Enumeration of far-apart pairs by decreasing distance for faster hyperbolicity computation

David Coudert, André Nusser, Laurent Viennot

Enumeration of far-apart pairs by decreasing distance for faster hyperbolicity computation*

David Coudert[1], André Nusser[2], and Laurent Viennot[3]

[1]Université Côte d'Azur, Inria, CNRS, I3S, France
[2]Max Planck Institute for Informatics and Graduate School of Computer Science, Saarland Informatics Campus, Saarbrücken, Germany
[3]Université Paris Cité, Inria, CNRS, Irif, France

Abstract

Hyperbolicity is a graph parameter which indicates how much the shortest-path distance metric of a graph deviates from a tree metric. It is used in various fields such as networking, security, and bioinformatics for the classification of complex networks, the design of routing schemes, and the analysis of graph algorithms. Despite recent progress, computing the hyperbolicity of a graph remains challenging. Indeed, the best known algorithm has time complexity $O(n^{3.69})$, which is prohibitive for large graphs, and the most efficient algorithms in practice have space complexity $O(n^2)$. Thus, time as well as space are bottlenecks for computing the hyperbolicity.

In this paper, we design a tool for enumerating all far-apart pairs of a graph by decreasing distances. A node pair (u, v) of a graph is far-apart if both v is a leaf of all shortest-path trees rooted at u and u is a leaf of all shortest-path trees rooted at v. This notion was previously used to drastically reduce the computation time for hyperbolicity in practice. However, it required the computation of the distance matrix to sort all pairs of nodes by decreasing distance, which requires an infeasible amount of memory already for medium-sized graphs. We present a new data structure that avoids this memory bottleneck in practice and for the first time enables computing the hyperbolicity of several large graphs that were far out of reach using previous algorithms. For some instances, we reduce the memory consumption by at least two orders of magnitude. Furthermore, we show that for many graphs, only a very small fraction of far-apart pairs has to be considered for the hyperbolicity computation, explaining this drastic reduction of memory.

As iterating over far-apart pairs in decreasing order without storing them explicitly is a very general tool, we believe that our approach might also be relevant to other problems.

Keywords: Gromov hyperbolicity; graph algorithms, far-apart pairs iterator.

*This work has been supported by the French government, through the UCA[JEDI] Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-15-IDEX-01, the ANR project Multimod with the reference number ANR-17-CE22-0016 and the ANR project Distancia with reference number ANR-17-CE40-0015.

1 Introduction

This paper aims at computing the hyperbolicity of graphs whose size ranges from tens of thousands to millions of nodes. The hyperbolicity is a parameter of a metric space generalizing the idea of Riemannian manifolds with negative curvature. When considering the metric of a graph, the hyperbolicity measures, to some extent, how much the metric of the graph deviates from a tree metric. This parameter was first introduced by Gromov in the context of automatic groups [43] in relation with their Cayley graphs.

Hyperbolicity has received great attention in computer science in the last decades as it seems to capture important properties of several large practical graphs such as Internet [57], the Web [51] and databases relations [64]. It is also used to classify complex networks [1, 5, 47] and was proposed as a measure of how much a network is "democratic" [4, 11]. Formal relationships between Gromov hyperbolicity and the existence of a core (a subset of nodes intersecting a constant fraction of all the shortest-paths) have also been investigated [20]. Reciprocally, the existence of a core has been shown to be inherent to any hyperbolic network [20]. Furthermore, small hyperbolicity has tractability implications and measuring hyperbolicity has applications in routing [9, 19, 48], approximating other graph parameters [18, 32] and bioinformatics [16, 39]. See [1, 37] for recent surveys.

Computing the hyperbolicity is often a prerequisite in the above applications. As hyperbolicity can be defined by a simple 4-point condition, it can be naively computed in $\mathcal{O}(n^4)$ time. As far as we know, the best theoretical algorithm [40] has time complexity $\mathcal{O}(n^{3.69})$. Although its complexity is $o(n^4)$, it is still supercubic and the algorithm appears to be impractical for graphs with a few tens of thousands of nodes. On the lower bounds side it was shown that under the Strong Exponential Time Hypothesis [46] hyperbolicity cannot be computed in subquadratic-time, even for sparse graphs [13, 24, 40].

The only practical algorithms that can manage larger graphs [12, 23] enumerate all pairs of nodes by decreasing distance. For each pair, each 4-tuple obtained with a previous pair is tested with regard to the 4-point condition defining hyperbolicity. Each test of a 4-tuple provides a lower bound of hyperbolicity. This approach allows to stop the enumeration as soon as the distance of the scanned pair equals twice the best hyperbolicity lower bound found so far. As it scans a portion of all 4-tuples, its worst case complexity is $\mathcal{O}(n^4)$ but it appears much faster in practice as first scanning pairs with large distances allows to find good lower bounds early in the enumeration. A main optimization for further reducing the number of pairs scanned, consists in considering only far-apart pairs, that is, pairs such that no neighbor of one node is further apart from the other node, see Section 2 for a formal definition. It can be proven that the 4-point condition defining hyperbolicity holds on all 4-tuples if it holds on 4-tuples made up of two such far-apart pairs [52, 59]. The main bottleneck of this method lies in its inherent quadratic space usage: all far-apart pairs and all-pair distances are stored in the current implementations of this approach. Computing the hyperbolicity of practical graphs with millions of nodes thus remains a great challenge.

1.1 Our approach

To make progress towards this challenge, we propose to enumerate far-apart pairs by decreasing distance without computing all-pair distances. The key of our approach is to first compute all eccentricities, that is, for each node, what is the largest distance from it. Computing all eccentricities is feasible in practice. Indeed, efficient algorithms for computing all eccentricities [60, 38, 50] came along in a line of research for improving diameter computation of real world graphs [2, 29, 14, 38].

Note that we obtain the diameter D of the graph as a side product, as it simply is the maximum eccentricity. We then scan nodes with eccentricity D and enumerate far-apart nodes at distance D from them, that is, nodes at distance D that form far-apart pairs with them. We then scan nodes with eccentricity at least $D-1$ and enumerate far-apart nodes at distance $D-1$ from them, and so on. We also include various optimizations proposed in previous studies [23, 12] to further reduce the number of 4-tuples considered. The main difficulty of our approach is that we have to compute distances on the fly as we do not store all-pair distances. This mainly requires to perform a breadth-first search (BFS) for each node of a far-apart pair considered. Interestingly, we show how to prune these BFS searches based on some of the optimizations previously proposed to prune the number of 4-tuples to consider [12]. Storing most recent BFS searches in a cache also increases performance for some instances as we observe some sharing of far-apart nodes in practice.

1.2 Main contributions

Our main contributions are the following.

- We present the first non-naive algorithm for iterating over far-apart pairs that neither computes and stores all distances explicitly, nor sorts the node pairs by recomputing all distances from scratch whenever they are needed.
- This, for the first time, enables enumerating all far-apart pairs of large graphs. Previously this was not possible either due to excessive amounts of time or memory needed for the computation of all far-apart pairs.
- As the prime application of our algorithm for iterating over far-apart pairs, we significantly reduce the memory consumption when computing the graph hyperbolicity. The memory reduction is at least two orders of magnitude for some instances.
- This drastic memory reduction enables us to compute the hyperbolicity of many large graphs for the first time.
- Due to the significance of far-apart pairs in a graph (e.g., they are the defining nodes for radius, eccentricities, diameter, ...), we believe that our contribution of a far-apart pair iterator is also relevant in other settings.

1.3 Organization

Definitions and notations used in this paper are introduced in Section 2. We then present our far-apart pair iterator in Section 3. Section 4 is devoted to hyperbolicity. We recall its definition and review the best know algorithmic results on this parameter. We then present our memory efficient algorithm based on the proposed far-apart pair iterator. In Section 5, we report on the experimental evaluation of the algorithms presented in this paper on various graphs. In particular, we compare our algorithm for computing hyperbolicity with the state-of-the-art algorithm [12]. We conclude this paper in Section 6 with some directions for future research.

2 Definitions and notations

We use standard graph terminology [10, 36]. All graphs considered in this paper are finite, undirected, connected, unweighted and simple. The graph $G = (V, E)$ has $n = |V|$ nodes and $m = |E|$ edges. The open neighborhood $N_G(S)$ of a set $S \subseteq V$ consists of all nodes in $V \setminus S$ with at least one neighbor in S.

Given two nodes u and v, a *uv-path* of length $\ell \geq 0$ is a sequence of nodes $(u = v_0 v_1 \dots v_\ell = v)$, such that $\{v_i, v_{i+1}\}$ is an edge for every i. In particular, a graph G is *connected* if there exists a uv-path for all pairs $u, v \in V$, and in such a case the *distance* $\mathrm{d}_G(u, v)$ is defined as the minimum length of a uv-path in G. When G is clear from the context, we write d (resp. N) instead of d_G (resp. N_G). The *eccentricity* $\mathrm{ecc}(u)$ of a node u is the maximum distance between u and any other node $v \in V$, i.e., $\mathrm{ecc}(u) = \max_{v \in V} \mathrm{d}(u, v)$. The maximum eccentricity is the *diameter* $\mathrm{diam}(G)$ and the minimum eccentricity is the *radius* $\mathrm{rad}(G)$.

The notion of *far-apart* pairs of nodes has been introduced [59, 52] to reduce the number of 4-tuples to consider in the computation of the hyperbolicity (see Section 4). Roughly, we say that two nodes $u, v \in V$ are *far-apart* if for all $w \in V$ neither u lies on a shortest path from w to v, nor v lies on a shortest path from u to w. More formally, we have:

Definition 1. *In a graph $G = (V, E)$, node u is* far *from node v, or v-far, if for any neighbor w of u, we have $\mathrm{d}(v, w) \leq \mathrm{d}(v, u)$. The pair u, v of nodes is* far-apart *if u is v-far and v is u-far.*

The number of far-apart pairs in a graph can be orders of magnitude smaller than the total number of pairs. For instance, a $p \times q$ grid has only 2 far-apart pairs. On the other hand, all pairs in a clique graph are far-apart.

The set of all far-apart pairs can be determined in time $\mathcal{O}(nm)$ in unweighted graphs through breadth-first search (BFS), and the interested reader will find in Appendix A a discussion on several time and space complexity trade-offs for determining far-apart pairs. We now present some interesting properties of far nodes.

Lemma 1. *For $v \in V$, node $u \in V$ is v-far if and only if u is a leaf of all shortest path trees rooted at v.*

Proof. Clearly, a non-leaf node u of a shortest path tree rooted at v has a neighbor at distance $\mathrm{d}(v, u) + 1$ and cannot be v-far. Reciprocally, if u is not v-far, it has a neighbor w satisfying $\mathrm{d}(v, w) = \mathrm{d}(v, u) + 1$. Modifying any shortest path tree by setting u as the parent of w yields a valid shortest path tree where u is not a leaf. □

From Lemma 1, we also get the following immediate results.

Corollary 1. *For each $u \in V$, any $v \in V$ such that $\mathrm{d}(u, v) = \mathrm{ecc}(u)$ is u-far.*

Corollary 2. *For any $u, v \in V$, if $\mathrm{d}(u, v) = \mathrm{diam}(G)$, then the pair (u, v) is far-apart.*

In particular, Corollary 1 implies that for any node u, the set of u-far nodes is non-empty.

Interestingly, knowing the far nodes of a node u and their distance to u, it is possible to scan distant nodes in a sort of backward BFS as described in Appendix B.

3 Iterator over far-apart pairs

In this section, we engineer a data structure and algorithms to determine the set of far-apart pairs and return these pairs sorted by decreasing distances. Our objective is to provide an iterator that determines the next pair to yield on the fly, that postpones computations as much as possible, and which has an acceptable memory consumption.

The high-level idea of our data structure is that we maintain for each distance up to the diameter a dictionary that, for each node, contains one of the following:

- **no entry:** In this case, this node is not involved in any far-apart pair at this distance.
- **a list of vertices:** These are all the far nodes for this node at this distance. We only have to check whether it is also a far-apart pair.
- **an empty list:** This is a marker that we still have to compute the far nodes for this node and then proceed according to the previous case.

We can iterate over the dictionaries in decreasing distance and compute the far-apart nodes lazily. This lazy computation and the possibility to delete entries that are not needed anymore are the reasons why we are able to save memory in practice. In the remainder of this section, we present the details of our data structure.

Data structure. To store and organize data, we use an array F indexed by distances in range from 1 to $\mathrm{diam}(G)$, so that cell F^d contains data related to far-apart pairs at distance d. More precisely, F^d is a hash map associating to a node $u \in V$ the subset F_u^d of u-far nodes at distance d from u. The subset F_u^d can be implemented using either a set data structure allowing to answer a membership query in $\mathcal{O}(1)$ time, or an array of size $|F_u^d|$ whose elements are sorted according to any total ordering of the nodes to enable membership queries in time $\mathcal{O}(\log |F_u^d|)$.

We assume, as it is the case for most modern programming languages, that it is possible to visit the elements of a hash map in an arbitrary order (e.g., insertion order). We have the same assumption for set data structures.

Initialization. Recall that our aim is to iterate over far-apart pairs by non-increasing distances and to postpone computation as much as possible. To this end, we initially store in F, for each node u with eccentricity $\mathrm{ecc}(u)$, an empty set of u-far nodes in F^d with $d = \mathrm{ecc}(u)$. This empty set will serve as an indicator to trigger the effective computation of the set of u-far nodes the first time it is needed (recall that by Corollary 1, this set is non-empty).

Clearly, this initialization procedure requires the knowledge of the eccentricities of all nodes. Fortunately, although the determination of the eccentricities has worst-case time complexity in $O(nm)$, practically efficient algorithms have been proposed to perform this task [38, 60]. These algorithms maintain upper and lower bounds on the eccentricity of each node and improve these bounds by computing distances from a few well chosen nodes until all gaps are closed. In practice, these algorithms are orders of magnitude faster than a naive algorithm performing a BFS from each node.

Filling. Each time we compute distances from a node u that has not been considered before, the corresponding sets F_u^d for all d can be inserted in F. This is done in particular while computing the eccentricities during the initialization of the data structure.

