row (CSR) representation [7] to fit sparse graphs with many millions of vertices and edges into the main memory.

Korf [10] presents a novel algorithm for finding the diameter of an undirected graph using partial BFS traversals. This approach takes advantage of the observation that larger eccentricities can only be found between two vertices that have not been starting vertices of earlier BFS calls. This involves maintaining a set S of active vertices. Each BFS traversal terminates as soon as all vertices in S have been visited. Upon termination, the starting vertex is removed from S. A parallel version of this algorithm was found to deliver speedups of up to 5× over complete BFS traversals. We do not use this optimization because we found early termination to hurt performance as it conflicts with our new techniques.

We have implemented, empirically evaluated, and adopted several of the aforementioned state-of-the-art optimizations. However, they do not sufficiently address the large number of BFS calls. To further reduce the number of these calls in an efficient and parallelizable manner, we devised Winnowing and Chain Processing.

3 Background

In this paper, we consider graphs $G = (V, E)$ containing $|V| = n$ vertices and $|E| = m$ edges that are undirected and unweighted. When describing the distance $d(x, y)$ between two vertices $x, y \in V$, we refer to the number of *edges* that must minimally be traversed to reach one vertex from the other. We call each traversal of an edge a *step*. For example, if the two vertices are adjacent, they are 1 step away from each other, meaning their distance is 1.

We denote the eccentricity of a vertex $v \in V$ by $ecc(v)$. As mentioned, the eccentricity is defined as the largest shortest distance, that is, $ecc(v) = max_{x \in V}(d(v, x))$. The diameter of a graph is the maximum eccentricity, that is, $diam(G) = max_{v \in V}(ecc(v))$. Vertices with eccentricities close to the diameter represent the graph's

periphery. Since the diameter of a disconnected graph is infinite, we focus on connected graphs in this background section.

The following paragraphs describe known theorems about eccentricities. We exploit them to find vertices whose eccentricity does not need to be computed while still guaranteeing that we will find the largest eccentricity, i.e., the true diameter of the graph.

Theorem 1. *The eccentricity of two adjacent vertices $x, y \in V$ cannot differ by more than* 1*, that is, $ecc(y)-1 \leq ecc(x) \leq ecc(y)+1$ and $ecc(x) - 1 \leq ecc(y) \leq ecc(x) + 1$.*

Proof: Since x and y are adjacent, $d(x, y) = 1$. Given that G is a connected graph, for all $v \in V$, we can find a path from x to v through y by combining the two shortest paths $x \rightarrow y$ and $y \rightarrow v$, which have lengths $d(x, y)$ and $d(y, v)$, respectively. Since $d(x, v)$ is the length of the shortest path between x and v, we know that $d(x, v) \leq d(x, y) + d(y, v) = 1 + d(y, v)$. By definition, $d(y, v) \leq ecc(y)$. Adding 1 on both sides, we find $1 + d(y, v) \leq 1 + ecc(y)$. Hence, $d(x, v) \leq 1 + d(y, v) \leq 1 + ecc(y)$ must hold for all $v \in V$, including the v that is the farthest away from x, in which case $d(x, v) = ecc(x)$. Putting everything together, we obtain $ecc(x) \leq 1 + ecc(y)$. Subtracting 1 on both sides yields $ecc(x) - 1 \leq ecc(y)$. Symmetrically, by swapping x and y, we get $ecc(y) \leq 1 + ecc(x)$ and $ecc(y) - 1 \leq ecc(x)$, proving Theorem (1).

Once the eccentricity of a vertex v has been determined, we can use Theorem (1) to compute an upper bound of the eccentricity of every other vertex z. Specifically, if z is s steps away from v, the eccentricity of z must be $ecc(z) \leq ecc(v) + s$. We use this fact to discard all vertices from consideration[1] whose upper bound is less than or equal to the largest eccentricity computed so far, as they cannot yield a higher eccentricity.

Theorem 2. *Any connected graph G with at least two vertices has two or more vertices with maximum eccentricity, that is, $\exists x, y \in V$ such that $x \neq y$ and $ecc(x) = ecc(y) = diam(G)$.*

Proof: A connected graph with at least two vertices contains at least one edge. The end-point vertices of this edge are adjacent and have a distance of 1 (see above). Therefore, the diameter of the graph must be at least 1. Since the diameter is the largest eccentricity of any vertex in the graph, there must be at least one vertex x for which $ecc(x) = diam(G)$. Let y be a vertex that is $ecc(x)$ steps away from x. Per the definition of the eccentricity, such a vertex must exist. Moreover, y cannot be x as $d(x, x) = 0$ and we already established that the diameter is at least 1. Since the graph is undirected, $d(x, y) = d(y, x)$ holds, meaning that $ecc(y)$ is at least as large as $ecc(x)$. But $ecc(x)$ is the largest eccentricity in the graph. Therefore, $ecc(y) = ecc(x)$ must hold, and we found a second vertex with the highest eccentricity, proving Theorem (2).

We use Theorem (2) to safely discard vertices from consideration for which we can guarantee that another vertex with the same eccentricity is still being considered. This is the key new insight behind our Winnowing technique. It is more powerful than prior approaches because it is the first technique that can safely discard some vertices with eccentricities that are *higher* than the current bound for the diameter.

[1]Throughout this paper, "removing a vertex from consideration" means we do not have to compute its eccentricity, but the vertex remains in the graph and can be traversed during the eccentricity calculations of other vertices.

Theorem 3. *The smallest eccentricity in any connected graph is at least half of the diameter, that is, $min_{v \in V}(ecc(v)) \geq diam(G)/2$.*

Proof: Assume we have a vertex x whose eccentricity is less than half of the diameter d. This means all vertices can be reached from x in fewer than $d/2$ steps. Since we consider undirected graphs, this also means that all vertices can reach x in fewer than $d/2$ steps. Therefore, all vertices can reach all other vertices via x in fewer than $d/2 + d/2 = d$ steps, contradicting that d is the diameter of the graph. Hence, no vertex with an eccentricity of less than half the diameter can exist, proving Theorem (3).

Winnowing uses Theorem (3) in combination with Theorem (2) to discard all vertices from consideration that are within $d/2$ steps of an arbitrary vertex u. This approach is most beneficial if u has the smallest eccentricity (called the *radius* of the graph), that is, if u is a "center" vertex. Finding such a vertex is as expensive as computing the diameter. Instead, we use the highest-degree vertex as it tends to be centrally located. For the same reason, the highest-degree vertex is sometimes used as the starting vertex in related work [4] and as a potentially central vertex in studies on core-periphery structures in networks [16].

In the rest of this paper, we denote the highest-degree vertex as u. It is important to note that high-degree vertices tend to be core vertices in the core-periphery structure of the graph and are some of the most "centrally" located [23] (in the sense of, for example, betweenness centrality [19]). As a consequence, they typically have some of the smallest eccentricities. Conversely, vertices with a low degree and, in particular, vertices with degree 1 tend to be on the "periphery" of a graph and are likely to have some of the highest eccentricities. Our Chain Processing targets such degree-1 vertices.

