A Streaming Algorithm for the Undirected Longest Path Problem[*]

Lasse Kliemann[1], Christian Schielke[2], and Anand Srivastav[3]

1 **Kiel University, Faculty of Engineering, Department of Computer Science, Kiel, Germany**
`lki@informatik.uni-kiel.de`
2 **Kiel University, Faculty of Engineering, Department of Computer Science, Kiel, Germany**
`csch@informatik.uni-kiel.de`
3 **Kiel University, Faculty of Engineering, Department of Computer Science, Kiel, Germany**
`asr@informatik.uni-kiel.de`

Abstract

We present the first streaming algorithm for the longest path problem in undirected graphs. The input graph is given as a stream of edges and RAM is limited to only a linear number of edges at a time (linear in the number of vertices n). We prove a per-edge processing time of $\mathcal{O}(n)$, where a naive solution would have required $\Omega(n^2)$. Moreover, we give a concrete linear upper bound on the number of bits of RAM that are required.

On a set of graphs with various structure, we experimentally compare our algorithm with three leading RAM algorithms: Warnsdorf (1823), Pohl-Warnsdorf (1967), and Pongrácz (2012). Although conducting only a small constant number of passes over the input, our algorithm delivers competitive results: with the exception of preferential attachment graphs, we deliver at least 71% of the solution of the best RAM algorithm. The same minimum relative performance of 71% is observed over *all* graph classes after removing the 10% worst cases. This comparison has strong meaning, since for each instance class there is one algorithm that on average delivers at least 84% of a Hamilton path. In some cases we deliver even better results than any of the RAM algorithms.

1998 ACM Subject Classification G.2.2 [Graph Theory] Graph Algorithms

Keywords and phrases Streaming Algorithms, Undirected Longest Path Problem, Graph Algorithms, Combinatorial Optimization

Digital Object Identifier 10.4230/LIPIcs.ESA.2016.56

1 Introduction

Let $G = (V, E)$ be an undirected, simple, and finite graph (all our graphs are of this type). A *path* of length k in G is a sequence of k distinct vertices $(v_1, \dots, v_k)$ such that $v_i v_{i+1} \in E$ for each $1 \leq i < k$. (We write uv or vu for the undirected edge $\{u, v\}$ between vertices u and v.) In the *longest path problem* (LPP), we ask for a path in G that has maximum length among all the paths in G. This problem is NP-hard as seen easily by a reduction from the Hamilton cycle problem. A long line of research has investigated its approximability,

[*] Supported by German Research Foundation (DFG), Grant SR7/15-1 *Algorithmic Foundations for Genome Assembly.*

and several polynomial-time heuristics and algorithms with proven worst-case guarantee are known. With increasing size of the input graph, not only the time complexity of an algorithm becomes important but also its space complexity. For large graphs G, e. g., graphs that occur in genome assembly, it is not realistic anymore to assume that G can be fully stored in (fast) random-access memory (RAM). Instead we should assume that it can only be accessed efficiently in a *sequential* manner. The *graph streaming model* formalizes this. Here, the graph is given as a sequence $e_1, \ldots, e_m$ of its edges, and access is only provided in the form of *passes*: a *pass* means each edge in the sequence is presented to the algorithm once. RAM is restricted to $\mathcal{O}(n \cdot \operatorname{poly}\log(n))$ bits, so essentially we can store a number of edges linear in the number n of vertices. Besides the solution quality, the most important property of a streaming algorithm is the number of passes that it has to conduct in order to obtain a good solution. To be practical, this number should be a small constant. This model does not only make sense when the input is stored on a disk or remote server, but also in the context of memory hierarchies, where cache RAM is several magnitudes faster than main RAM.

The standard graph search techniques breadth-first search (BFS) and depth-first search (DFS) cannot be done in the streaming model within a constant number of passes [13]. However, all relevant existing algorithms for the LPP use some form of graph search and/or require super-linear data structures (as in the dynamic programming part for color coding [1]). We therefore take a different approach that will never have to do a graph search on a graph with more than a linear number of edges and moreover does not require more than a linear amount of RAM for additional data structures.

1.1 Our Contribution

We give a streaming algorithm for the longest path problem in undirected graphs with a proven per-edge processing time of $\mathcal{O}(n)$. Our algorithm works in two phases, which we outline here briefly and explain in detail in Section 3. In the first phase, global information on the graph is gathered in form of a constant number of spanning trees[1] $T_1, \ldots, T_\tau$. This is possible in the streaming model since roughly speaking, for a spanning tree we can 'take edges as they come'. A spanning tree can be constructed in just one pass – we however use multiple passes and limit the maximum degree during the first passes in order to favor path-like structures and avoid clusters of edges. Experiments clearly indicate that this degree-limiting is essential for solution quality. The spanning trees fit into RAM, since we consider τ as constant (we will in fact have $\tau = 1$ or $\tau = 2$ in the experiments). After construction of the τ trees, they are merged into one graph U by taking the union of their edges. Then we use standard algorithms to determine a long path P in U, isolate P, and finally add enough edges around P to obtain a tree T.