Next. We define a function $\texttt{next}(F)$ that yields the next far-apart pair in the ordering. Observe that, since F is used to iterate over the far-apart pairs by non-increasing distances, when starting to consider far-apart pairs at distance d, we know that all pairs at distance $d' > d$ have already been considered and that at initialization, we have added in F^d an empty set for each node with eccentricity d. Hence, all nodes involved in a far-apart pair at distance d are known.

Therefore, we can define a function $\texttt{next}(F^d)$ that returns either the next item (u, F_u^d) in the arbitrary ordering on the items stored in the hash map F^d, or `Stop` when all items have been considered. Similarly, we define a function $\texttt{next}(F_u^d)$ that returns either the next node in the ordering defined over the nodes in F_u^d, or `Stop` when all nodes have been considered.

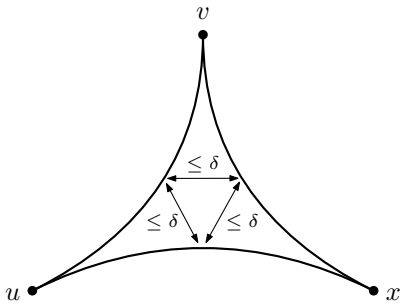
For a distance d and a node u, we repeat calls to $\texttt{next}(F_u^d)$ until finding a node w such that u is w-far (by testing if u is in F_w^d), in which case the far-apart pair (u, w) at distance d is returned. If no node w is found, we go to the next node in F^d through a call to $\texttt{next}(F^d)$. When all nodes in F^d have been considered, we start considering far-apart pairs at distance $d - 1$.

The use of a total ordering on the nodes allows to ensure that a far-apart pair (u, v) is returned only once, i.e., when $u < v$. During these operations, if we encounter an empty set F_x^d, we know from the initialization step that x has never been considered before and that $\mathrm{ecc}(x) = d$. So, we compute distances from x and store the x-far nodes in F via the filling step. That is, we insert in F the corresponding sets $F_x^{d'}$ for each $d' \leq d$.

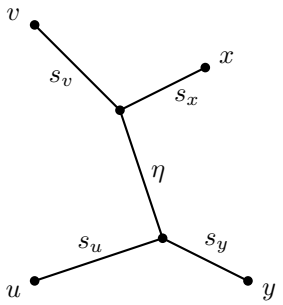
Improvements. Depending on the usage of this data structure, some simple improvements can be done, such as:

1. When iterating over the nodes in F_u^d, each time a node $w \in F_u^d$ is found such that u is not w-far, we can remove w from F_u^d. This avoids checking twice if a pair is far-apart, and, at the end of the iterations on F_u^d, this set will contain a node w only if u is w-far. Observe that this improvement is interesting only if an algorithm has to iterate more than once over the far-apart pairs. We use this improvement in our algorithm for computing hyperbolicity.
2. When all nodes in F^d have been considered, one can delete F^d to reduce the memory consumption if appropriate with the usage. Similarly, one can avoid storing F^d if only far-apart pairs at distance $d' > d$ are requested, as is the case for instance to enumerate the diameters of a graph or when computing hyperbolicity when a lower-bound has been found.

The time complexity for iterating over all far-apart pairs sorted by non-increasing distance is in expected time $\mathcal{O}(nm)$ when using sets for F_u^d (resp., $\mathcal{O}(nm + n^2 \log n)$ when using arrays). Indeed, the algorithm computes BFS distances from each node of the graph (some during initialization, and others during queries). Furthermore, the query time to report all far-apart pairs involving a node u is in $\mathcal{O}(n)$ since $|\cup_{d=1}^{\mathrm{ecc}(u)} F_u^d| = \mathcal{O}(n)$ and checking if $u \in F_w^d$ requires time $\mathcal{O}(1)$ (resp., $\mathcal{O}(\log n)$). The sorting operation over the far-apart pairs is implicit and thus adds no extra cost. The space complexity of the data structure is in $\mathcal{O}(n^2)$. However, we will see in Section 5 that it is much smaller in practice.



(a) A δ-thin triangle.



(b) 4-points condition on a tree.

Figure 1: Hyperbolicity in a metric space (Fig. 1a) and in a tree (Fig. 1b).

4 Use Case: Gromov Hyperbolicity

We now present an interesting use case for our far-apart pair iterator with the computation of the Gromov hyperbolicity of a graph [43].

4.1 Definitions

A metric space (V, d) is δ-hyperbolic if any triangle (u, v, x) in this metric space is δ-thin, that is if any point y along one of the geodesics (i.e., shortest path in a metric space) uv, ux or vx is at distance at most δ from one of the two other geodesics (see Fig. 1a for an illustration). This notion has been extended to graphs [43]. More precisely, a metric space (V, d_G) is a *tree metric* if there exists a distance-preserving mapping from V to the nodes of an edge-weighted tree. If so, the graph G is said to be 0-hyperbolic because it satisfies the 4-points condition below with parameter $\delta = 0$. Introducing some slack δ allows to measure how much the metric of a graph deviates from that of a tree. This slack δ is called the *hyperbolicity* of the graph:

Definition 2 (**4-points Condition, [43]**). *Let G be a connected graph. For every 4-tuple u, v, x, y of nodes of G, we define $\delta(u, v, x, y)$ as half of the difference between the two largest sums among $S_1 = \mathrm{d}(u, v) + \mathrm{d}(x, y)$, $S_2 = \mathrm{d}(u, x) + \mathrm{d}(v, y)$, and $S_3 = \mathrm{d}(u, y) + \mathrm{d}(v, x)$.*

The hyperbolicity of G, denoted by $\delta(G)$, is equal to $\max_{u,v,x,y \in V(G)} \delta(u, v, x, y)$. Moreover, we say that G is δ-hyperbolic *whenever $\delta \geq \delta(G)$.*

Other characterizations exist for 0-hyperbolic graphs and yield other definitions for the hyperbolicity δ of a graph that differ only by a small constant factor [7, 33, 43].

Let us illustrate the behavior of the 4-points condition on trees. Without loss of generality, consider the example of Fig. 1b, where s_u, s_v, s_x, s_y and η denote the lengths of the corresponding paths in the figure. We have $S_1 = \mathrm{d}(u, v) + \mathrm{d}(x, y) = s_u + s_v + s_x + s_y + 2\eta$, $S_2 = \mathrm{d}(u, x) + \mathrm{d}(v, y) = s_u + s_v + s_x + s_y$, and $S_3 = \mathrm{d}(u, y) + \mathrm{d}(v, x) = s_u + s_v + s_x + s_y + 2\eta$. The two largest sums S_1 and S_3 are such that $S_1 = S_3$, and so $\delta(u, v, x, y) = 0$. The same holds for any quadruple in a tree and so trees are 0-hyperbolic.

From the 4-points condition above, it is straightforward to compute graph hyperbolicity in $\Theta(n^4)$-time. In theory, it can be decreased to $\mathcal{O}(n^{3.69})$ by using a clever $(\max, \min)$-matrix product [40]; however, in practice, the best-known algorithms still run in $\mathcal{O}(n^4)$-time [12, 23]. Graphs with small hyperbolicity can be recognized faster. In particular, 0-hyperbolic graphs coincide with

block graphs, that are graphs whose biconnected components are complete subgraphs [6, 45]. Hence, deciding whether a graph is 0-hyperbolic can be done in time $\mathcal{O}(n+m)$. The latter characterization of 0-hyperbolic graphs follows from a more general result stating that the hyperbolicity of a graph is the maximum hyperbolicity of its biconnected components (see e.g. [23] for a proof). More recently, it has been proven that the recognition of $\frac{1}{2}$-hyperbolic graphs is computationally equivalent to deciding whether there exists a chordless cycle of length 4 in a graph [24]. The latter problem can be solved in deterministic $\mathcal{O}(n^{3.26})$-time [49] and in randomized $\mathcal{O}(n^{2.373})$-time [63] by using fast matrix multiplication.

Several pre-processing methods for reducing the size of the input graph have been proposed. In particular, it has been proved [59] that the hyperbolicity of G is equal to the maximum of the hyperbolicity of the graphs resulting from both a *modular* [41, 44] or a *split* [31, 30] decomposition of G. These decompositions can be computed in linear time [17] and algorithms with time complexity in $\mathcal{O}(\mathrm{mw}(G)^3 \cdot n+m)$ and $\mathcal{O}(\mathrm{sw}(G)^3 \cdot n+m)$ when parameterized respectively by the *modular width* $\mathrm{mw}(G)$ and the *split width* $\mathrm{sw}(G)$ have been proposed [26]. Moreover, one can use modified versions of the atoms of a decomposition of G by clique-minimal separators [61, 8, 25] as a pre-processing for hyperbolicity [22]. This decomposition can be obtained in time $\mathcal{O}(nm)$. Although we do not use these pre-processing methods in this work, they are complementary to our approach and it is thus possible to combine them with our approach.

4.2 Previous algorithm

In this section, we recall the state of the art algorithm for computing hyperbolicity [12]. This algorithm, referenced as Borassi et al. algorithm in the sequel, improves upon a former algorithm [23] by adding pruning techniques to further reduce the number of 4-tuples to consider. To the best of our knowledge, these algorithms are the only one enabling to compute the exact hyperbolicity of graphs with up to 50 000 nodes. These algorithms are based on the following lemmas.

Lemma 2 ([59, 23])**.** *Let G be a connected graph. There exist two far-apart pairs (u,v) and (x,y) satisfying $\delta(u,v,x,y) = \delta(G)$.*

Lemma 3 ([23])**.** *For every 4-tuple u,v,x,y of nodes of a connected graph G, we have $\delta(u,v,x,y) \leq \min_{a,b\in\{u,v,x,y\}} \mathrm{d}(a,b)$. Furthermore, if $S_1 = \mathrm{d}(u,v) + \mathrm{d}(x,y)$ is the largest of the sums defined in Definition 2 (which can be assumed w.l.o.g.), we have $\delta(u,v,x,y) \leq \frac{1}{2}\min\{\mathrm{d}(u,v), \mathrm{d}(x,y)\}$.*

For the sake of completeness, we quickly give the proof of Lemma 3. Without loss of generality the second largest sum is $S_2 = d(u,x) + d(v,y)$. By triangle inequality, we have $\mathrm{d}(u,v) \leq \mathrm{d}(u,x) + \mathrm{d}(x,y) + \mathrm{d}(y,v) = S_2 + \mathrm{d}(x,y)$, yielding $S_1 - S_2 \leq 2\,\mathrm{d}(x,y)$. We can similarly obtain $S_1 - S_2 \leq 2\,\mathrm{d}(u,v)$.

The key idea of the state of the art algorithms [12, 23] is to visit the most promising 4-tuples first, that is, those made of pairs of far-apart nodes at largest distance, and to stop computation as soon as the bounds of Lemma 3 are reached. These algorithms thus need to iterate over far-apart pairs ordered by decreasing distances.

More precisely, Algorithm 1 below (see also [12]) iterates first over the far-apart pairs sorted by non-increasing distances. Then, given the i-th far-apart pair (x_i, y_i), it iterates over the previous far-apart pairs (v,w), such that $\mathrm{d}(v,w) \geq \mathrm{d}(x_i,y_i)$ in order to consider quadruples (v,w,x_i,y_i) such that $S_1 = d(v,w) + d(x_i,y_i)$ is the largest sum. Note that such pairs (v,w) have been considered previously and satisfy $(v,w) = (x_j,y_j)$ for some $j < i$. This ensures by Lemma 3 that as soon

Algorithm 1: Borassi et al. algorithm for computing the hyperbolicity [12]

Input: $\mathcal{F} = (\{x_1, y_1\}, \ldots, \{x_N, y_N\})$: an ordered list of far-apart pairs; d: the distance matrix.

```
1  δ_L ← 0
2  mates[v] ← ∅ for each v
3  for i ∈ [1, N] do
4      if d(x_i, y_i) ≤ 2δ_L then
5          return δ_L
6      (acceptable, valuable) ← computeAccVal(x_i, y_i, δ_L, d)
7      for v ∈ valuable do
8          for w ∈ mates[v] do
9              if w ∈ acceptable then
10                 δ_L ← max{δ_L, δ(x_i, y_i, v, w)}
11     add y_i to mates[x_i]
12     add x_i to mates[y_i]
13 return δ_L
```

as $\mathrm{d}(x_i, y_i) \leq 2\delta_L$, where δ_L is the current best solution, we determined the hyperbolicity of the graph. So, the hyperbolicity δ_L of the graph is then returned in Line 5. This ensures by Lemma 3 that as soon as $\mathrm{d}(x_i, y_i) \leq 2\delta_L$, where δ_L is the current best solution, we found the hyperbolicity of the graph. So, the hyperbolicity δ_L of the graph is then returned in Line 5. To iterate over the pairs (v, w) such that $\mathrm{d}(v, w) \geq \mathrm{d}(x_i, y_i)$, the algorithm maintains the *mates* of each node. Node w is a mate of v if (v, w) is a far-apart pair satisfying $\mathrm{d}(v, w) \geq \mathrm{d}(x_i, y_i)$. In other words, (v, w) is a far-apart pair previously considered for some $j < i$ such that $(x_j, y_j) = (v, w)$ or $(x_j, y_j) = (w, v)$.

To further prune the search space, we use the notions of *skippable*, *acceptable* and *valuable* nodes of the Borassi et al. approach that are computed by `computeAccVal` according to the definitions given below [12]. Algorithm 1 can be read before considering the details of this optimization. Its time complexity is in $\mathcal{O}(n^4)$ and its space complexity is in $\Theta(n^2)$. Indeed, it not only needs to store the list of far-apart pairs, but also the distance matrix, and the lists of mates.

Skippable, acceptable and valuable. Given a pair x, y of nodes and a lower bound δ_L on hyperbolicity, Borassi et al. [12] propose a classification of the nodes to prune the nodes that cannot lead to any improvement of the lower-bound δ_L known so far. For instance, a node v such that $\min\{\mathrm{d}(x, v), \mathrm{d}(y, v)\} \leq \delta_L$ can be skipped, since by Lemma 3 we then have $\delta(x, y, v, w) \leq \delta_L$ for any w. In Lemma 4 we summarize the conditions defining (x, y, δ_L)-*skippable* nodes.