4 Approach

This section describes our F-Diam algorithm and its parallel implementation in detail. Our code computes the eccentricity of a vertex v by performing a parallel level-synchronous BFS starting from v and counting the number of levels. Section 4.6 outlines how this works. Doing so for all vertices in the graph to find the largest eccentricity (i.e., the diameter) would be prohibitively slow. Instead, F-Diam employs various optimizations to minimize the number of vertices whose eccentricity needs to be computed. This strategy is outlined in Algorithm 1 along with the following subsections explaining our optimizations, how they are put together in F-Diam, and how they are parallelized.

Note that Algorithm 1 and the following algorithms use a *counter* value to check whether a vertex has already been visited in the current iteration. We use a counter rather than a flag to avoid a costly reset procedure after each BFS traversal. Note further that any write to a vertex's eccentricity (e.g., v_{ecc}) not only sets the eccentricity but also removes the vertex from consideration.

Algorithm 1 outlines the overall procedure. First, we compute an initial bound using the 2-sweep approximation with u as the starting vertex. In this process, we compute the eccentricity using Algorithm 2 of both u and a vertex w that is as far away as possible from u. After computing the initial lower bound, we call Winnow (Algorithm 3) and Chain-processing (Algorithm 4) to remove a large number of vertices from consideration. Then, we compute the eccentricities of some of the remaining vertices via BFS calls.

Algorithm 1 Fast Exact Diameter Algorithm

Require: graph G, max-degree vertex u, int *counter*

```
1:  u_ecc ← BFS(G, u, counter)
2:  w ← wl1[0]                                  // farthest vertex from u
3:  bound ← BFS(G, w, counter)
4:  Winnow(G, u, bound, counter)
5:  Chain(G, counter)
6:  v ← 0
7:  while v < G.size do
8:    while v_ecc ≠ active do
9:      v ← v + 1
10:   if v = G.size then
11:     break
12:   v_ecc ← BFS(G, v, counter)
13:   if v_ecc > bound then
14:     old ← bound
15:     bound ← v_ecc
16:     Winnow(G, u, bound, counter)
17:     for i ∈ G do
18:       if i_ecc = old then
19:         Eliminate(G, i, old, bound, counter)
20:   else
21:     Eliminate(G, v, v_ecc, bound, counter)
22: diameter ← bound
```

When a new bound for the diameter is found, we call our Eliminate function (Algorithm 5) from all vertices that have an eccentricity equal to the previous bound. Otherwise, we eliminate the vertices around v that cannot yield a new bound.

4.1 Initial Diameter

The first step of F-Diam is to compute an initial lower bound of the diameter, which we call *bound*. In theory, we could start from any vertex, compute its eccentricity, and use that as the bound. However, we want this bound to be as close to the actual diameter as possible. F-Diam employs the following strategy to achieve this. It starts with the highest-degree vertex u and computes u's eccentricity, $ecc(u)$. Recall that this vertex is likely to be centrally located and to have a low eccentricity. So, $ecc(u)$ is not a good approximation of the bound we seek. Instead, we pick a vertex v from the last iteration of the BFS that computed $ecc(u)$. Vertex v is maximally far away from u and, therefore, likely on the "periphery" of the graph. Next, we compute $ecc(v)$ and use it as our initial lower bound for the diameter as is done in some related work.

Performing these operations on the example graph shown in Figure 2 yields the following. First, we identify the highest-degree vertex, which is i. Then, we compute its eccentricity, which is 4. Next, we take one of the vertices with maximal distance from i, for example vertex d. Finally, we compute $ecc(d)$, which is 6 because vertex m is 6 steps away from vertex d. We use this eccentricity as our initial lower bound for the diameter, that is, we set *bound* to 6. For reference, we also show the eccentricities in Figure 2.

Algorithm 2 BFS Traversal to Compute Eccentricity

Require: graph G, vertex *source*, int *counter*

```
1:  counter ← counter + 1
2:  source_cnt ← counter
3:  wl1 ← {source}
4:  wl2 ← ∅
5:  level ← 0
6:  while wl1 ≠ ∅ do
7:    level ← level + 1
8:    ratio ← wl1.size/|V|
9:    if ratio ≤ 0.1 then
10:     for each vertex v ∈ wl1 do                  // parallel loop
11:       for each vertex n ∈ adj(v) do
12:         if n_cnt ≠ counter then
13:           n_cnt ← counter
14:           wl2 ← wl2 ∪ {n}
15:   else
16:     for each vertex v ∈ G do                    // parallel loop
17:       if v_cnt ≠ counter do
18:         for each vertex n ∈ adj(v) do
19:           if n_cnt = counter do
20:             wl2 ← wl2 ∪ {v}
21:             break
22:     for each vertex v ∈ wl2 do                  // parallel loop
23:       v_cnt ← counter
24:   wl1 ← ∅
25:   swap wl1 with wl2
26: return level − 1
```

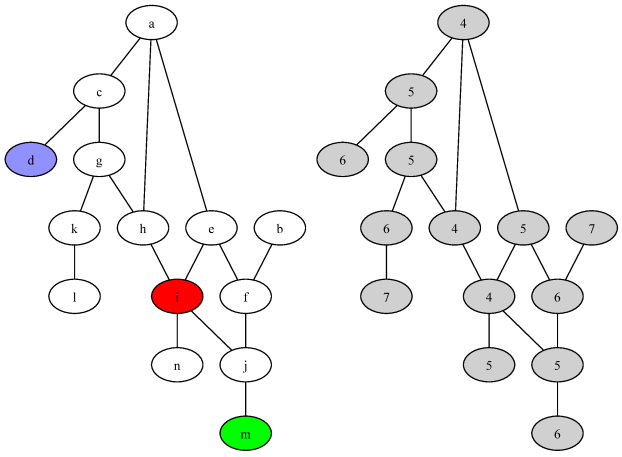

Figure 2: Example of finding the initial diameter (left); the eccentricity of each vertex (right)

4.2 Winnow

The key novelty in F-Diam is that it exploits Theorem (2) to remove, in parallel, a large number of vertices from consideration. Given the current lower bound on the diameter (*bound*) and an arbitrary starting vertex v, it *winnows* all vertices that lie within a distance of $\lfloor bound/2 \rfloor$ from v. Assume w to be such a vertex. This vertex can

reach all other vertices in the winnowed region in no more than *bound* steps because it can reach v in $\lfloor bound/2 \rfloor$ steps, from where all other vertices in the region can be reached in another $\lfloor bound/2 \rfloor$ steps. For the same reason, all other vertices in the winnowed region can reach w in no more than *bound* steps. Therefore, if a pair of vertices exist that are more than *bound* steps apart, at least one of those two vertices must be outside of the winnowed region. If no such pair exists, we have already found the true diameter. Hence, it is safe to winnow all vertices within a distance of $\lfloor bound/2 \rfloor$ from v in all cases. Algorithm 3 outlines how F-Diam implements this winnowing using a partial BFS traversal, where we parallelize the outer *for each* loop using atomic operations.