Lemma 4 ([12])**.** *A node v is (x, y, δ_L)-skippable if and only if it satisfies any of the following conditions:*

1. *v does not belong to any far-apart pair considered before (x, y) (Lemma 5 in [12]);*
2. *$\min\{\mathrm{d}(x, v), \mathrm{d}(y, v)\} \leq \delta_L$ (Lemma 3);*
3. *$2\,\mathrm{ecc}(v) - \mathrm{d}(x, v) - \mathrm{d}(y, v) < 4\delta_L + 2 - \mathrm{d}(x, y)$ (Lemma 8 in [12]);*

4. $\mathrm{ecc}(v) + \mathrm{d}(x,y) - 3\delta_L - \frac{3}{2} < \max\{\mathrm{d}(x,v), \mathrm{d}(y,v)\}$ *(Lemma 9 in [12]).*

A node that does not satisfy any condition of Lemma 4 is defined as (x, y, δ_L)-*acceptable*, and so it must be considered. This class is further refined by Borassi et al. [12] with the subset of *c-valuable* nodes, where c is any fixed node (a good choice is a node with small eccentricity or centrality) as specified in the following Lemma.

Lemma 5 ([12])**.** *Let c be any fixed node. A (x, y, δ_L)-acceptable node v is c-valuable if* $2\,\mathrm{d}(c,v) - 2\delta_L > \mathrm{d}(x,v) + \mathrm{d}(y,v) - \mathrm{d}(x,y)$.

In Algorithm 1, the 4-tuples considered with far-apart pair (x_i, y_i) are such that v is c-valuable, w is (x, y, δ_L)-acceptable and (v, w) is a far-apart pair seen previously. Overall, the classification of the nodes is done in overall time $\mathcal{O}(n)$ for a given pair (x_i, y_i) and lower bound δ_L. Previous experiments [12] show that this classification leads to a significant reduction of the number of considered 4-tuples as well as computation time. Intuitively, the running time improvements using the classification into acceptable and valuable nodes stem from the following insights. If a node v is not valuable, then we avoid checking all mates of v. If a node v is valuable but one of its mates w is not acceptable, then we avoid the memory accesses to distances to compute $\delta(x, y, v, w)$.

4.3 Hub labeling

A main bottleneck of Algorithm 1 comes from the $\Theta(n^2)$ memory usage. This can be alleviated by using hub labeling [34], a technique that allows to encode distances in a graph. The technique is also called two-hop labeling [21]. It appears to give a very efficient space-time tradeoff in practice. We tried to use it as a replacement of the distance matrix in Algorithm 1. It perfectly fits in memory with all practical graphs we could test. However, we could not find a way to identify valuable nodes v for a pair x, y without computing the distances $d(x, v)$ and $d(y, v)$ for all v. It is then more efficient to perform BFSs from x and y. As the technique appeared as an inefficient way to extend Algorithm 1, we abandoned it.

4.4 Our algorithm

To improve upon Algorithm 1, and in particular to reduce the memory usage in practice, we do the following.

- We use the far-apart pair iterator presented in Section 3 to avoid the pre-computation and storage of the list of far-apart pairs. The use of Improvement 1 of Section 3 ensures that at the end of the visit of F_u^d for some u and d, it contains only the nodes w such that u is w-far, and so the pair (v, w) is far-apart.

- Let x, y be the currently considered far-apart pair. We design a function `mates`(F, v, d_{xy}) to iterate over the far-apart pairs at distance $d \geq d_{xy}$ that involve v and that have previously been reported by `next`(F). In the remainder we always use the function `mates` with the argument d_{xy} set to $d(x, y)$.

  When $\mathrm{d}(x,y) < d \leq \mathrm{diam}(G)$, this function simply yields nodes from F_v^d since the order of operations of the algorithm when using Improvement 1 ensures that F_v^d contains only the nodes forming far-apart pairs with v.

Algorithm 2: New algorithm for computing the hyperbolicity

Input: $G = (V, E)$

Initialize the far-apart pair iterator F
$\delta_L \leftarrow$ `lowerBoundHeuristic`(G)
while `has_next`(F) **do**
  $(x, y) \leftarrow$ `next`(F) `// provides` d(x, y)
  if $\mathrm{d}(x, y) \leq 2\delta_L$ **then**
    return δ_L
  $\mathrm{d}_x \leftarrow c_{x,y,\delta_L}$-`prunedBFS`$(x)$
  $\mathrm{d}_y \leftarrow c_{y,x,\delta_L}$-`prunedBFS`$(y)$
  (`acceptable`, `valuable`) $\leftarrow$ `computeAccVal`(x, y, δ_L, G)
  for $v \in$ `valuable` **do**
    for $w \in$ `mates`$(F, v, d(x, y))$ **do** `// provides` d(v, w)
      if $w \in$ `acceptable` **then**
        $\delta_L \leftarrow \max\{\delta_L, \delta(x, y, v, w)\}$
return δ_L

When $d = \mathrm{d}(x, y)$, we have to ensure that a v-far node w is yielded if and only if the pair (v, w) has previously been reported by a call to `next`(F) (so the pair (v, w) is far-apart). To do so, we modify the far-apart pairs iterator and its `next`(F) function as follows. We use an extra hash map T (initially empty) associating to a node u the subset of u-far nodes at distance d from u that have previously been reported by `next`(F). Then, when `next`(F) yields a far-apart pair (x, y), we store y in T_x and x in T_y. This way, when $d = \mathrm{d}(x, y)$, function `mates`(F, v, d) simply has to yield nodes from T_v. Finally, as soon as function `next`(F) starts reporting pairs at distance $d - 1$, we exchange hash maps T and F^d (alternative implementation of Improvement 1), and proceed with a cleared hash map T.

- Instead of giving the distance matrix as input to the algorithm, we compute for each pair (x, y) the BFS distances from x and y before the call to `computeAccVal()`. Since the distance $\mathrm{d}(v, w)$ is obtained while extracting the mates of v from the far-apart pair iterator, we get all the needed distances to compute $\delta(x, y, v, w)$.

Although these repeated computations of BFS distances have no impact on the overall time complexity of the algorithm, which remains in $\mathcal{O}(n^4)$, they represent a significant computation time in practice. More precisely, the algorithm inspects all far-apart pairs (x, y) such that $d(x, y) \geq 2\delta(G)$. Let $\Phi \leq n^2$ denote their number. The running time of the algorithm is then $\mathcal{O}(nm+\Phi m+\Phi^2)$ as the initialization performs at most one BFS per node, the main loop runs two BFSs per inspected far-apart pair (x, y), and the number of (v, w) pairs inspected for (x, y) is at most the number of pairs considered so far, which is at most Φ. To reduce the impact on the overall computation time, we propose Optimizations 2 (Section 4.4.2) and 3 (Section 4.4.3) below.

See Algorithm 2 for the overall presentation of our algorithm. In the following, we present some optimizations aiming at reducing the computation time.

4.4.1 Optimization 1: Lower bound initialisation

The technique of acceptable and valuable nodes becomes more efficient when δ_L is larger as Inequalities 3 and 4 of Lemma 4 and the inequality of Lemma 5 become stricter. For that reason, we first use the heuristic described in [23] to set an initial value to δ_L. It is referenced as lowerBoundHeuristic in Algorithm 2.

4.4.2 Optimization 2: Cache of BFSs

We use a cache of BFSs to avoid recomputing a BFS that has recently been computed. This cache has a bounded capacity (e.g., 1000 BFSs) and stores the vectors of distances obtained through the most recent BFSs performed. This cache is used in two places: when computing distances from x and y for testing quadruples involving a pair (x, y), and when computing far-apart pairs within the function `has_next`() of the far-apart pair iterator. Observe that even a cache of 2 BFSs is beneficial as function `next`(F) reports successively all far-apart pairs at distance d involving a node x and such that $x < y$. The BFS from x is thus performed only once as long as we scan (x, y) pairs with same x node.

4.4.3 Optimization 3: Pruned BFS for searching acceptable nodes

When considering a pair (x, y), we perform BFS searches from both x and y to obtain distances from x and y and detecting both acceptable and valuable nodes. Lemma 4 allows to restrict both searches as follows.

First, we observe that if a node v satisfies $\mathrm{ecc}(v) - \mathrm{d}(x, v) < 3\delta_L - \frac{3}{2} - \mathrm{d}(x, y)$, then Lemma 4 applies and v is (x, y, δ_L)-skippable. A (x, y, δ_L)-acceptable node v must thus satisfy:

$$\mathrm{ecc}(v) - \mathrm{d}(x, v) \geq c_{x,y,\delta_L}, \quad \text{where} \quad c_{x,y,\delta_L} = 3\delta_L - \frac{3}{2} - \mathrm{d}(x, y). \tag{1}$$

We then define the *c_{x,y,δ_L}-pruned BFS search* from x as a BFS search from x that visits only nodes satisfying Equation (1). More precisely, when visiting a node u, we enqueue only neighbors v of u that satisfy Equation (1). Note that c_{x,y,δ_L} is constant given x, y and δ_L. We can then safely replace the regular BFS from x by a pruned BFS as stated by the following lemma.

Lemma 6. *A c_{x,y,δ_L}-pruned BFS search from x visits all (x, y, δ_L)-acceptable nodes.*

Proof. The proof follows from two facts. First, any (x, y, δ_L)-acceptable node must satisfy Equation (1). Second, any node $v \neq x$ satisfying Equation (1) must have some neighbor closer to x that satisfies Equation (1). This second condition proven below allows to easily prove by induction that all nodes satisfying Equation (1) are visited by a c_{x,y,δ_L}-pruned BFS search from x.

To prove the above second condition, consider a neighbor u of v at distance $\mathrm{d}(x, v) - 1$ from x. By triangle inequality, we have $\mathrm{ecc}(u) \geq \mathrm{ecc}(v) - 1$, implying $\mathrm{ecc}(u) - \mathrm{d}(x, u) \geq \mathrm{ecc}(v) - \mathrm{d}(x, v)$. As v satisfies Equation (1), so does u. □

Notice that the set V_{x,y,δ_L} of nodes visited by a c_{x,y,δ_L}-pruned BFS search from x is larger than the set of (x, y, δ_L)-acceptable nodes. Indeed, Conditions 1 to 3 of Lemma 4 cannot be used for the search. Furthermore, remark that the set V_{x,y,δ_L} depends on the distance $\mathrm{d}(x, y)$ and not on the precise node y. That is,

Lemma 7. *For any z such that $\mathrm{d}(x,y) = \mathrm{d}(x,z)$, we have $V_{x,z,\delta_L} = V_{x,y,\delta_L}$.*

The following results are direct consequences of Equation (1) and Lemma 7.

Corollary 3. *For any z such that $\mathrm{d}(x,y) \geq \mathrm{d}(x,z)$, we have $V_{x,z,\delta_L} \subseteq V_{x,y,\delta_L}$.*

Corollary 4. *For any $\delta > \delta_L$, we have $V_{x,y,\delta} \subseteq V_{x,y,\delta_L}$.*

Lemma 7 and corollaries 3 and 4 enable the use of our pruned BFS in combination with a cache of BFSs. Indeed, during the execution of the algorithm both the considered distance $\mathrm{d}(x,y)$ decreases and the lower bound δ_L increases. Hence, a cached c_{x,y,δ_L}-pruned BFS remains valid for future use. Hence, for Line 7 of Algorithm 2, we first check if a BFS from x is in the cache, and if so we retrieve it. Otherwise, we perform a c_{x,y,δ_L}-pruned BFS search from x and add it to the cache. We proceed similarly for y.

Observe also that to determine the sets of (x,y,δ_L)-acceptable and c-valuable nodes, it suffices to consider nodes that have been visited by the c_{x,y,δ_L}-pruned BFS search, or by the c_{y,x,δ_L}-pruned BFS search if this set is smaller.

4.5 Parallelization

In this section, we discuss possibilities to parallelize our algorithm. The main issue that arises when parallelizing our algorithm is that we rely on knowing the mates, which in turn results from processing far-apart pairs. In a sequential implementation this is not an issue as a quadruple will be considered from one of the two pairs; however, if we process far-apart pairs in parallel, we might miss quadruples.

To overcome this issue, we can first build the set F^d of far-apart pairs at distance d, thus pre-computing all required mates. This can be done either sequentially, by iterating over the far-apart pairs at distance d, or in parallel, by first finding for each node u with eccentricity d the set of u-far nodes in parallel (we already know the u-far nodes of the nodes with larger eccentricities), and then consolidating F^d to keep only far-apart pairs. Observe that each thread used to determine a set of u-far nodes requires to compute a BFS and thus uses space $\mathcal{O}(n)$.

Once the set F^d has been consolidated, one can consider the far-apart pairs in F^d in parallel, examining their associated quadruples. The only point of synchronization between threads is then the correct propagation of improvements of the lower bound δ_L. Note however, that this only affects pruning and thus it is not relevant for the correctness of the algorithm but only for the performance. As we later see in the experiments (Figure 4) most of the time of the algorithm is spent visiting quadruples (and thus, to update δ_L). Hence, this two steps approach for a parallel version of our algorithm seems promising.

5 Experimental results

In this section we conduct experiments to test the performance of our new algorithm in comparison to the previous state-of-the-art one. We additionally test the impact that each proposed optimization has on the overall performance. To gain further insights into our main tool, the algorithm for efficiently enumerating far-apart pairs, we also conduct experiments to analyze the performance and the number of far-apart pairs that graphs exhibit.

5.1 Implementation notes

We have implemented all the algorithms in C++. Our code is publicly available (see [28]). In this section, we discuss some implementation choices.

We have implemented a cache of BFSs with bounded capacity κ that additionally maintains the age information of the data it stores. We use a counter τ, initialized to 0, that is increased by one each time a BFS that is already in the cache is accessed, or a BFS that is not in the cache is added. We associate to a cached BFS from a node x an age information a_x, initialized to the value of τ at insertion time. Then, we set a_x to the current value of τ each time the BFS from x is accessed. Hence, the last accessed BFS is such that $a_x = \tau$. The use of a hash map associating to a node the corresponding BFS and age information enables to decide in time $\mathcal{O}(1)$ if a BFS from x is in the cache, and if so to return a pointer on the corresponding data. Updating the age information is done in time $\mathcal{O}(1)$. The insertion of a BFS in the cache takes expected time $\mathcal{O}(1)$ if the cache is not full, and time $\mathcal{O}(\kappa)$ as soon as it has reached its maximum capacity. Indeed, the insertion of a BFS when the cache is full requires to remove first the BFS with largest age information, that is, the one of the node x maximizing $\tau - a_x$, and so with smallest a_x. Note that for κ much smaller than n, the time required for managing the cache is negligible with respect to the time required for a BFS. Observe that this cache will be accessed $\mathcal{O}(n^2)$ times by Algorithm 2, and more precisely at most twice the number of far-apart pairs at distance $2\delta(G)$ or more.

5.2 Data & Hardware

To evaluate the performances of our algorithms, we use graphs from the BioGRID interaction database (BG-*) [53]; a protein interactions network (dip20170205) [55]; and graphs of the autonomous systems from the Internet (CAIDA_as_* and DIMES_*) [62, 56]. We also test computer networks (Gnutella, Skitter), web graphs (notreDame), social networks (Brightkite, Epinions, Facebook, Slashdot), co-author graphs (ca-*, dblp), road networks (oregon2, FLA-t), a 3D triangular mesh (buddha), a graph from a computer game (FrozenSea), and grid-like graphs from VLSI applications (alue7065). The data is available from `snap.stanford.edu`, `webgraph.di.unimi.it`, `www.dis.uniroma1.it/challenge9`, `graphics.stanford.edu`, `steinlib.zib.de`, and `movingai.com`. Furthermore, we use synthetic inputs: grid300-10 and grid500-10 are square grids with respective sizes 301×301, and 501×501 where 10% of the edges were randomly deleted. Each graph is taken as an undirected unweighted graph and we consider only its largest (with respect to number of vertices) bi-connected component (available from [28]). See Table 1 for the characteristics of these graphs. This set of graphs has been selected as it is representative of the variety of networks (complex networks, road networks, etc.) that are considered in different fields. Most of these graphs have already been used to evaluate the performances of previously proposed exact and heuristic algorithms for hyperbolicity [12, 22, 23, 47, 59].

We used a computer equipped with two Intel Xeon Gold 6240 CPUs operating at 2.6GHz and 192G RAM to run our experiments. Note that our code uses a single thread.

5.3 Parameter choice

As there are instances which do not terminate in reasonable time or need unreasonable amounts of memory, we cap both resources at a fixed value to obtain a clear picture. More precisely, for each graph and each experiment, we kill the process as soon as it takes more than 6 hours or uses more than 192 GB of memory. We use the  symbol to indicate a killed process and, if applicable, put

Table 1: Some graph parameters of all the graphs that we use in our experiments. Note that for all graphs, we extracted the largest biconnected component and restrict to this subgraph in our experiments.

Graph	**#nodes**	**#edges**	**radius**	**diameter**	**mean ecc.**
BG-MV-Physical	9 851	45 558	11	22	14.45
BG-S-Affinity_Capture-MS	17 793	174 210	6	12	9.02
BG-S-Affinity_Capture-RNA	3 339	10 408	3	5	4.00
BG-S-Affinity_Capture-Western	9 971	44 331	10	20	12.60
BG-S-Biochemical_Activity	29 44	10 444	7	13	9.20
BG-S-Dosage_Rescue	1 521	4 143	9	17	12.21
BG-S-Synthetic_Growth_Defect	3 013	21 341	4	8	5.72
BG-S-Synthetic_Lethality	2 258	12 187	4	7	5.45
dip20170205	13 969	60 621	10	17	12.95
CAIDA_as_20000102	4 009	10 101	4	8	5.62
CAIDA_as_20040105	10 424	27 061	4	8	5.98
CAIDA_as_20050905	12 957	33 541	5	8	6.11
CAIDA_as_20110116	23 214	89 783	4	8	6.04
CAIDA_as_20120101	25 614	109 180	4	8	6.10
CAIDA_as_20130101	27 454	124 672	5	10	7.21
CAIDA_as_20131101	29 432	143 000	5	9	6.46
DIMES_201012	18 764	84 851	4	7	5.16
DIMES_201204	16 907	66 489	4	7	5.07
p2p-Gnutella09	5 606	23 510	5	8	6.31
gnutella31-d	33 812	119 127	6	9	7.41
notreDame-d	134 958	833 732	18	36	20.99
ca-CondMat	17 234	84 595	6	12	8.44
ca-HepPh	9 025	114 046	6	11	7.83
ca-HepTh	5 898	20 983	7	11	8.63
com-dblp.ungraph	211 409	883 570	8	15	10.92
dblp-2010	140 610	572 873	10	17	11.99
email-Enron	20 416	163 257	5	9	6.54
epinions1-d	36 111	365 253	5	9	6.36
facebook_combined	3 698	85 963	4	6	5.26
loc-brightkite	33 187	188 577	6	11	7.95
loc-gowalla_edges	137 519	887 929	6	11	7.91
slashdot0902-d	51 528	473 218	5	8	6.04
oregon2_010331	7 602	27 870	4	8	5.89
t.FLA-w	691 175	941 893	890	1780	1378.52
buddha-w	543 652	1 631 574	244	487	360.44
froz-w	749 520	2 895 228	812	1451	1130.38
grid300-10	90 211	162 152	300	600	450.50
grid500-10	250 041	449 831	500	1000	750.49
z-alue7065	34 040	54 835	213	426	319.43

this symbol in the column (i.e., time or memory) that caused the process to be killed. Furthermore, for the cache described in Section 5.1, we use a size of 1000, unless mentioned otherwise.

5.4 Comparison to previous work

To evaluate the performance improvement over previous approaches, we compare to the practical state-of-the-art algorithm which is the Borassi et al. algorithm [12]. To this end, we measure the single-threaded computation time, the memory, and the best upper and lower bound found of our new algorithm as well as Borassi et al. algorithm. The results of this experiment are shown in Table 2.

There are several interesting observations that one can derive from these experiments. First, the new approach that we present in this paper needs significantly less memory. More precisely, the memory consumption is up to a factor of 28 times lower than for Borassi et al. algorithm (see the slashdot0902-d graph), only considering the graphs where both approaches stay within the limits. If we also consider the graphs where Borassi et al. algorithm runs out of memory, we use at least a factor 177 less memory (see grid300-10). Mainly due to this significant reduction of memory consumption, we are able to compute the hyperbolicity for graphs which were previously out of reach due to excessive amounts of memory which would have been necessary. However, there are also graphs for which we can compute the hyperbolicity, which hit the timeout limit of 6 hours when using Borassi et al. algorithm. Note that in these two cases, the computation roughly takes between 4 and 5 hours, so Borassi et al. algorithm takes at least 1 to 2 hours more time to finish on these instances. In general, we note that all graphs for which Borassi et al. algorithm computes the hyperbolicity within the time and memory limits, also stay within the limits using our approach. Over all 40 benchmark instances, our new approach computes the hyperbolicity for 8 more graphs than Borassi et al. algorithm. If the hyperbolicity cannot be computed within the limits with our new approach, then this is due to running out of time instead of running out of memory. This again shows that we achieve a drastic reduction in memory consumption for computing the hyperbolicity.

While we are sometimes faster than Borassi et al. algorithm, one could also assume that sometimes it is the other way around. This is indeed the case, and while for almost all graphs we are slightly faster or slightly slower, there is one instance (namely, DIMES_201012) where we are a factor of 13 slower than Borassi et al. algorithm. We suspect that for this graph, there are many nodes from which we repeatedly perform BFSs to obtain distances. Indeed, as both algorithms follow a similar enumeration of pairs and quadruples, one could expect a slowdown with our algorithm since it needs to recompute some distances. However, the main bottleneck comes from the inspection of quadruples with regards to the 4-points condition (see Section 5.5): for each pair (x, y) inspected, we perform at most two BFSs with cost $O(n + m)$, while the number of quadruples (x, y, v, w) inspected can be proportional to n^2. The fact that both algorithms have comparable running times is thus explained by a marginal cost of distance computations, even with our approach that requires to recompute some of them several times. Note also that both algorithms use different orderings for visiting pairs and quadruples. This implies that one can improve its lower bound on hyperbolicity earlier than the other, greatly reducing the number of quadruples to be subsequently inspected. This is probably the main reason for the fluctuations of running time between the two algorithms.

We specifically want to highlight two grid graphs, grid300-10 and grid500-10, for which our approach only takes 10 seconds and 90 seconds, respectively, while Borassi et al. algorithm fails to compute the hyperbolicity due to exceeding the 192 GB memory limit. Note that for the former grid

Table 2: Comparing the memory and time consumption for our algorithm and Borassi et al. algorithm. If there is a timeout, we state the best lower and upper bounds on the hyperbolicity that we obtained. The last column states the cache usage by our algorithm. For an analysis of the tradeoff between improved running time and increased memory consumption of the cache, see Sections 5.6 and 5.7. We highlight the values in the table where one algorithm evidently dominates the other algorithm with respect to running time or memory usage, respectively.

Graph	**Borassi et al. [12]**			**Our Algorithm**			
	time (s)	**memory**	**hyperb.**	**time (s)**	**memory**	**hyperb.**	**cache usage**
BG-MV-Physical	10 429.0	1.03 GB	4.5	**6 725.0**	**166.33 MB**	4.5	115.83 MB
BG-S-Affinity_Capture-MS	☠	–	[2.0, 4.0]	☠	–	[2.0, 4.0]	–
BG-S-Affinity_Capture-RNA	0.8	157.18 MB	2.0	**0.7**	**57.29 MB**	2.0	39.22 MB
BG-S-Affinity_Capture-Western	7 827.0	1.05 GB	4.0	**5 255.0**	**154.66 MB**	4.0	116.82 MB
BG-S-Biochemical_Activity	**8.4**	104.90 MB	3.0	16.3	**46.16 MB**	3.0	34.68 MB
BG-S-Dosage_Rescue	**12.0**	30.54 MB	4.0	14.3	**24.43 MB**	4.0	17.82 MB
BG-S-Synthetic_Growth_Defect	**1.7**	108.76 MB	2.0	2.9	**43.82 MB**	2.0	35.22 MB
BG-S-Synthetic_Lethality	**1.1**	77.09 MB	2.0	1.8	**33.40 MB**	2.0	26.60 MB
dip20170205	☠	–	[4.5, 5.0]	**18 318.0**	339.30 MB	4.5	163.96 MB
CAIDA_as_20000102	1.3	260.87 MB	2.5	**1.2**	**56.40 MB**	2.5	47.04 MB
CAIDA_as_20040105	**18.4**	1.90 GB	2.5	25.9	**166.40 MB**	2.5	122.06 MB
CAIDA_as_20050905	**16.9**	2.37 GB	3.0	21.8	**203.72 MB**	3.0	151.99 MB
CAIDA_as_20110116	13 806.0	8.41 GB	2.0	**13 491.0**	**556.72 MB**	2.0	271.38 MB
CAIDA_as_20120101	☠	–	[2.0, 2.5]	☠	–	[2.0, 2.5]	–
CAIDA_as_20130101	**2 961.9**	10.09 GB	2.5	3 535.8	**1.47 GB**	2.5	321.09 MB
CAIDA_as_20131101	☠	–	[2.0, 2.5]	**16 108.0**	593.03 MB	2.5	344.76 MB
DIMES_201012	**465.8**	4.86 GB	2.0	6 043.0	**482.41 MB**	2.0	219.88 MB
DIMES_201204	66.3	4.34 GB	2.0	**56.2**	**304.48 MB**	2.0	197.75 MB
p2p-Gnutella09	2.9	381.64 MB	3.0	**2.4**	**98.38 MB**	3.0	65.45 MB
gnutella31-d	139.5	13.13 GB	3.5	**109.5**	**1.51 GB**	3.5	395.77 MB
notreDame-d	–	☠	–	4 514.0	**53.02 GB**	8.0	1.58 GB
ca-CondMat	**112.8**	3.38 GB	3.5	172.9	**281.18 MB**	3.5	201.81 MB
ca-HepPh	**37.0**	1.17 GB	3.0	86.3	**152.86 MB**	3.0	105.79 MB
ca-HepTh	4.0	407.78 MB	4.0	**3.0**	**91.54 MB**	4.0	69.00 MB
com-dblp.ungraph	–	☠	–	5 449.0	**14.20 GB**	5.0	2.47 GB
dblp-2010	–	☠	–	6 904.0	**7.51 GB**	5.5	1.64 GB
email-Enron	**754.1**	5.36 GB	2.5	897.7	**404.38 MB**	2.5	239.05 MB
epinions1-d	**696.7**	18.60 GB	2.5	2 331.3	**759.20 MB**	2.5	422.24 MB
facebook_combined	6 919.0	248.00 MB	1.5	**4 551.0**	**136.65 MB**	1.5	44.99 MB
loc-brightkite	910.4	12.81 GB	3.0	**910.1**	**812.98 MB**	3.0	388.69 MB
loc-gowalla_edges	–	☠	–	14 478.0	**4.04 GB**	3.5	1.61 GB
slashdot0902-d	9 560.0	37.54 GB	2.5	**4 718.0**	**1.34 GB**	2.5	602.96 MB
oregon2_010331	73.0	980.82 MB	2.0	**65.1**	**133.76 MB**	2.0	88.80 MB
t.FLA-w	–	☠	–	☠	–	[81.0, 835.5]	–
buddha-w	–	☠	–	☠	–	[93.0, 221.0]	–
froz-w	–	☠	–	☠	–	[367.5, 633.5]	–
grid300-10	–	☠	–	10.1	**1.08 GB**	280.0	1.05 GB
xgrid500-10	–	☠	–	99.6	**2.99 GB**	463.0	2.92 GB
z-alue7065	35.0	9.07 GB	138.0	**33.5**	**431.18 MB**	138.0	398.36 MB

graph, only 1.08 GB of memory is needed for our approach and thus the memory usage is lower by at least a factor of 177. This drastic reduction of memory comes from the highly structured input: a perfect grid only has 2 far-apart pairs, the two pairs of opposing corners. Thus, the hyperbolicity computation only needs to consider these two pairs and computing the shortest paths between all pairs is hugely wasteful. Our approach includes this natural intuition to lazily compute the distances between the pairs of nodes to avoid huge running times and memory consumption in cases like these.