Algorithm 3 Winnow Operation

Require: graph G, vertex *source*, int *bound*, int *counter*

```
1:  counter ← counter + 1
2:  source_cnt ← counter
3:  wl1 ← {source}
4:  wl2 ← ∅
5:  level ← 0
6:  while level < bound do
7:      level ← level + 1
8:      for each vertex v ∈ wl1 do                    // parallel loop
9:          for each vertex n ∈ adj(v) do
10:             if n_cnt ≠ counter then
11:                 n_cnt ← counter
12:                 n_ecc ← winnowed                  // eliminates vertex n
13:                 wl2 ← wl2 ∪ {n}
14:     wl1 ← ∅
15:     swap wl1 with wl2
```

Winnowing is particularly effective when v is centrally located since central vertices maximize the number of vertices in the winnowed region. In contrast, a vertex on the "periphery" would likely have fewer vertices in the region because it does not have neighbors "in some directions". As a consequence, F-Diam uses the highest-degree vertex u as the starting vertex for winnowing, which tends to be centrally located as discussed before.

It is critical to note that winnowing can only be done from one vertex. If we also winnowed from an additional vertex, we would lose the guarantee that at least one of the two vertices with maximum eccentricity, based on Theorem (2), is still being considered.

We found winnowing to be relatively low cost in comparison to more complex approaches such as fringe sets. After the first invocation, the winnow function is only called when a new bound for the diameter is found that is larger than the old estimate by at least 2 so that $bound/2$ increases by at least 1. We have experimentally found our initial diameter to often be very close to the exact diameter, which is why F-Diam rarely calls winnow multiple times for a given input. Since it tends to remove a majority of the vertices in a single partial BFS call, winnow can be very effective.

Figure 3 provides an example of winnowing. The highest-degree vertex is c. For reference, the two vertices with the largest eccentricity are h and j (but the algorithm does not know this yet). Assuming *bound* to be 5, we can remove all vertices that are no more than $\lfloor 5/2 \rfloor = 2$ steps away from vertex c from consideration. Note that this includes one of the vertices with maximum eccentricity (h), but that is okay because we keep the other such vertex (j) active.

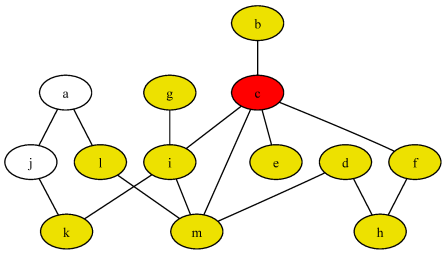

Figure 3: Example of Winnow

4.3 Chain Processing

Another novelty of F-Diam is the processing of what we call chains. Many graphs have some degree-1 vertices. For any such vertex x that is adjacent to a vertex y, every shortest path starting in x must pass through y. Hence, whatever the distance from y to some other vertex z is, the distance from x to z must be one step longer. This implies that the eccentricity of x must be one larger than the eccentricity of y (assuming the graph has more than one edge). As a consequence, we can safely remove all "y" vertices that have a degree-1 neighbor from consideration. In fact, we can go further and also remove all direct neighbors of y (other than x) because, according to Theorem (1), those neighbors cannot have a larger eccentricity than x. So, keeping just x suffices. Note that this can be done without computing a single eccentricity.

F-Diam incorporates an extended version of this idea. If the degree-1 vertex x is adjacent to a degree-2 vertex, which may be adjacent to another degree-2 vertex, etc., we follow this *chain*, which looks like a linked list, until we reach a vertex w whose degree is not 2. Moreover, we keep track of the length of the chain. Assuming this length is s and applying the same reasoning as above, we find that $ecc(w)$ must be equal to $ecc(x) - s$ if there exists at least one other vertex z that is also at a distance of s from w.

Interestingly, this means we can safely remove all vertices (other than x) from consideration that lie within a distance of s from w, irrespective of whether such a vertex z exists. If z does exist, $ecc(w) = ecc(x) - s$ holds and removing those vertices is safe. If no such vertex z exists, the depth d of the subgraph rooted in w (excluding the linked list) must be less than s. This means all vertices in the subgraph can reach each other in no more than $2d$ steps (via w). However, the farthest of those vertices are $d+s$ steps away from x (again via w). Since $s > d$, that means x must have the highest eccentricity of the entire graph in this case. Therefore, it is again safe to remove all other vertices from consideration. Algorithm 4 outlines how this works. The constant MAX is $INT\_MAX - 1$. F-Diam treats vertices with eccentricities less than INT_MAX as having been removed from consideration (i.e., as inactive). The Eliminate function is defined in the next subsection.

Figure 4 shows an example of which vertices can be removed from consideration using the Chain Processing technique (without any winnowed vertices). There are two chains, one starting at vertex e and another starting at vertex g. The first chain is 2 edges long.

Algorithm 4 Chain Processing

Require: Graph G, int *counter*
1: **for** each vertex $v \in G$ **do**
2: **if** $deg(v) = 1$ **then**
3: $n \leftarrow adj(v)$
4: $len \leftarrow 1$
5: **while** $deg(n) = 2$ **do**
6: $n \leftarrow adj(n)$ // forward direction
7: $len \leftarrow len + 1$
8: Eliminate(G, n, $MAX - len$, MAX, *counter*)
9: $v_{ecc} \leftarrow active$

Hence, all vertices along the chain (other than the starting vertex e) can be removed from consideration, as can all vertices up to 2 steps away from vertex c, where the chain ends. The chain starting at g is only 1 edge long and ends in i, so we can only remove i and its direct neighbors (other than g). In this example, Chain Processing removes a majority of the vertices, but this is not usually the case. Nevertheless, it generally removes vertices that are out of the reach of Winnow and, therefore, complements Winnow well.

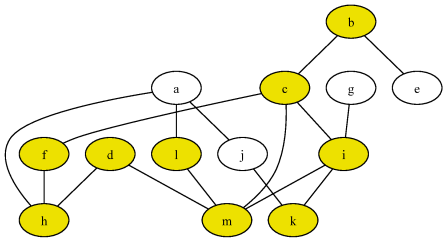

Figure 4: Example of eliminating the chains starting in e & g

4.4 Eliminate

If any active vertices are left after Winnow and Chain Processing, F-Diam randomly picks such a vertex x and computes its eccentricity. If the eccentricity turns out to be greater than *bound*, F-Diam updates *bound* to the new value. Otherwise, we employ Theorem (1) to *eliminate* some vertices from consideration. For instance, if $s = bound - ecc(x) > 0$, the theorem tells us that all vertices that are no more than s steps away from vertex x must have an eccentricity that is no larger than *bound*. Consequently, there is no need to compute their eccentricities as they cannot change the value of *bound*. Algorithm 5 outlines how this works. Since this code tends to only execute a couple of iterations with just a few elements on the worklist, F-Diam runs it serially as there is typically not enough work to warrant parallelization. Note that Eliminate is a known technique that is, for example, used in Graph-diameter [1], which also runs it serially. The number of BFS calls eliminated is equal to the number of active vertices reachable from x in s steps. Whereas other approaches update the bounds of all vertices in the same connected component with a similar method, F-Diam utilizes a much faster partial-BFS to eliminate vertices that are known to not change the value of *bound*.