For a better overview, we plotted a comparison of the time and memory usage of both algorithms, see Figures 2 and 3. In Figure 2 we can see that the times indeed are very similar for both approaches except for the single outlier mentioned above. In Figure 3 we can see the drastic reduction in memory usage. Note again that the graphs included in this figure are only the graphs on which both algorithms terminate within the limits. The graphs where Borassi et al. algorithm exceeds the limits while our approach stays withing the limits are not shown. In particular, the memory reduction by at least a factor of 177 is not shown in this plot. Furthermore, we can see that with increasing memory consumption, also the advantage that our algorithm has over Borassi et al. algorithm with respect to memory usage becomes more and more pronounced.

Table 2 also lists the memory usage due to the BFS cache in our algorithm (last column). The intent of choosing a cache size of 1000 was to obtain better running times while having a reasonable memory consumption for big instances. This choice is questionable for some instances and we come back on that matter later on. Note however, that we obtain a much lower memory usage compared to Borassi et al. even with this choice of large cache.

5.5 Running time of parts

Additional to the comparison with the previous state-of-the-art algorithm, we also analyze the running time of the different parts of our algorithm. See Figure 4 for a visualization of how much running time as percentage of the total running time is spent in specific parts of our algorithm. First note that for each part of the algorithm, there exists and instance for which it is the bottleneck. However, we note that the initialization of our algorithm is only the bottleneck for instances that have a small total running time (see Table 2 for the running times). The other parts of our algorithm are also bottlenecks of harder instances. The most frequent bottleneck on our instances is the update of δ_L, which is the part of the algorithm where we explicitly check quadruples. This suggests that the approach of enumerating far apart pairs using previous pruning techniques is at its limit and a speed-up has to come from an improved pruning of quadruples.

The next pair computation appears to be the bottleneck in instances where very few pairs are far-apart so that there are very few quadruples to check. Note also that Borassi et al. algorithm is faster on instances where the computation of distances from x and y is the bottleneck (see Table 2). This is expected as the time wasted in recomputing some distances then becomes critical.

5.6 Impact of different optimizations

We also conduct experiments to obtain insights into the impact of the different optimizations that we used in our algorithm. In particular, we want to know the impact of the lower bound initialization (see Section 4.4.1), the cache size (see Section 4.4.2), and the pruning (see Section 4.4.3). We conduct experiments with all our graphs with the normal cache size of 1000 with no heuristic and no pruning, heuristic but no pruning, pruning but no heuristic, and heuristic and pruning. To gain

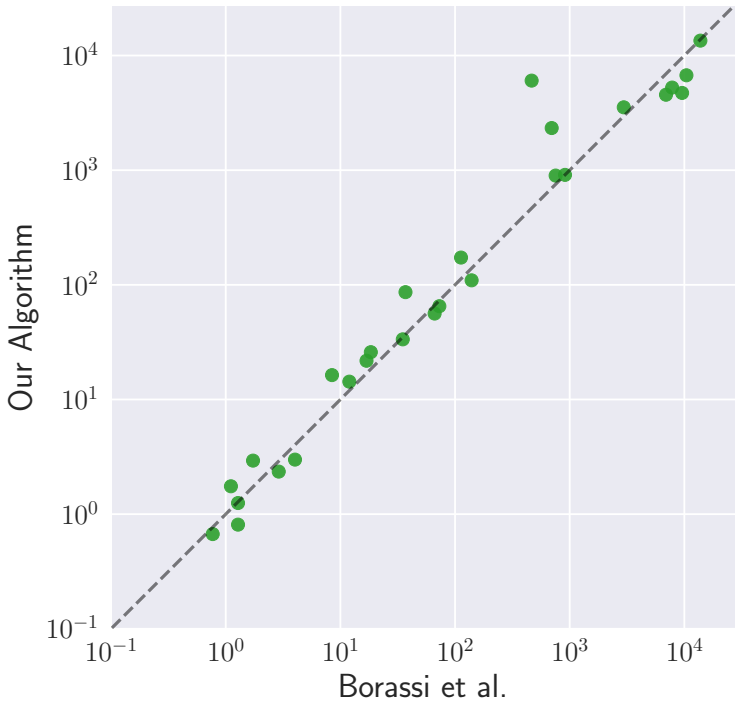


Figure 2: Comparing the *time* in seconds for computing the hyperbolicity on all graphs that finish using both algorithms. The dashed line is the identity, i.e., where both algorithms would take the same time.

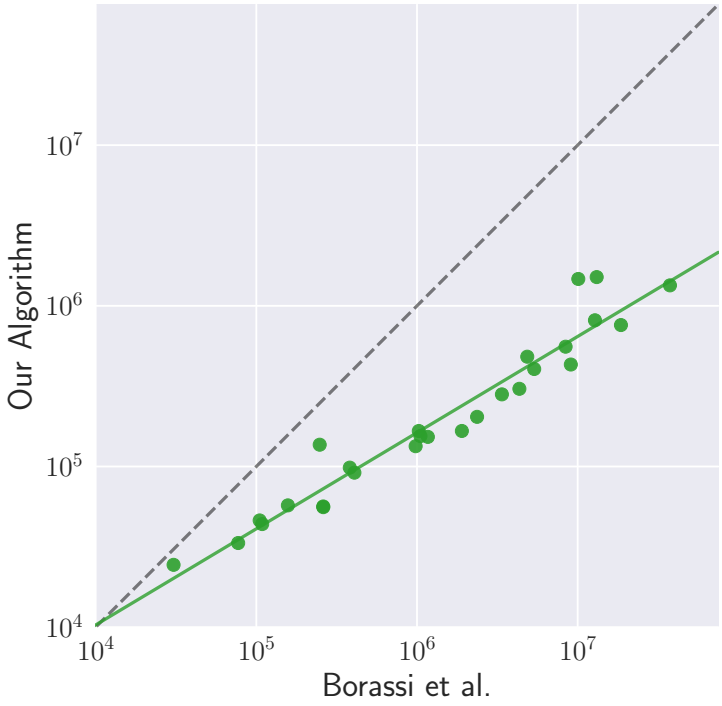


Figure 3: Comparing the *memory* in MB for computing the hyperbolicity on all graphs that finish using both algorithms. The dashed line is the identity, i.e., where both algorithms would use the same amount of memory. The solid green line ($2^x \mapsto 2^{5.38+0.6x}$) is a linear regression on the logarithmic coordinates. It suggests an exponential improvement in the memory consumption.

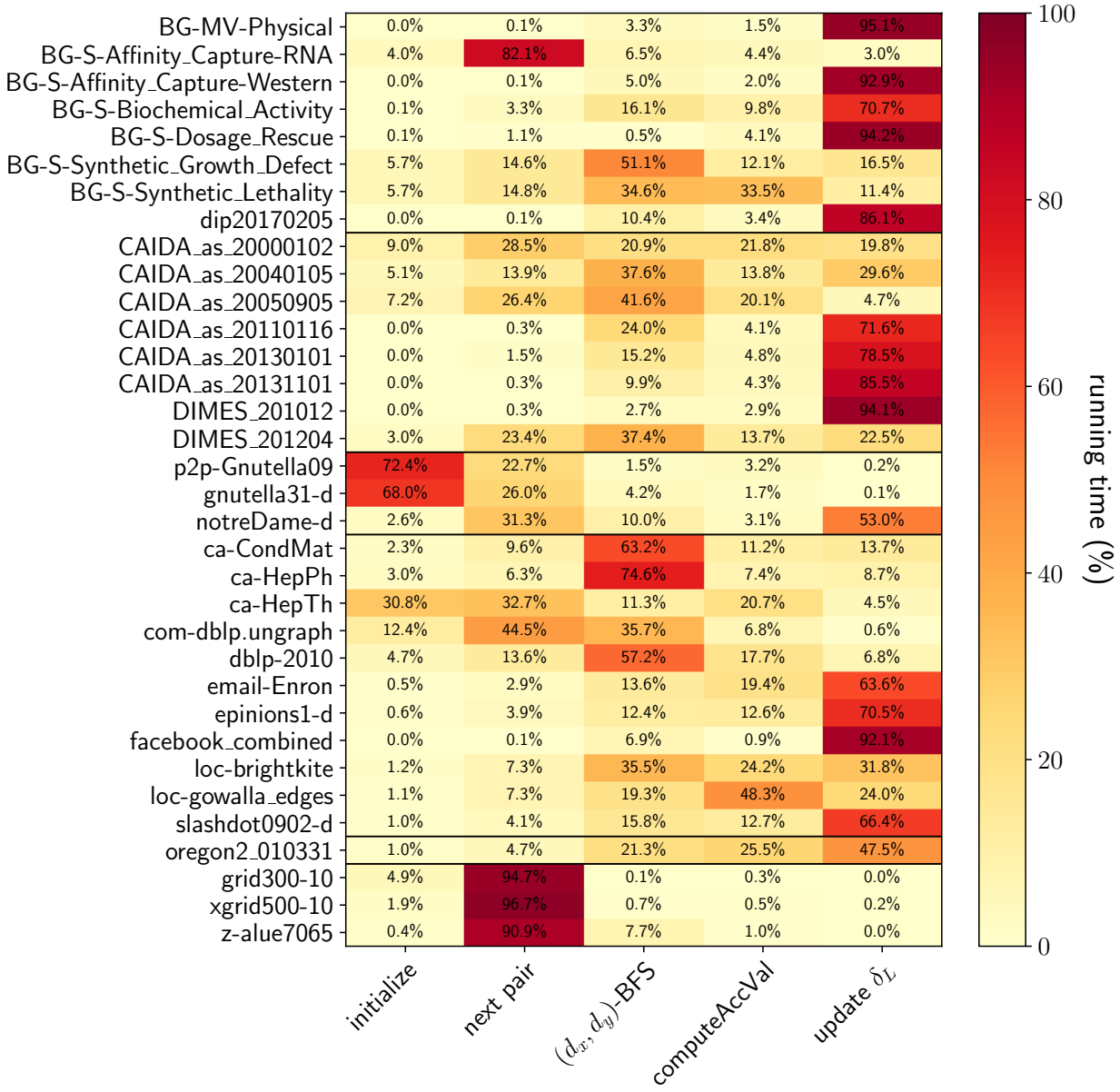


Figure 4: Percentage of time spent in different parts of our algorithm. Each row corresponds to the execution of the algorithm that was performed in Table 2 on the respective graph. The columns refer to the different parts of Algorithm 2. More precisely, they correspond to the following lines in the algorithm: initialize (l. 1-2), next pair (l. 3-4), (d_x, d_y)-BFS (l. 7-8), computeAccVal (l. 9), update (l. 10-13).

more insight into the effect of the BFS cache, we also conduct the last experiment with a cache size of 2, which means that we only store the current two BFSs in memory. See Table 3 and Figure 5 for the results of these experiments.

The overall results of these experiments are that there is no significant benefit for the running time when using the heuristically computed initial lower bound (see Figures 5.a-b). However, the pruning has significant impact depending on the graph: sometimes it does not help much, but in several cases it does decrease the running time by a larger factor approaching 2 and up to a factor of 5 (see Figure 5.c and the DIMES_201204 in Table 3). The cache optimization can reduce the running time by a factor of 1.4, but it also sometimes slightly increases it (see Figure 5.d). The benefit of a general caching strategy thus seems marginal compared to just storing distances from x while scanning (x, y) pairs with same x. Note that this simple strategy is equivalent to our caching optimization with a cache size of $c = 2$. Furthermore, while the cache does not dominate the memory consumption for most instances, it still increases the memory usage significantly, see the last column of Table 2.

5.7 BFS cache size experiments

To gain further insights into how the BFS cache we use affects the running time behavior, we run experiments with different cache sizes on selected graphs, see Figure 6. We again see different behavior on different instances. While the times monotonously decrease with increasing cache size for two of the graphs, for the other two we have increase-decrease and decrease-increase patterns. One reason for this behavior might be the cache size of the processor. In particular, if the graph already fits in the CPU cache (e.g., L1), then computing a BFS is quite fast. Especially for large BFS cache sizes which might push the graph out of the CPU cache and also might reside in a higher level CPU cache, the computation can become slower. Overall, the benefit of the cache appears to be rather low in terms of running time while it can be quite costly in terms of memory usage, advocating for choosing cache size $c = 2$ in the end as stated in the previous paragraph.

5.8 Far-apart pairs iterator experiments

Finally, we perform experiments to only analyze the behavior of the far-apart pairs iterator. To this end, we let the far-apart pairs iterator run on all our benchmark graphs and measure the time as well as memory consumption. Additionally, we are interested in the number of far-apart pairs that the different instances have as well as how many pairs of an instance are necessary for the hyperbolicity calculation of our new algorithm. As the graphs are of different sizes, we put the number of pairs in the last two columns in relation to the total number of node pairs in the graph. The results of these experiments are shown in Table 4. For all except four graphs, the far-apart pairs iterator runs through in the given memory and time limits. This is one more tractable instance compared to hyperbolicity computation. Note, however, that there are graphs for which we can compute the hyperbolicity but the far-apart pairs iterator does not run through within the limits (e.g., dblp-2010). This is explained by the fact that to compute the hyperbolicity, we can stop at some point of iterating through the far-apart pairs and do not have to compute them all. More specifically, we show in column "hyp pairs" what percentage of all pairs are necessary to compute the hyperbolicity with our algorithm on each instance. As we only consider mates as third and fourth nodes (i.e., previous far-apart pairs) in a quadruple, counting the number of far apart pairs until our algorithm terminates indeed counts all the pairs of nodes that we consider as first and

Table 3: Times for computing the hyperbolicity with different optimizations enabled. All entries are in seconds. The columns with "heur" use the lower bound initialization presented in Section 4.4.1, and the columns with "prune" use the pruning of Section 4.4.3. The value of c in the second row gives the size of the BFS cache presented in Section 4.4.2.

Graph	–	**heur**	**prune**	**heur & prune**	
	$c = 1000$	$c = 1000$	$c = 1000$	$c = 2$	$c = 1000$
BG-MV-Physical	7 040.0	7 047.0	6 745.0	6 867.0	6 725.0
BG-S-Affinity_Capture-MS	☠	☠	☠	☠	☠
BG-S-Affinity_Capture-RNA	0.7	0.7	0.7	0.7	0.7
BG-S-Affinity_Capture-Western	5 464.0	5 450.0	5 236.0	5 297.0	5 255.0
BG-S-Biochemical_Activity	16.5	16.5	16.2	20.2	16.3
BG-S-Dosage_Rescue	14.9	14.7	14.6	16.5	14.3
BG-S-Synthetic_Growth_Defect	3.2	3.3	3.0	4.6	2.9
BG-S-Synthetic_Lethality	2.1	2.2	1.8	2.5	1.8
dip20170205	19 088.0	19 144.0	18 492.0	18 473.0	18 318.0
CAIDA_as_20000102	1.2	1.3	1.2	1.7	1.2
CAIDA_as_20040105	39.9	40.0	25.7	35.0	25.9
CAIDA_as_20050905	35.6	36.9	21.7	23.8	21.8
CAIDA_as_20110116	16 079.0	16 132.0	13 700.0	13 412.0	13 491.0
CAIDA_as_20120101	☠	☠	☠	☠	☠
CAIDA_as_20130101	3 292.8	3 313.8	3 594.9	3 848.0	3 535.8
CAIDA_as_20131101	15 785.0	15 766.0	16 036.0	16 683.0	16 108.0
DIMES_201012	6 144.0	6 139.0	6 032.0	6 574.0	6 043.0
DIMES_201204	294.1	292.5	56.0	68.7	56.2
p2p-Gnutella09	2.0	2.4	1.9	2.3	2.4
gnutella31-d	172.2	191.2	91.1	104.7	109.5
notreDame-d	4 317.0	4 315.0	4 472.0	4 795.0	4 514.0
ca-CondMat	386.0	387.4	177.2	197.9	172.9
ca-HepPh	144.7	145.0	87.8	103.4	86.3
ca-HepTh	3.8	4.1	2.8	4.7	3.0
com-dblp.ungraph	☠	☠	5 352.0	5 713.0	5 449.0
dblp-2010	14 677.0	14 675.0	6 610.0	6 560.0	6 904.0
email-Enron	1 121.2	1 108.8	906.5	1 184.8	897.7
epinions1-d	2 626.6	2 607.0	2 331.4	3 563.1	2 331.3
facebook_combined	4 677.0	4 623.0	4 552.0	4 609.0	4 551.0
loc-brightkite	1 696.6	1 688.6	928.5	1 108.5	910.1
loc-gowalla_edges	☠	☠	14 259.0	13 169.0	14 478.0
slashdot0902-d	4 492.0	4 463.0	4 775.0	6 766.0	4 718.0
oregon2_010331	124.2	124.5	65.6	79.8	65.1
t.FLA-w	☠	☠	☠	☠	☠
buddha-w	☠	☠	☠	☠	☠
froz-w	☠	☠	☠	☠	☠
grid300-10	10.2	10.4	10.1	8.7	10.1
xgrid500-10	102.2	102.3	99.8	83.1	99.6
z-alue7065	36.5	37.0	33.8	33.6	33.5

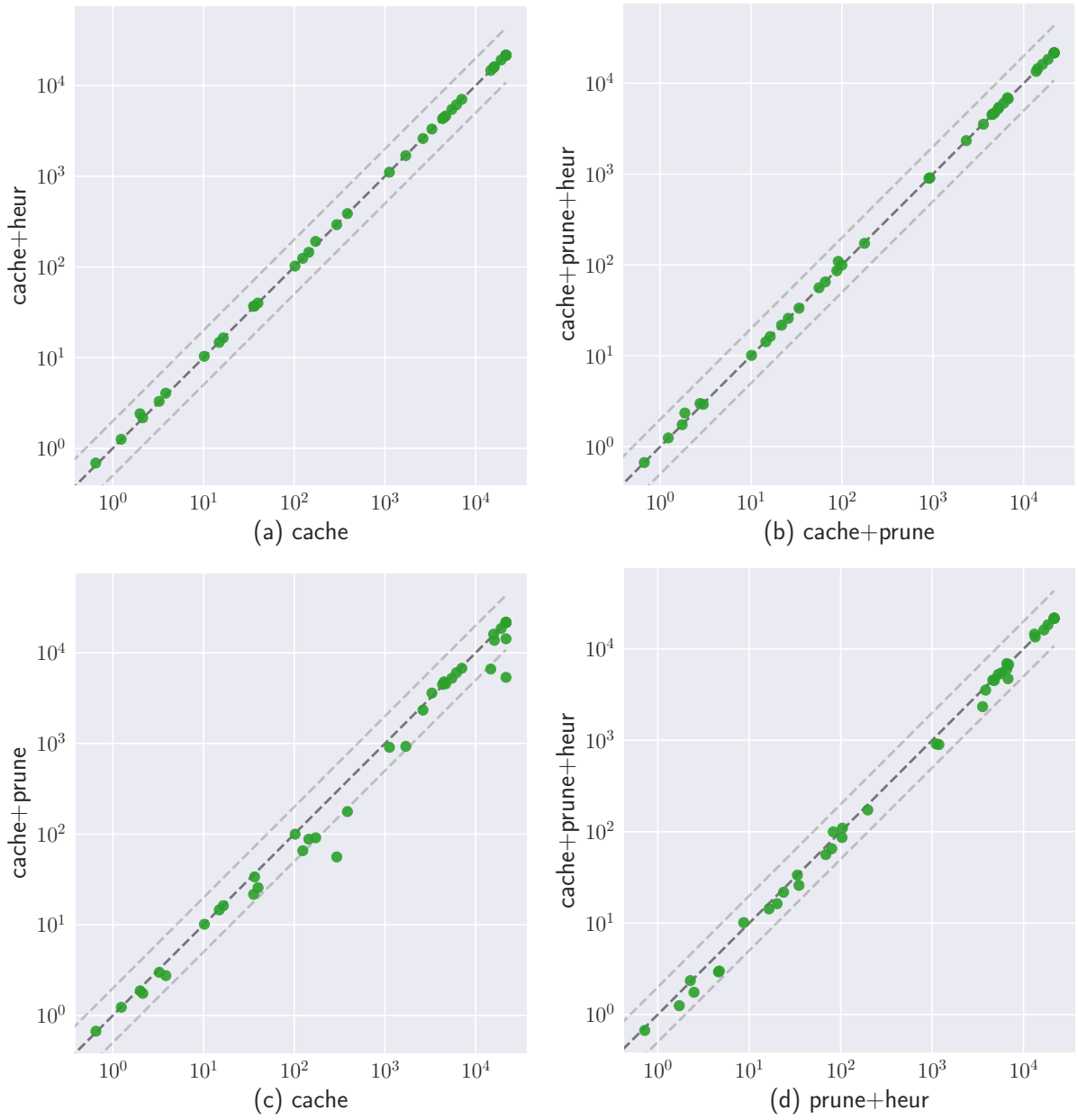


Figure 5: Comparison of various combinations of optimizations in terms of running time (in seconds) on all graphs that finish: "cache" refers to a cache of size $c = 1000$ (instead of $c = 2$), "prune" refers to pruning, and "heur" refers to the lower-bound heuristic. The light-gray dashed lines ($y = 2 * x$ and $y = x/2$) correspond to ratio 2.

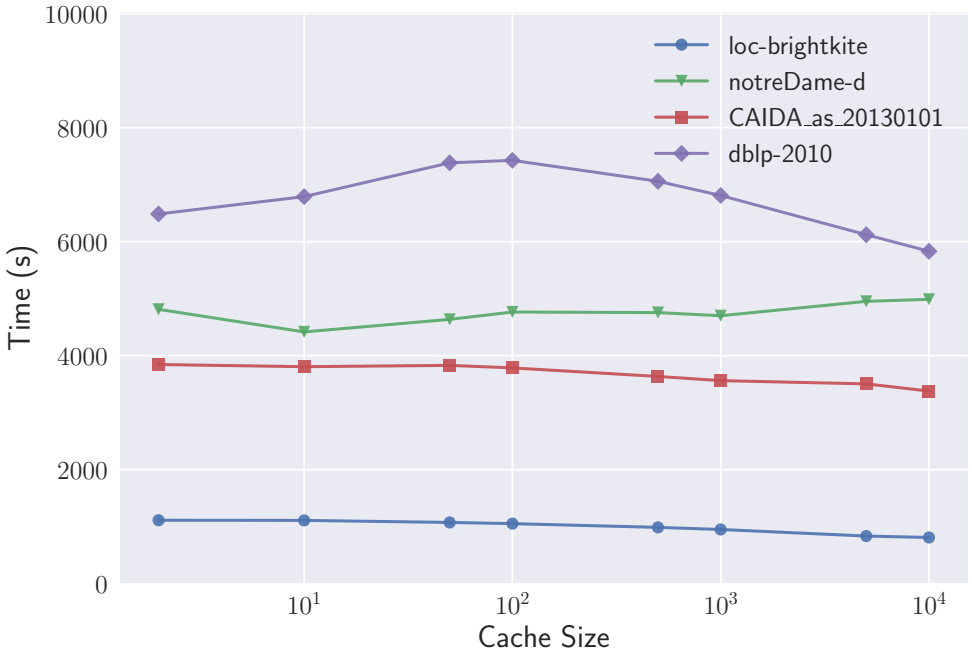


Figure 6: Plot for four graphs showing the running time development depending on the BFS cache size.

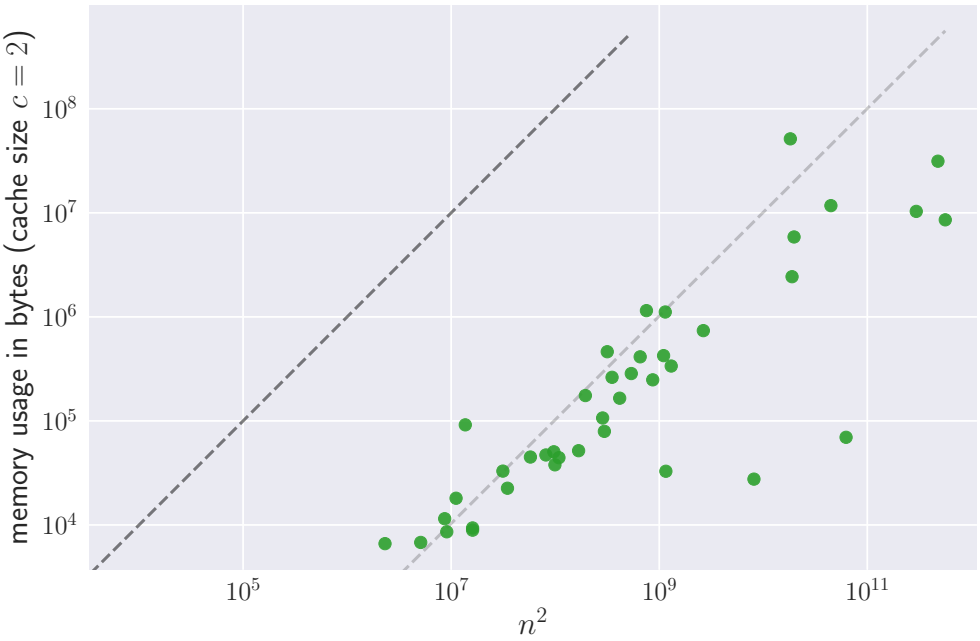


Figure 7: Memory consumption of hyperbolicity computation compared to the size n^2 of the distance matrix. We use a cache size of $c = 2$. The dashed line corresponds to identity while the light-gray dashed line corresponds to a factor 1/1000.

second tuple in the hyperbolicity computation. Observe that for the dblp-2010 graph, only around 0.27% of pairs are relevant for the hyperbolicity computation while 48.38% of the pairs are far-apart. In general, most instances only have a single-digit percentage or less of relevant pairs for the hyperbolicity computation while the fraction of far-apart pairs is above 24% for all graphs except grid-like graphs. Indeed, grid-like graphs (grid300-10, grid500-10 and z-alue7065) have less than 0.1% of far-apart pairs and even less relevant pairs. The grid graphs, grid300-10 and grid500-10, have such a small number of relevant pairs, that they are rounded to 0 in the precision that we choose for the numbers in the table. These low numbers of relevant pairs give the main explanation of the drastic memory reduction that we achieve with our algorithm. A second explanation comes from the fact that we store far-apart pairs whose distance remains greater than the hyperbolicity lower-bound δ_L. As we compute and store more far-apart pairs, the lower-bound increases and some previously computed far-apart pairs are cleared from memory. Figure 7 illustrates the maximum memory consumption of our hyperbolicity computation compared to the total number n^2 of pairs. We use a cache size $c = 2$ so that memory usage comes mainly from the storage of far-apart pairs. All in all, we can see that the memory consumption is lower than the size of the full distance matrix by roughly three orders of magnitudes for most graphs.