Figure 5 provides an example. Assuming that *bound* is currently 5 and that the eccentricity of vertex c is 4, we can eliminate all

Algorithm 5 Eliminate Operation

Require: graph G, vertex *source*, int *ecc*, int *bound*, int *counter*
1: $wl1 \leftarrow \{source\}$
2: $wl2 \leftarrow \emptyset$
3: $counter \leftarrow counter + 1$
4: **while** $ecc < bound$ **do**
5: $ecc \leftarrow ecc + 1$
6: **for** each vertex $v \in wl1$ **do**
7: **for** each vertex $n \in adj(v)$ **do**
8: **if** $n_{cnt} \neq counter$ **then**
9: $n_{cnt} \leftarrow counter$
10: $n_{ecc} \leftarrow ecc$ // sets ecc and eliminates vertex n
11: $wl2 \leftarrow wl2 \cup \{n\}$
12: $wl1 \leftarrow \emptyset$
13: swap $wl1$ with $wl2$

neighbors of c from consideration that are no more than $5 - 4 = 1$ steps away from c. In other words, we can eliminate the direct neighbors of c in this example, which are highlighted in yellow.

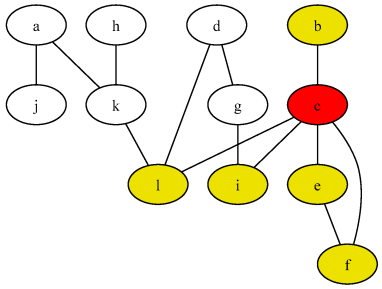

Figure 5: Example of Eliminate

4.5 Extending Winnow and Eliminate

F-Diam, like many related algorithms, starts out by first computing an initial lower bound of the diameter (*bound*). Since the winnowing and elimination steps are more effective for larger bounds, it tries to find a large bound as outlined in Section 4.1. This requires first determining the highest-degree vertex u in the input graph.

Next, F-Diam performs winnowing around vertex u using *bound* as described in Section 4.2. For many graphs, this winnows the majority of the vertices. Based on Theorem (3), we know that Winnow reaches at least as many vertices as Eliminate. Hence, F-Diam does not call Eliminate on vertex u. Then, it processes the chains as explained in Section 4.3. The Chain Processing technique is often able to remove vertices that are out of reach of Winnow and Eliminate. Finally, F-Diam loops over the remaining vertices in random order. For each vertex v, it performs a BFS to compute the eccentricity. If the eccentricity of v is lower than *bound*, F-Diam calls Eliminate as described in Section 4.4. If it is equal to *bound*, F-Diam only eliminates v. If the eccentricity is larger than *bound*, F-Diam just found a new lower bound for the diameter. In this case, it updates *bound*, *extends* the winnowed region around u, and *extends* the regions of eliminated vertices as follows.

Incrementally extending the winnowed region is trivial as it is centered around one starting vertex. Extending all prior eliminated regions is more involved. Since we just found a new largest eccentricity, all previously computed eccentricities must be smaller. Therefore, we theoretically need to call Eliminate on all prior vertices for which an eccentricity was computed, which would be slow. F-Diam avoids this by recording the upper bound for the eccentricity in each eliminated vertex. This allows it to place all vertices with an eccentricity bound that is equal to the old *bound* value onto a worklist and then performs a single, partial, multi-source, level-synchronous BFS. It is partial because we only eliminate as many levels as the difference between the new and the old *bound* values. Hence, incrementally extending all prior eliminated regions is efficient and independent of the number of prior evaluated vertices.

F-Diam repeats the above steps until each vertex has either been eliminated or its eccentricity computed. The final *bound* value is the true diameter of the graph.

4.6 Parallelization

F-Diam often eliminates a majority of the vertices due to a combination of the previously mentioned techniques. However, the number of removed vertices is dependent on the topology of the input. In the worst case, which is a graph where all vertices have the same eccentricity, the complexity of F-Diam is $O(nm)$ because the Chain Processing and Eliminate techniques do not apply and Winnow can only remove fewer than half of the vertices. Hence, most vertices still need to have their eccentricity computed explicitly.

We parallelized our implementation to boost the performance of F-Diam even on such inputs. Specifically, every level-synchronous BFS traversal of the entire graph is done in parallel. As an alternative, we also tried running multiple BFS traversals in parallel. However, this did not yield a speedup because it resulted in too much redundant work, as concurrent Eliminate operations would overlap in removing vertices from consideration. After all, our approach tends to require only a few eccentricity evaluations (see Section 6.3), and it is difficult to predict which vertices need to have their eccentricity computed.

In our parallel code, each thread is assigned a chunk of vertices from the current worklist. The threads then visit the neighbors of those vertices in parallel and atomically check if these neighbors have already been visited. Neighbors that have not are atomically added to the second worklist. All threads synchronize at a barrier before the two worklists are swapped and the procedure repeats. The traversal ends when the new worklist is empty.

F-Diam employs state-of-the-art direction-optimized BFS [2, 22] to reduce the number of edges examined. This hybrid approach switches between a conventional data-driven top-down BFS and a topology-driven bottom-up BFS when the worklist exceeds a size threshold that is computed based on the number of vertices in the graph. The bottom-up BFS requires no atomic operations but performs some wasted work, which is why it is only preferable when the worklist size is large. F-Diam switches back to the top-down BFS when the frontier size drops below the threshold again in line with the latest direction-optimized BFS implementations [22].

We experimentally determined a threshold of 10% of the number of vertices to yield good performance. Once the worklist size reaches this threshold, the following frontier consists of the unvisited neighbors of these vertices, which is often close to 50% of the graph, making the bottom-up BFS very effective.

5 Evaluation Methodology

We compare the performance of a serial and a parallel implementation of F-Diam with serial Graph-Diameter [1] and serial and parallel implementations of iFUB [4]. These are the fastest publicly available implementations we could find. Although Graph-Diameter was developed for directed graphs, it also works on undirected graphs in CSR format, in which each undirected edge is represented by two directed edges in opposite directions.