We can also see that for many graphs, the far-apart pairs iterator is very fast, while the hyperbolicity computation takes a long time, which is explained by the fact that we might spend quadratic time per pair to compute the hyperbolicity. Considering the percentage of far-apart pairs, we see that most graphs have roughly 30% to 70% far-apart pairs. The graphs with grid structure are extreme outliers, exhibiting a very low percentage of far-apart pairs. This can be explained by the grid structure, for which – considering a grid without missing edges – only the two pairs of opposing corners of the grid are far-apart. Furthermore, the facebook_combined graph has a very large percentage of far-apart pairs. This is explained by the fact that it has, by far, the highest average degree of all the graphs we consider. For such a well-connected graph, BFS trees have a very large number of leaves as shortest paths are not extended further from most nodes, which is necessary to produce such a large number of far-apart pairs.

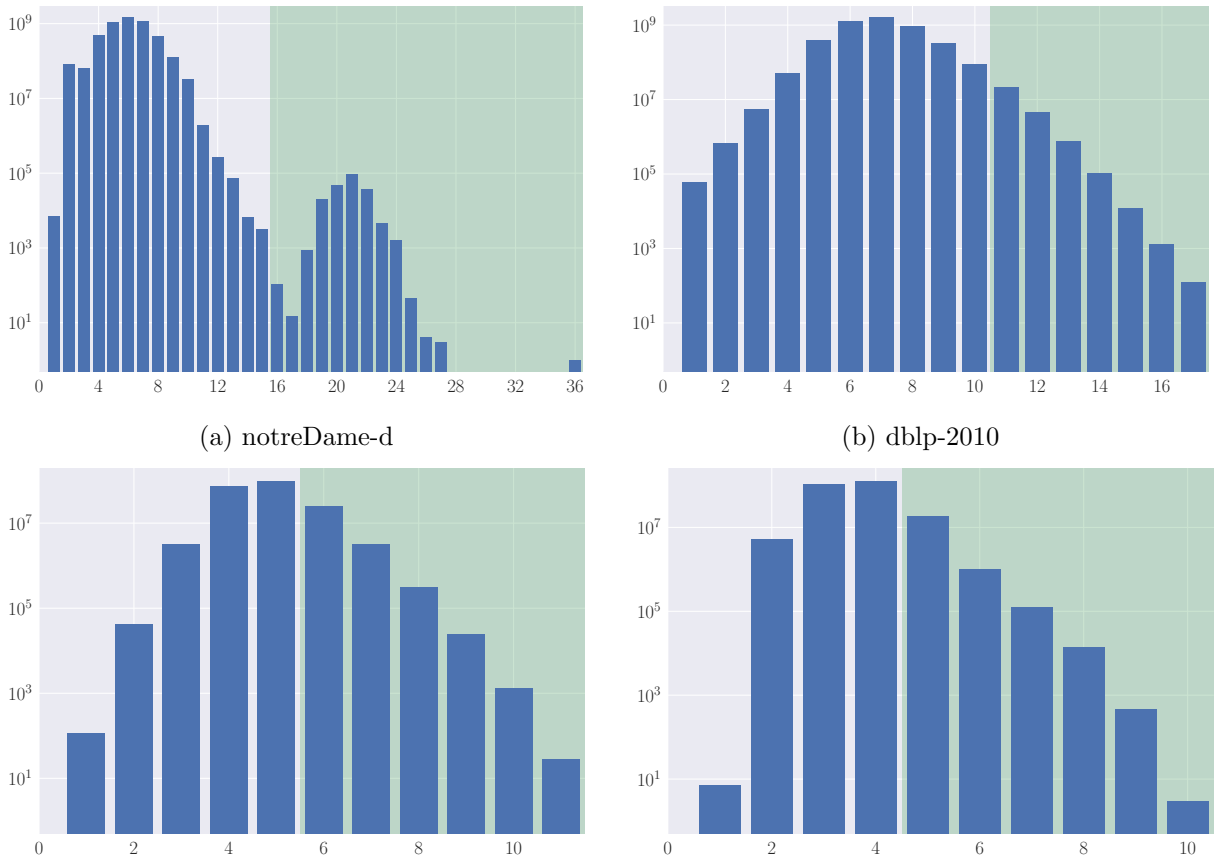
To further gain insights into the distribution of far-apart pairs, i.e., how they are distributed over various distances, we put all the distances between far-apart pairs into a histogram for four selected graphs, see Figure 8. While the distribution looks somewhat reasonable and expected for three of the four graphs, the notreDame-d graph shows an interesting distribution with two modes. Note that anomalies with respect to coreness were already found in this graph [58] where a special structure around a propeller-shaped subgraph was identified. We suspect that this structure connects a large fraction of pairs and could be related to this bi-modal observation. Finally, we also marked the part of the histogram that contains the far-apart pairs that are relevant for the hyperbolicity calculation. We observe that this region occurs after the peak of the histogram in all four instances that we show. In the notreDame-d instance, it even completely excludes the first, much larger mode.

6 Conclusion

In this work, we developed a fundamental algorithm to iterate over all far-apart pairs in non-increasing distance. As primary application we consider the computation of graph hyperbolicity. Our new algorithm enables us to compute, for the first time, the hyperbolicity of some graphs with more than a hundred thousand nodes with non trivial structure (e.g., notreDame-d, loc-

Table 4: Time and memory consumption of the far-apart pairs iterator only. We additionally show the percentage of far-apart pairs ("far pairs") and also the percentage of pairs (both with respect to the number of all node pairs in the graph) which have to be considered during the hyperbolicity computation of our algorithm ("hyp pairs").

Graph	**time (s)**	**memory**	**far pairs (%)**	**hyp pairs (%)**
BG-MV-Physical	21.9	391.23 MB	30.91	4.043
BG-S-Affinity_Capture-MS	94.3	1.33 GB	36.92	–
BG-S-Affinity_Capture-RNA	3.3	124.32 MB	70.86	0.602
BG-S-Affinity_Capture-Western	23.3	416.42 MB	32.42	5.355
BG-S-Biochemical_Activity	1.8	75.47 MB	42.95	8.196
BG-S-Dosage_Rescue	0.3	29.13 MB	24.67	5.223
BG-S-Synthetic_Growth_Defect	2.1	87.80 MB	46.12	17.973
BG-S-Synthetic_Lethality	1.1	54.61 MB	42.75	24.658
dip20170205	45.9	686.76 MB	30.12	11.245
CAIDA_as_20000102	4.4	150.73 MB	63.94	5.456
CAIDA_as_20040105	40.4	835.38 MB	67.35	5.806
CAIDA_as_20050905	65.7	1.25 GB	69.01	0.502
CAIDA_as_20110116	261.3	3.64 GB	69.56	40.066
CAIDA_as_20120101	361.9	4.39 GB	68.86	–
CAIDA_as_20130101	439.0	5.02 GB	68.89	5.292
CAIDA_as_20131101	490.4	5.81 GB	69.26	5.075
DIMES_201012	174.1	3.19 GB	74.89	13.431
DIMES_201204	144.6	2.40 GB	72.51	21.459
p2p-Gnutella09	6.5	190.00 MB	28.42	4.036
gnutella31-d	407.6	4.67 GB	29.49	3.787
notreDame-d	9 817.0	86.14 GB	54.38	0.002
ca-CondMat	105.1	1.49 GB	44.0	4.785
ca-HepPh	25.7	453.93 MB	42.31	8.630
ca-HepTh	7.8	196.91 MB	33.72	1.961
com-dblp.ungraph	–	☠	–	–
dblp-2010	14 008.0	87.83 GB	48.38	0.268
email-Enron	172.8	2.67 GB	55.87	10.950
epinions1-d	630.5	7.38 GB	45.46	6.683
facebook_combined	4.5	158.98 MB	89.08	71.779
loc-brightkite	425.1	4.45 GB	36.35	5.019
loc-gowalla_edges	10 509.0	63.43 GB	33.19	0.683
slashdot0902-d	1 442.0	15.35 GB	45.72	5.238
oregon2_010331	19.5	453.85 MB	66.43	31.757
t.FLA-w	–	☠	–	–
buddha-w	☠	–	–	–
froz-w	☠	–	–	–
grid300-10	345.4	4.24 GB	0.04	$5 \cdot 10^{-6}$
xgrid500-10	2 723.7	19.24 GB	0.04	$7 \cdot 10^{-6}$
z-alue7065	47.0	940.21 MB	0.06	0.002



(a) notreDame-d

(b) dblp-2010

(c) loc-brightkite

(d) CAIDA_as_20130101

Figure 8: Histograms of distribution of far-apart pairs for selected graphs. On the x-axis we plot the distance and on the y-axis the number of far-apart node pairs that have this distance. The green area shows the distance range for pairs that have to be evaluated in our hyperbolicity algorithm. We can see that this area always excludes the peak of the histogram in the shown examples.

gowalla_edges, com-dblp.ungraph). We reduce the memory usage significantly, while not compromising on performance. Non-trivial graphs with more than five hundred thousands nodes unfortunately still remain out of reach with our method. We thus plan to investigate alternative approaches in future work in order to get closer to the million nodes barrier. Furthermore, we believe that iterating over far-apart pairs in non-increasing distance is such a fundamental task, that our work will enable faster algorithms also in other settings.

References

[1] Muad Abu-Ata and Feodor F. Dragan. Metric tree-like structures in real-life networks: an empirical study. *Networks*, 67(1):49–69, 2016.

[2] Takuya Akiba, Yoichi Iwata, and Yuki Kawata. An exact algorithm for diameters of large real directed graphs. In *International Symposium on Experimental Algorithms - SEA*, volume 9125 of *Lecture Notes in Computer Science*, pages 56–67. Springer, 2015.

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *ACM SIGMOD International Conference on Management of Data - SIGMOD*, pages 349–360, 2013.

[4] Réka Albert, Bhaskar DasGupta, and Nasim Mobasheri. Topological implications of negative curvature for biological and social networks. *Physical Review E*, 89(3):032811, 2014.

[5] Hend Alrasheed and Feodor F. Dragan. Core-periphery models for graphs based on their δ-hyperbolicity: An example using biological networks. In *6th Workshop on Complex Networks - CompleNet*, volume 597 of *Studies in Computational Intelligence*, pages 65–77. Springer, 2015.

[6] Hans-Jürgen Bandelt and Henry Martyn Mulder. Distance-hereditary graphs. *Journal of Combinatorial Theory, Series B*, 41(2):182–208, 1986.

[7] Sergio Bermudo, José M. Rodríguez, José M. Sigarreta, and Jean-Marie Vilaire. Gromov hyperbolic graphs. *Discrete Mathematics*, 313(15):1575–1585, 2013.

[8] Anne Berry, Romain Pogorelcnik, and Geneviève Simonet. An introduction to clique minimal separator decomposition. *Algorithms*, 3(2):197–215, 2010.

[9] Marián Boguñá, Fragkiskos Papadopoulos, and Dmitri V. Krioukov. Sustaining the Internet with hyperbolic mapping. *Nature Communications*, 1(62):1–18, October 2010.

[10] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[11] Michele Borassi, Alessandro Chessa, and Guido Caldarelli. Hyperbolicity measures democracy in real-world networks. *Physical Review E*, 92(3):032812, 2015.

[12] Michele Borassi, David Coudert, Pierluigi Crescenzi, and Andrea Marino. On computing the hyperbolicity of real-world graphs. In *European Symposium on Algorithms - ESA*, volume 9294 of *Lecture Notes in Computer Science*, pages 215–226. Springer, September 2015.

[13] Michele Borassi, Pierluigi Crescenzi, and Michel Habib. Into the square: On the complexity of some quadratic-time solvable problems. *Electronic Notes in Theoretical Computer Science*, 322:51–67, 2016.

[14] Michele Borassi, Pierluigi Crescenzi, Michel Habib, Walter A. Kosters, Andrea Marino, and Frank W. Takes. Fast diameter and radius BFS-based computation in (weakly connected) real-world graphs: With an application to the six degrees of separation games. *Theoretical Computer Science*, 586:59–80, 2015.

[15] Norberto Castillo-García and Paula Hernández Hernández. *A New Heuristic Algorithm for the Vertex Separation Problem*, pages 487–500. Springer International Publishing, 2018.

[16] John Chakerian and Susan Holmes. Computational tools for evaluating phylogenetic and hierarchical clustering trees. *Journal of Computational and Graphical Statistics*, 21(3):581–599, 2012.

[17] Pierre Charbit, Fabien de Montgolfier, and Mathieu Raffinot. Linear time split decomposition revisited. *SIAM Journal on Discrete Mathematics*, 26(2):499–514, 2012.

[18] Victor Chepoi, Feodor F. Dragan, Bertrand Estellon, Michel Habib, and Yann Vaxès. Diameters, centers, and approximating trees of delta-hyperbolic geodesic spaces and graphs. In *Annual Symposium on Computational Geometry - SoCG*, pages 59–68. ACM, 2008.

[19] Victor Chepoi, Feodor F. Dragan, Bertrand Estellon, Michel Habib, Yann Vaxès, and Yang Xiang. Additive spanners and distance and routing labeling schemes for hyperbolic graphs. *Algorithmica*, 62(3-4):713–732, 2012.

[20] Victor Chepoi, Feodor F. Dragan, and Yann Vaxès. Core congestion is inherent in hyperbolic networks. In Philip N. Klein, editor, *ACM-SIAM Symposium on Discrete Algorithms - SODA*, pages 2264–2279. SIAM, 2017.

[21] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In David Eppstein, editor, *ACM-SIAM Symposium on Discrete Algorithms - SODA*, pages 937–946. ACM/SIAM, 2002.

[22] Nathann Cohen, David Coudert, Guillaume Ducoffe, and Aurélien Lancin. Applying clique-decomposition for computing gromov hyperbolicity. *Theoretical Computer Science*, 690:114–139, 2017.

[23] Nathann Cohen, David Coudert, and Aurélien Lancin. On computing the gromov hyperbolicity. *ACM Journal of Experimental Algorithmics*, 20:1–18, 2015.

[24] David Coudert and Guillaume Ducoffe. Recognition of C_4-free and 1/2-hyperbolic graphs. *SIAM Journal on Discrete Mathematics*, 28(3):1601–1617, September 2014.

[25] David Coudert and Guillaume Ducoffe. Revisiting Decomposition by Clique Separators. *SIAM Journal on Discrete Mathematics*, 32(1):682 – 694, January 2018.

[26] David Coudert, Guillaume Ducoffe, and Alexandru Popa. Fully polynomial FPT algorithms for some classes of bounded clique-width graphs. *ACM Transactions on Algorithms*, 15(3):1–57, June 2019.

[27] David Coudert, Dorian Mazauric, and Nicolas Nisse. Experimental evaluation of a branch-and-bound algorithm for computing pathwidth and directed pathwidth. *ACM Journal of Experimental Algorithmics*, 21(1):1.3:1–1.3:23, 2016.

[28] David Coudert, André Nusser, and Laurent Viennot. Hyperbolicity (version 1.0). https://gitlab.inria.fr/dcoudert/hyperbolicity/, 2021.

[29] Pierluigi Crescenzi, Roberto Grossi, Michel Habib, Leonardo Lanzi, and Andrea Marino. On computing the diameter of real-world undirected graphs. *Theoretical Compututer Science*, 514:84–95, 2013.

[30] William H. Cunningham. Decomposition of directed graphs. *SIAM Journal on Algebraic Discrete Methods*, 3(2):214–228, 1982.

[31] William H. Cunningham and Jack Edmonds. A combinatorial decomposition theory. *Canadian Journal of Mathematics*, 32(3):734–765, 1980.

[32] Bhaskar DasGupta, Marek Karpinski, Nasim Mobasheri, and Farzane Yahyanejad. Effect of Gromov-hyperbolicity parameter on cuts and expansions in graphs and some algorithmic implications. *Algorithmica*, 80(2):772–800, 2018.

[33] Pierre de La Harpe and Etienne Ghys. *Sur les groupes hyperboliques d'après Mikhael Gromov*, volume 83. Progress in Mathematics, 1990.

[34] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Robust distance queries on massive networks. In *European Symposium on Algorithms - ESA*, volume 8737 of *Lecture Notes in Computer Science*, pages 321–333. Springer, 2014.

[35] Josep Díaz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34(3):313–356, September 2002.

[36] Reinhard Diestel. *Graph Theory, 5th edition*, volume 173 of *Graduate Texts in Mathematics*. Springer, Heidelberg, 2017.

[37] Feodor F. Dragan. Tree-like structures in graphs: A metric point of view. In *39th International Workshop on Graph-Theoretic Concepts in Computer Science - WG*, volume 8165 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2013.

[38] Feodor F. Dragan, Michel Habib, and Laurent Viennot. Revisiting radius, diameter, and all eccentricity computation in graphs through certificates. *CoRR*, abs/1803.04660, 2018.

[39] Andreas Dress, Katharina Huber, Jacobus Koolen, Vincent Moulton, and Andreas Spillner. *Basic Phylogenetic Combinatorics*. Cambridge University Press, Cambridge, UK, December 2011.

[40] Hervé Fournier, Anas Ismail, and Antoine Vigneron. Computing the gromov hyperbolicity of a discrete metric space. *Information Processing Letters*, 115(6):576–579, 2015.

[41] Tibor Gallai. Transitiv orientierbare graphen. *Acta Mathematica Hungarica*, 18(1):25–66, 1967.

[42] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1979.

[43] Micha Gromov. Hyperbolic groups. In S.M. Gersten, editor, *Essays in Group Theory*, volume 8 of *Mathematical Sciences Research Institute Publications*, pages 75–263. Springer, New York, 1987.

[44] Michel Habib and Christophe Paul. A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, 4(1):41–59, 2010.

[45] Edward Howorka. On metric properties of certain clique graphs. *Journal of Combinatorial Theory, Series B*, 27(1):67–74, 1979.

[46] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, December 2001.

[47] W. Sean Kennedy, Iraj Saniee, and Onuttom Narayan. On the hyperbolicity of large-scale networks and its estimation. In *International Conference on Big Data*, pages 3344–3351. IEEE, 2016.

[48] Robert Krauthgamer and James R. Lee. Algorithms on negatively curved spaces. In *IEEE Symposium on Foundations of Computer Science - FOCS*, pages 119–132. IEEE, 2006.

[49] François Le Gall. Faster algorithms for rectangular matrix multiplication. In *IEEE Symposium on Foundations of Computer Science - FOCS*, pages 514–523, New Brunswick, NJ, USA, 2012. IEEE.

[50] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. Exacting eccentricity for small-world networks. In *IEEE International Conference on Data Engineering - ICDE*, pages 785–796, April 2018.

[51] Tamara Munzner and Paul Burchard. Visualizing the structure of the world wide web in 3d hyperbolic space. In David R. Nadeau and John L. Moreland, editors, *Symposium on Virtual Reality Modeling Language - VRML*, pages 33–38. ACM, 1995.

[52] Damien Noguès. δ-hyperbolicité et graphes. Master's thesis, MPRI, Université Paris 7, 2009.

[53] Rose Oughtred, Chris Stark, Bobby-Joe Breitkreutz, Jennifer Rust, Lorrie Boucher, Christie Chang, Nadine Kolas, Lara O'Donnell, Genie Leung, Rochelle McAdam, et al. The biogrid interaction database: 2019 update. *Nucleic acids research*, 47(D1):D529–D541, 2019.

[54] Jordi Petit. Addenda to the survey of layout problems. *Bulletin of the EATCS*, 105:177–201, 2011.

[55] Lukasz Salwinski, Christopher S. Miller, Adam J. Smith, Frank K. Pettit, James U. Bowie, and David Eisenberg. The database of interacting proteins: 2004 update. *Nucleic acids research*, 32(suppl_1):D449–D451, 2004.

[56] Yuval Shavitt and Eran Shir. DIMES: Let the internet measure itself. *ACM SIGCOMM Computer Communication Review*, 35(5):71–74, October 2005.

[57] Yuval Shavitt and Tomer Tankel. On the curvature of the internet and its usage for overlay construction and distance estimation. In *Annual Joint Conference of the IEEE Computer and Communications Societies - INFOCOM*, 2004.

[58] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. Patterns and anomalies in k-cores of real-world graphs with applications. *Knowl. Inf. Syst.*, 54(3):677–710, 2018.

[59] Mauricio Abel Soto Gómez. *Quelques propriétés topologiques des graphes et applications à internet et aux réseaux*. PhD thesis, Univ. Paris Diderot (Paris 7), 2011.

[60] Frank W. Takes and Walter A. Kosters. Computing the eccentricity distribution of large graphs. *Algorithms*, 6(1):100–118, 2013.

[61] Robert Endre Tarjan. Decomposition by clique separators. *Discrete Mathematics*, 55(2):221–232, 1985.

[62] The Cooperative Association for Internet Data Analysis (CAIDA). The CAIDA AS relationships dataset. http://www.caida.org/data/active/as-relationships/, 2013.

[63] Virginia Vassilevska Williams, Joshua R. Wang, Richard Ryan Williams, and Huacheng Yu. Finding four-node subgraphs in triangle time. In *ACM-SIAM Symposium on Discrete Algorithms - SODA*, pages 1671–1680. SIAM, 2015.

[64] Jörg A. Walter and Helge J. Ritter. On interactive visualization of high-dimensional data using the hyperbolic plane. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD*, pages 123–132. ACM, 2002.

A Time and space trade-offs for determining all far-apart pairs

In this section, we present algorithms offering different time and space trade-offs for the problem of computing all far-apart pairs. The space complexity considered here is the working memory, hence excluding the space needed to store the result, unless needed during computations.

A first algorithm to determine the set of far-apart pairs is to: 1) determine for each node $u \in V$ the set F_u of u-far nodes using BFS, and then 2) check for each node $v \in F_u$ if u is v-far (i.e., if $u \in F_v$). This can be done in time $\mathcal{O}(nm)$ and space $\mathcal{O}(n^2)$ since $|F_u| \in \mathcal{O}(n)$.

Another algorithm is to execute two steps for each node $u \in V$: 1) determine the set F_u of u-far nodes using BFS, and then 2) for each node $v \in F_u$, check if there is $w \in N(u)$ such that $\mathrm{d}(u, v) < \mathrm{d}(w, v)$. The second step requires to compute distances from w and so the time complexity of this algorithm is $\mathcal{O}((n+m)\sum_{u \in V}(1 + |N(u)|)) = \mathcal{O}(m^2)$. Observe that during the processing of each node u, this algorithm stores the set F_u, the distances from u, and the distances from one neighbor w of u. Hence, the space complexity is $\mathcal{O}(n)$. The second method can be improved using the bit-parallel BFS proposed in [3]. Indeed, this algorithm computes simultaneously distances from u and b of its neighbors in time $\mathcal{O}(n+m)$ and space $\mathcal{O}(n)$, assuming that b is a constant and using bitwise operations on bit vectors of size b (typically 32 or 64). Hence, the number of BFSs to perform is reduced to $\sum_{u \in V} \lceil |N(u)|/b \rceil$.

Let us now show how to modify the above algorithm to obtain an algorithm with time complexity in $\mathcal{O}(nm)$ and space complexity in $\mathcal{O}(n\,\mathrm{pw}(G))$, where $\mathrm{pw}(G)$ denotes the pathwidth of G [35, 54]. The main idea is to compute the distances from u only once and to store them during as few

iterations as possible. For that, let $\pi : V \to [n]$ be a linear ordering of the nodes (i.e., a bijective mapping) related to the pathwidth as described later and let $\Pi(V)$ be the set of all such orderings. The algorithm iterates over the nodes in the order $\pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(n)$. The distances from u are used for the processing of node u, and for the processing of each neighbor $w \in N(u)$. Let $w_L \coloneqq \arg\min_{w \in N(u)} \pi(w)$ and $w_R \coloneqq \arg\max_{w \in N(u)} \pi(w)$. Hence, distances from u must be computed at iteration $\pi(w_L)$ and stored until iteration $\pi(w_R)$. Consequently, at iteration i, we have stored distances from all the nodes in

$$\{u \in V \mid \exists uv \in E \text{ s.t. } \pi(u) < i \leq \pi(v)\} \cup \{\pi^{-1}(i)\} \cup \{v \in V \mid \exists uv \in E \text{ s.t. } \pi(u) \leq i \leq \pi(v)\}.$$

Now observe that the *pathwidth* [35, 54] of a graph G is defined as $\mathrm{pw}(G) = \min_{\pi \in \Pi(V)} p(G, \pi)$, where $p(G, \pi) = \max_{i=1}^{n} |\{u \in V \mid \exists uv \in E \text{ such that } \pi(u) < i \leq \pi(v)\}|$. Hence, at iteration i we store distances from at most $2p(G, \pi) + 1$ nodes and there is an ordering ensuring to store distances from at most $2\,\mathrm{pw}(G) + 1$ nodes. Consequently, the space complexity of this algorithm is in $\mathcal{O}(n\,\mathrm{pw}(G))$ and its time complexity is $\mathcal{O}(nm)$ assuming that the ordering π such that $p(G, \pi) = \mathrm{pw}(G)$ is given. However, the problem of determining an ordering π such that $p(G, \pi) = \mathrm{pw}(G)$ is NP-complete [35, 54]. Nonetheless, efficient heuristic algorithms have been proposed [27, 15].

Finally, recall that the *bandwidth* of a graph G is defined as $\mathrm{bw}(G) = \min_{\pi \in \Pi(V)} b(G, \pi)$, where $b(G, \pi) = \max_{uv \in E} |\pi(u) - \pi(v)|$ [35, 54]. Therefore, distances from u are stored for at most $2b(G, \pi) + 1$ iterations, and there is an ordering ensuring that this number of iterations is at most $2\,\mathrm{bw}(G) + 1$. However, the problem of determining such an ordering is NP-hard [42].

B Retrieving distances from far nodes

First note the following corollary which is a direct consequence of Lemma 1:

Corollary 5. *For any $v \in V$, let F_v be the set of v-far nodes. The number of leaves of any shortest path tree rooted at v is at least $|F_v|$.*

Observe however that the lower bound of Corollary 5 does not imply the existence of a shortest path tree with $|F_v|$ leaves. For instance, consider the 4-cycle (u_1, u_2, u_3, u_4). We have $|F_{u_1}| = |\{u_3\}| = 1$, but all shortest path trees rooted at u_1 have 2 leaves (either $\{u_2, u_3\}$ or $\{u_3, u_4\}$).

We now show that, given the v-far nodes and a constant c, one can determine the nodes at distance at least c from v.

Lemma 8. *Let $v \in V$ and let S be the set of all nodes u for which $\mathrm{d}(u, v) \geq c$. Given the set F_v of v-far nodes and the distances from v to each of these nodes, one can compute the set S in time $\mathcal{O}(|F_v| + \sum_{u \in S} |N(u)|)$.*

Proof. First, observe that if u is v-far, then for each $w \in N(u)$ it holds that

$$\mathrm{d}(v, u) - 1 \leq \mathrm{d}(v, w) \leq \mathrm{d}(v, u).$$

Furthermore, a neighbor $w \in N(u)$ with $\mathrm{d}(v, w) = \mathrm{d}(v, u) - 1$ is not v-far, and if a neighbor $w' \in N(u)$ is v-far then $\mathrm{d}(v, u) = \mathrm{d}(v, w')$.

To determine all the nodes that are at distance at least c from v, it suffices to perform a reverse breadth-first search, starting from far nodes. More precisely, let $\{L_d\}_{d \in \{c, \ldots, \mathrm{ecc}(v)+1\}}$ be a set family where L_d is initialized with the v-far nodes at distance d from v, and let $L_{\mathrm{ecc}(v)+1} = \emptyset$. Then,

consider these sets in decreasing distance d from v, starting with $d = \mathrm{ecc}(v)$, and stopping when $d = c - 1$. For each node $u \in L_d$, add each $w \in N(u)$ to L_{d-1} for which $w \notin L_d \cup L_{d+1}$. At the end of this procedure, we have that if $\mathrm{d}(v, u) = d$, then $u \in L_d$ and all the sets are disjoint, i.e. $L_d \cap L_{d'} = \emptyset$ for any $c \leq d, d' \leq \mathrm{ecc}(v)$ with $d \neq d'$. The claimed time bound follows immediately from the description of the algorithm. □