We use the 17 real-world and synthetic graphs from Table 1 as inputs. The table lists the graph name, origin, type, number of vertices, number of edges (including back edges), average degree, maximum degree, and the largest eccentricity in any connected component. Several of these graphs are disconnected, meaning the actual diameter is infinite. This is flagged in the output of our code. F-Diam and all other tested codes support disconnected graphs and report the largest eccentricity among all connected components.

We choose these inputs for their variety in topology, degree, size, type, and diameter. They stem from the Galois framework [9], Stanford Network Analysis Platform [11], SuiteSparse Matrix Collection [17], and the Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome [6].

We evaluated all codes on a shared-memory NUMA system with an AMD Ryzen Threadripper 3970X 32-core processor. We compiled the codes using *g++* version 13.3.0, which supports OpenMP 5.2.

We run the codes 9 times on each input and use the median runtime. This is done to account for variability in system performance. Moreover, we limited the running time to 2.5 hours per input. We primarily report performance in terms of throughput, which is the number of vertices in the graph divided by the running time. Doing so normalizes the results as the graphs vary greatly in size and, therefore, in runtime. Throughput is a higher-is-better metric.

6 Results

In this section, we first compare the performance of F-Diam with the state of the art across different graph topologies. Then, we separately evaluate the performance of various features of F-Diam.

6.1 Performance Comparison

Figure 6 shows the throughput (in vertices per second) of our serial and parallel CPU codes as well as that of iFUB and Graph-Diameter. Note that the y-axis is logarithmic. The x-axis lists the input graphs. iFUB times out on the *2d-2e20.sym*, *cit-Patents*, *delaunay_n24*, *europe_osm*, *kron_g500-logn21*, *uk-2002*, *USA-road-d.NY*, and *USA-road-d.USA* inputs. Graph-Diameter times out on *delaunay_n24*. Table 2 lists the corresponding runtimes.

The serial code of F-Diam already outperforms iFUB and Graph-Diameter on all tested inputs. In fact, it is nearly an order of magnitude faster in most cases. Our parallel code is even faster and outperforms our serial version on each input.

Based on the geometric-mean throughputs, the serial implementation of F-Diam is 1267.0× faster than serial iFUB, 686.4× faster

Table 1: Information about the input graphs

name	origin	type	vertices	edges	avg degree	max degree	CC diameter
2d-2e20.sym	Lonestar	grid	1,048,576	4,190,208	4.0	4	2,046
amazon0601	SNAP	product co-purchases	403,394	4,886,816	12.1	2,752	25
as-skitter	SNAP	Internet topology	1,696,415	22,190,596	13.1	35,455	31
citationCiteseer	UoFSMC	publication citations	268,495	2,313,294	8.6	1,318	36
cit-Patents	UoFSMC	patent citations	3,774,768	33,037,894	8.8	793	26
coPapersDBLP	UoFSMC	publication citations	540,486	30,491,458	56.4	3,299	23
delaunay_n24	UoFSMC	triangulation	16,777,216	100,663,202	6.0	26	1,722
europe_osm	UoFSMC	road map	50,912,018	108,109,320	2.1	13	30,102
in-2004	UoFSMC	web links	1,382,908	27,182,946	19.7	21,869	43
internet	UoFSMC	Internet topology	124,651	387,240	3.1	151	30
kron_g500-logn21	UoFSMC	Kronecker	2,097,152	182,081,864	86.8	213,904	7
rmat16.sym	Lonestar	RMAT	65,536	967,866	14.8	569	14
rmat22.sym	Lonestar	RMAT	4,194,304	65,660,814	15.7	3,687	18
soc-LiveJournal1	SNAP	journal community	4,847,571	85,702,474	17.7	20,333	20
uk-2002	UoFSMC	web links	18,520,486	523,574,516	28.3	194,955	45
USA-road-d.NY	Dimacs9	road map	264,346	730,100	2.8	8	720
USA-road-d.USA	Dimacs9	road map	23,947,347	57,708,624	2.4	9	8,440

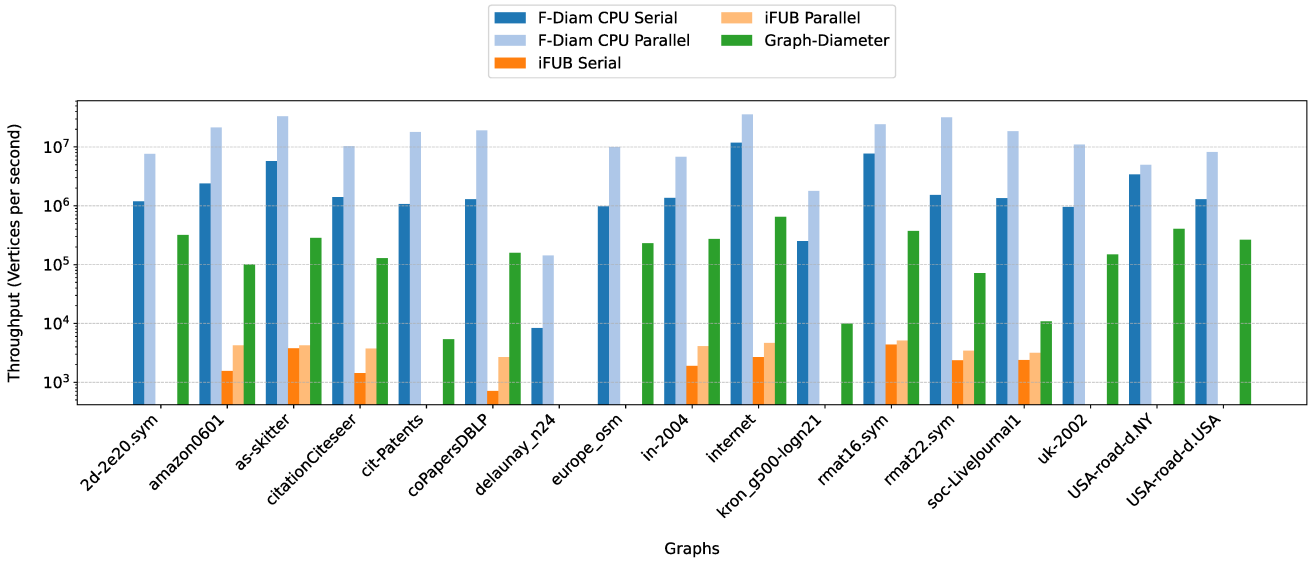

Figure 6: Throughput of various diameter codes on 17 inputs (missing bars denote timeouts)

Table 2: Measured runtimes in seconds (T/O = timeout)

Graphs	F-Diam (ser)	F-Diam (par)	iFUB (ser)	iFUB (par)	Graph-Diam.
2d-2e20.sym	0.885	0.138	T/O	T/O	3.285
amazon0601	0.169	0.019	259.004	94.916	3.983
as-skitter	0.296	0.051	451.391	402.688	5.959
citationCiteseer	0.192	0.026	187.226	71.575	2.098
cit-Patents	3.520	0.209	T/O	T/O	705.259
coPapersDBLP	0.417	0.028	761.575	203.028	3.426
delaunay_n24	2017.863	116.999	T/O	T/O	T/O
europe_osm	52.169	5.095	T/O	T/O	219.913
in-2004	1.018	0.204	728.197	336.903	5.098
internet	0.011	0.003	46.813	26.922	0.192
kron_g500-logn21	8.394	1.175	T/O	T/O	210.495
rmat16.sym	0.009	0.003	14.985	12.893	0.176
rmat22.sym	2.740	0.132	1772.274	1226.946	58.329
soc-LiveJournal1	3.610	0.262	2024.930	1541.236	448.948
uk-2002	19.369	1.690	T/O	T/O	123.839
USA-road-d.NY	0.077	0.053	T/O	T/O	0.650
USA-road-d.USA	18.548	2.914	T/O	T/O	90.976

than parallel iFUB, and 14.6× faster than Graph-Diameter[2]. On the worst measured input, F-Diam is still 560.9× faster than serial iFUB, 330.8× faster than parallel iFUB, and 3.71× faster than Graph-Diameter. On the best measured input, F-Diam is 4416.3× faster than serial iFUB, 2539.8× faster than parallel iFUB, and 200.3× faster than Graph-Diameter.

Our parallel code delivers a geometric-mean throughput that is 9518.8× and 5158.7× higher than iFUB's serial and parallel implementations, respectively, and 106.7× faster than Graph-Diameter's. In the worst measured case, parallel F-Diam is 3569.6×, 1651.5×, and 12.22× faster, and in the best measured case it is 26,816×, 9288.0× and 3368.0× faster than the serial and parallel iFUB implementations and Graph-Diameter, respectively.

[2]All speedups are computed based on the geometric-mean throughput over only the inputs on which neither code being compared times out.

The high performance of F-Diam is primarily due to Winnow. This new technique is very effective on all tested inputs (see Section 6.4), especially on "small world" graphs with low diameters and high maximum degrees, making F-Diam orders of magnitude faster than the leading prior approaches.

6.2 Scalability

Based on the geometric mean, our parallel implementation is 7.67× faster than our serial implementation. The smallest speedup we see with our parallel code is 1.45× and the largest speedup is 20.74× over our serial version. The reason for this difference in performance is the structure of the graphs. The lowest speedup stems from *USA-road-d.NY*, which is a small road map with an average degree of 3 and a maximum degree of 8. The highest speedup stems from *coPapersDBLP*, a power-law graph with an average degree of 56. This graph yields much more parallelism in each BFS call, resulting in better utilization of the multiple threads and a higher speedup. We also see this trend in other small-diameter inputs like *kron_g500-logn21*, where the worklist encompasses large portions of the graph. In contrast, in graphs with diameters as high as 30,102 (e.g., *europe_osm*), the worklist size never passes the threshold to utilize the bottom-up code in the direction-optimized BFS.

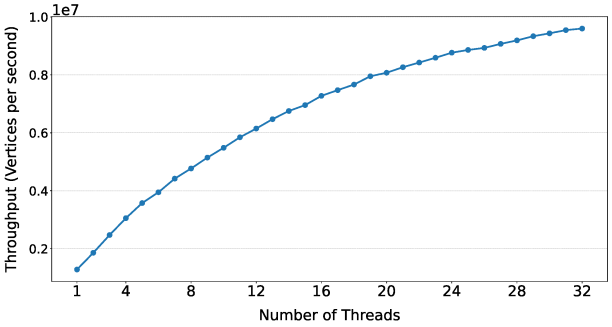

Figure 7: F-Diam throughput for different thread counts

Figure 7 shows the geometric mean of F-Diam's throughput over all inputs for different thread counts. The x-axis lists the number of threads and the y-axis the throughput. The performance increases up to 32 threads, which is the number of physical cores in our system. As mentioned, the geometric mean speedup is about 7.67× across all inputs. It is limited for two reasons. First, the BFS traversals start out with little parallelism and may end with little as well, especially on high-diameter inputs where the bottom-up BFS is not invoked. Second, the main-memory bandwidth does not scale with the core count on this irregular computation that incurs many cache misses.

6.3 Number of BFS Traversals

Table 3 lists the number of BFS traversals performed by the various codes on each input. We count a BFS traversal as either the computation of the eccentricity of a vertex or the use of the Winnow function. Although the Winnow function is a partial BFS, it typically traverses and eliminates a majority of the vertices, so the cost may be close to that of an entire BFS. In contrast, the Eliminate function typically only traverses a small portion of the graph, so we do not count it in the number of BFS calls.

Table 3: Number of BFS Traversals

Graphs	F-Diam	iFUB	Graph-Diameter
2d-2e20.sym	10	timeout	6
amazon0601	15	19	35
as-skitter	44	7	767
citationCiteseer	12	22	27
cit-Patents	788	timeout	4154
coPapersDBLP	11	38	10
delaunay_n24	3151	timeout	timeout
europe_osm	22	timeout	29
in-2004	102	15	122
internet	3	14	14
kron_g500-logn21	37	timeout	264
rmat16.sym	3	7	158
rmat22.sym	67	11	19285
soc-LiveJournal1	198	10	1172
uk-2002	481	timeout	1090
USA-road-d.NY	17	timeout	26
USA-road-d.USA	26	timeout	31

The number of traversals performed by any of the three codes is orders of magnitude lower than the number of vertices in the graphs, showing that the codes represent great improvements over a basic implementation that simply computes the eccentricity of every vertex. On most inputs, Graph-Diameter performs the highest number of traversals, explaining its lower throughput. On about half the inputs, iFUB performs fewer traversals than F-Diam. This is due to iFUB's use of fringe sets [4], which can result in fewer BFS calls but are expensive to maintain. Hence, the number of BFS traversals is not the main factor determining the overall performance.

6.4 Effectiveness of Stages in F-Diam

To shed more light on the effectiveness of the various stages used in F-Diam, Table 4 lists the percentage of vertices removed from consideration by the Winnow, Eliminate, and Chain Processing steps. We also show the percentage of vertices with no neighbors, which have an eccentricity of 0 and require no computation.

Both Eliminate and Chain Processing are only effective on some inputs. For example, inputs with no degree-1 vertices render Chain Processing useless. Even when degree-1 vertices are present, the function typically does not remove a large number of vertices, but it tends to get rid of vertices with relatively high eccentricities, which, as mentioned, are often out of reach of Winnow and Eliminate.

Eliminate also appears to be useless on many inputs. However, that is only because it runs after Winnow, which, in these cases, covers all vertices that Eliminate can remove. Our new Winnow technique is by far the most effective stage in F-Diam. It manages to remove over 70% of the vertices on all tested inputs. On over half of them, it eliminates more than 99% of the vertices that need computation. These results show that Winnow is the primary contributor of F-Diam's performance gain over the state of the art.

Figure 8 shows the fraction of F-Diam's overall runtime that each function contributes, as well as any other runtime. For all inputs, the few eccentricity computations take the majority of the runtime,

Table 4: Percentage of vertices removed from consideration by Winnow, Eliminate, and Chain as well as 0-degree vertices

Graphs	Winnow	Eliminate	Chain	Degree-0 Vertices
2d-2e20.sym	75.74%	24.25%	0.00%	0.00%
amazon0601	99.98%	0.01%	0.00%	0.00%
as-skitter	99.89%	0.00%	0.04%	0.00%
citationCiteseer	99.99%	0.00%	0.00%	0.00%
cit-Patents	99.72%	0.00%	0.15%	0.00%
coPapersDBLP	99.99%	0.00%	0.00%	0.00%
delaunay_n24	82.46%	17.53%	0.00%	0.00%
europe_osm	97.23%	0.85%	1.50%	0.00%
in-2004	97.89%	1.27%	0.83%	0.00%
internet	99.99%	0.00%	0.00%	0.00%
kron_g500-logn21	73.62%	0.00%	0.00%	26.37%
rmat16.sym	93.81%	0.00%	0.22%	5.72%
rmat22.sym	89.27%	0.00%	0.46%	9.76%
soc-LiveJournal1	99.92%	0.00%	0.02%	0.01%
uk-2002	99.67%	0.06%	0.05%	0.20%
USA-road-d.NY	98.79%	0.52%	0.67%	0.00%
USA-road-d.USA	71.11%	14.03%	14.23%	0.00%

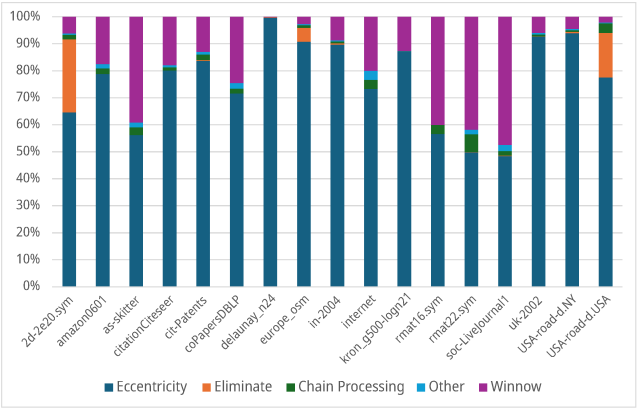

Figure 8: Percentage of runtime of each function in F-Diam

highlighting how inexpensive the other stages are in comparison. In particular, and despite its high effectiveness, Winnowing is fast.

6.5 Optimizations

Figure 9 shows the throughput of F-Diam and three more versions of F-Diam in which we disabled Winnow, Eliminate, or the use of the highest-degree vertex as the starting vertex. We only disable one feature at a time as disabling multiple together mostly results in timeouts. All versions are executed in parallel. As before, the throughput is shown on a logarithmic scale. Note that the missing bars indicate timeouts, that is, removing the feature results in a runtime of over two and a half hours on the given input.

Based on the geometric-mean throughputs, disabling the Winnow function has by far the largest impact on performance, causing F-Diam to run at a mere 2% of its normal speed, that is, two orders of magnitude slower. This highlights the importance and the benefit of our new technique. Changing the starting point from the maximum-degree vertex u to the vertex with ID zero causes the code to run at 17% of its usual speed. Clearly, starting out with a relatively central vertex is important as it boosts the effectiveness of the first Winnow and Eliminate calls. Finally, disabling the Eliminate function causes the code to run at 22% of its normal speed and times out on *delaunay_n24*, *europe_osm*, and *USA-road-d.USA*. Hence, the actual impact of disabling this feature is higher.

There are two graphs where changing the starting vertex from u to the first vertex in the input yields a speedup. In both cases, the number of BFS calls is lower or equal to what we see when all optimizations are enabled, as shown in Table 5. This suggests that the vertex with ID 0 is topologically closer to the center of the graph than vertex u with the maximum degree, which allows Winnow or Eliminate to prune more vertices. However, for most inputs, the number of BFS calls is significantly higher when disabling one of the tested features than in the baseline F-Diam version. In several cases, it is orders of magnitude higher, highlighting the importance of using Winnow, Eliminate, and u together.

Table 5: Number of BFS calls in different versions of F-Diam

Graphs	F-Diam	no Winnow	no Elim.	no 'u'
2d-2e20.sym	10	12	timeout	10
amazon0601	15	605	71	30
as-skitter	44	1382	92	44
citationCiteseer	12	432	12	24
cit-Patents	788	11234	984	2597
coPapersDBLP	11	491	13	44
delaunay_n24	3151	6351	timeout	4700
europe_osm	22	37	timeout	17
in-2004	102	161	17722	105
internet	3	3021	3	1088
kron_g500-logn21	37	28372	37	25348
rmat16.sym	3	2095	3	151
rmat22.sym	67	57374	68	277
soc-LiveJournal1	198	12465	633	203
uk-2002	481	962	12914	764
USA-road-d.NY	17	26	1407	91
USA-road-d.USA	26	47	timeout	105

7 Conclusion

The diameter of a graph is an important metric needed in various applications. Despite numerous enhancements to the basic approach of using the all-pairs shortest-path (APSP) algorithm for determining the diameter, the running time of the state-of-the-art solutions is often still impractically high for large sparse graphs. Our approach, called F-Diam, improves efficiency by minimizing the number of BFS calls through a set of pre-existing and new techniques. In particular, F-Diam introduces Winnow and Chain Removal to quickly remove vertices that do not need to be considered, which we show to be very effective in practice. We implemented F-Diam in both serial and parallel code that incorporates state-of-the-art direction-optimized BFS. Our experimental results demonstrate that both versions of F-Diam outperform the leading implementations by one to three orders of magnitude on a range of large real-world and synthetic graphs from different domains.

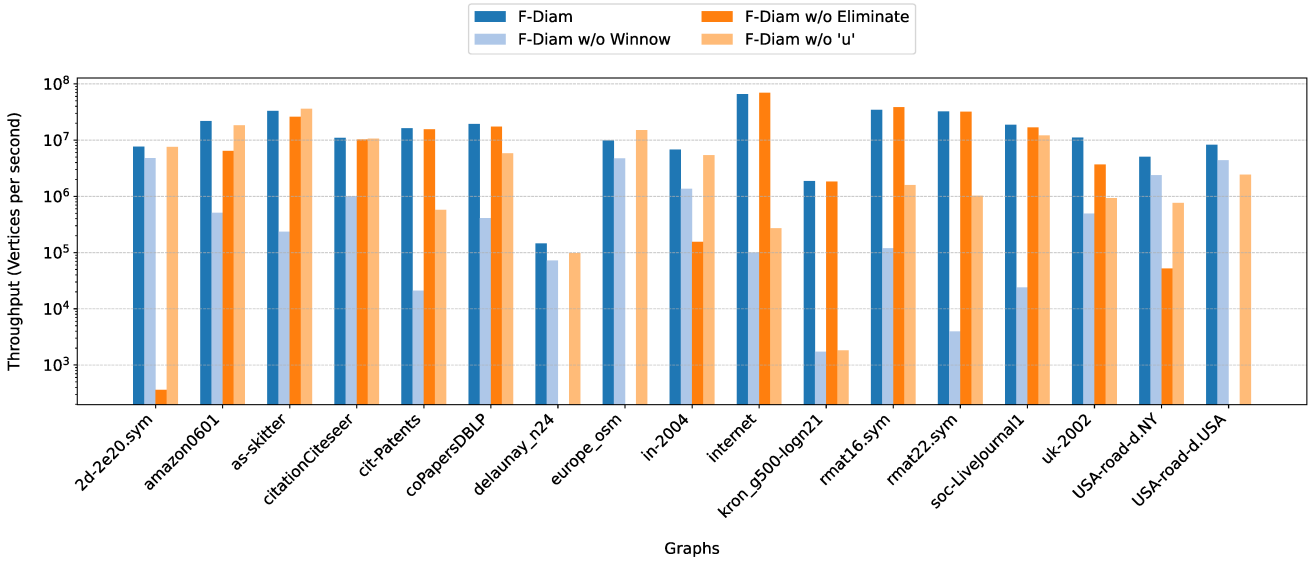

Figure 9: Throughput of various F-Diam versions (missing bars denote timeouts)

Acknowledgments

We thank the reviewers for their valuable feedback. This work has been supported by the National Science Foundation under Award #1955367.

References

[1] Takuya Akiba, Yoichi Iwata, and Yuki Kawata. 2015. An Exact Algorithm for Diameters of Large Real Directed Graphs. In *Experimental Algorithms*, Evripidis Bampis (Ed.). Springer International Publishing, Cham, 56–67.

[2] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10. https://doi.org/10.1109/SC.2012.50

[3] Fan RK Chung. 1987. Diameters of graphs: Old problems and new results. *Congressus Numerantium* 60, 2 (1987), 295–317.

[4] Pilu Crescenzi, Roberto Grossi, Michel Habib, Leonardo Lanzi, and Andrea Marino. 2013. On computing the diameter of real-world undirected graphs. *Theoretical Computer Science* 514 (2013), 84–95.

[5] Mina Dalirrooyfard and Nicole Wein. 2021. Tight conditional lower bounds for approximating diameter in directed graphs. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing* (Virtual, Italy) *(STOC 2021)*. Association for Computing Machinery, New York, NY, USA, 1697–1710. https://doi.org/10.1145/3406325.3451130

[6] DIMACS. 2006. Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome (DIMACS). https://www.diag.uniroma1.it/challenge9/download.shtml. Accessed: 2025-04-30.

[7] Jack Dongarra. 2000. Compressed row storage. http://www.netlib.org/utk/people/JackDongarra/etemplates/node373.html. Accessed: 2025-2-26.

[8] Leo Egghe and Ronald Rousseau. 2024. The small-world phenomenon: a model, explanations, characterizations and examples. arXiv:2402.10233 [cs.SI] https://arxiv.org/abs/2402.10233

[9] Galois. 2018. https://iss.oden.utexas.edu/?p=projects/galois/download

[10] Richard E. Korf. 2021. Finding the Exact Diameter of a Graph with Partial Breadth-First Searches. *Proceedings of the International Symposium on Combinatorial Search* 12, 1 (Jul. 2021), 73–78. https://doi.org/10.1609/socs.v12i1.18553

[11] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. Accessed: 2025-04-30.

[12] Behrooz Parhami and Chi-Hsiang Yeh. 2000. Why Network Diameter is Still Important. In *Proceedings of the International Conference on Communications in Computing (CIC-2000)* (Las Vegas, Nevada, USA). CSREA Press, 1–4. https://api.semanticscholar.org/CorpusID:16765597

[13] Georgios A Pavlopoulos, Maria Secrier, Charalampos N Moschopoulos, Theodoros G Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G Bagos. 2011. Using graph theory to analyze biological networks - biodata mining. https://biodatamining.biomedcentral.com/articles/10.1186/1756-0381-4-10

[14] Corey Pennycuff and Tim Weninger. 2015. Fast, exact graph diameter computation with vertex programming. In *Proceedings of the 1st High Performance Graph Mining Workshop (HPGM)*. https://api.semanticscholar.org/CorpusID:18390525

[15] Liam Roditty and Virginia Vassilevska Williams. 2013. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing* (Palo Alto, California, USA) *(STOC '13)*. Association for Computing Machinery, New York, NY, USA, 515–524. https://doi.org/10.1145/2488608.2488673

[16] M. Puck Rombach, Mason A. Porter, James H. Fowler, and Peter J. Mucha. 2013. Core-Periphery Structure in Networks. arXiv:1202.2684 [cs.SI] https://arxiv.org/abs/1202.2684

[17] SuiteSparse Matrix Collection. 2019. https://sparse.tamu.edu/ Accessed: 2025-04-30.

[18] Daisuke Takafuji, Koji Nakano, and Yasuaki Ito. 2019. Efficient GPU Implementations to Compute the Diameter of a Graph. In *2019 Seventh International Symposium on Computing and Networking (CANDAR)*. 102–111. https://doi.org/10.1109/CANDAR.2019.00020

[19] Yingfei Tu. 2013. Centrality characteristics analysis of urban rail network. In *2013 IEEE International Conference on Intelligent Rail Transportation Proceedings*. 285–290. https://doi.org/10.1109/ICIRT.2013.6696309

[20] Hongru Wang and Feng Li. 2023. The Wide Diameter of Strong Product Graphs of Complete Graphs and any Graphs. In *2023 IEEE 11th International Conference on Information, Communication and Networks (ICICN)*. 255–260. https://doi.org/10.1109/ICICN59530.2023.10392303

[21] Liu Xiaoshan, Wang Qi, and Liu Yuhui. 2009. Minimal Diameter Orientation of Complete Split Graph. In *2009 Asia-Pacific Conference on Information Processing*, Vol. 1. 285–287. https://doi.org/10.1109/APCIP.2009.79

[22] Haoshen Yang, Hao Lu, Naw Safrin Sattar, Hang Liu, and Feiyi Wang. 2024. Establish the basis for Breadth-First Search on Frontier System: XBFS on AMD GPUs. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 650–658. https://doi.org/10.1109/SCW63240.2024.00090

[23] Yanping Zhang, Yuanyuan Bao, Shu Zhao, Jie Chen, and Jie Tang. 2015. Identifying Node Importance by Combining Betweenness Centrality and Katz Centrality. In *2015 International Conference on Cloud Computing and Big Data (CCBD)*. 354–357. https://doi.org/10.1109/CCBD.2015.19