Then, in the second phase, we conduct further passes during which we test if the exchange of single edges of T can improve the longest path in it. (A longest path in a tree can be found by conducting DFS two times [10]; the length of a longest path in a tree is its diameter.) The main challenge in the second phase is to quickly determine which edges should be exchanged. We show that this decision can be made in linear time, hence yielding a per-edge processing time of $\mathcal{O}(n)$.

An experimental study is conducted on randomly generated instances with different structure, including ones created with the recently published generator for hyperbolic geometric

[1] For simplicity, we always assume that our input graphs are connected, but it would be easy to adapt our algorithm to the general case.

random graphs [30]. Different variants of our streaming algorithm are compared with four RAM algorithms: Warnsdorf and Pohl-Warnsdorf (two related classical heuristics [24, 25]), Pongrácz (a recently published heuristic [26]), and a simple randomized DFS. Experiments show that although we never do more than 11 passes, results delivered by our algorithm are competitive. We deliver at least 71% of the best result delivered by any of the tested RAM algorithms, with the exception of preferential attachment graphs. By considering low percentiles, we observe a similar quality without any restriction on the graph class. This is a good result also in absolute terms, since we observe that for each graph class and set of parameters, there is one algorithm that on average gives a path of length $0.84 \cdot n$, i. e., 84% of a Hamilton path. On some graph classes, we outperform any of the tested RAM algorithms, which makes our algorithm interesting even outside of the streaming setting. A detailed discussion of results is given in Section 7.

1.2 Previous and Related Work

Algorithms for the LPP have been studied extensively in the RAM model. We start listing algorithms with proven guarantees. Bodlaender [9] and Monien [22] gave algorithms that find a path of length k (if it exists) in $\mathcal{O}(2^k k!\, n)$ and $\mathcal{O}(k!\, nm)$ time, respectively. Alon et al. [1] introduced the method of *color coding* and based on that gave an algorithm running in expected time $2^{\mathcal{O}(k)} n$. There is a recent randomized algorithm by Björklund et al. [7] that given k, finds a path of length k (if it exists) in $\mathcal{O}(1.66^k \cdot \operatorname{poly} n)$ time (see also [19, 32, 5]). Those works show that the problem is fixed-parameter tractable: a path of length k can be found (if it exists) in polynomial time, for fixed k. The particular dependence of the running time on k (factorial or exponential) determines up to which k we stay polynomial and thus determines the length guarantee for a polynomial-time approximation algorithm.

In Hamiltonian graphs, a path of length $\Omega\Big(\big(\frac{\log(n)}{\log\log(n)}\big)^2\Big)$ can be found with the algorithm by Vishwanathan [29]; and Feder et al. gave further results for sparse Hamiltonian graphs [11]. Björklund and Husfeldt [6] gave an algorithm that finds a path of length $\Omega\big(\frac{(\log(\mathrm{opt}))^2}{\log\log(\mathrm{opt})}\big)$, where opt is the length of a longest path. It works by a decomposition of the graph into paths and cycles. Their technique subsequently was extended by Gabow [14] and Gabow and Nie [15] yielding guarantees for the length of the path of $\exp\big(\Omega\big(\sqrt{\frac{\log(\mathrm{opt})}{\log\log(\mathrm{opt})}}\big)\big)$ and $\exp(\Omega(\sqrt{\log(\mathrm{opt})}))$, respectively. Apart from that, the field is dominated by heuristics, such as (Pohl-)Warnsdorf [24, 25] and Pongrácz [26].

The Björklund-Husfeldt algorithm uses color coding as an important subroutine. We implemented and tested a simple algorithm based on color coding, which gave inferior results and more importantly took very long time to complete, substantially longer than (Pohl-)Warnsdorf, Pongrácz, or our algorithm. Details will be given in the full version. Thus we refrained from further implementing the Björklund-Husfeldt algorithm. (The original description in [6] uses Bodlaender's algorithm [9], which has an even higher running time than color coding.) The Gabow-Nie algorithm [15] does not use color coding, but at the time of writing was only available as a short conference version, making it difficult to implement.

Several non-approximability results have been shown by Karger et al. [17]: a constant-factor approximation is NP-hard; and for any $\varepsilon > 0$, the LPP cannot be approximated with a ratio of $2^{\mathcal{O}(\log^{1-\varepsilon}(n))}$, unless $\mathrm{NP} \subseteq \mathrm{DTIME}(2^{\mathcal{O}(\log^{1/\varepsilon}(n))})$, that is, such an approximation is quasi-NP-hard. Bazgan et al. showed that the same holds even when restricting to cubic Hamiltonian graphs [4].

The LPP is also interesting in directed graphs. For any $\varepsilon > 0$, it is NP-hard to approximate in directed graphs within $n^{1-\varepsilon}$ [8]. The best approximation guarantee in the directed case (unless restricting to special classes of graphs) is still the color coding algorithm that also works in the undirected case [1]. For special graph classes, there exist exact polynomial-time algorithms, e. g., for (undirected) trees (given by Dijkstra around 1960, see [10] for a proof), for directed acyclic graphs [27, pp. 661-666], for grid graphs [18], and for cactus graphs [28, 21].

The study of graph problems in streaming models started around the beginning of the 21st century, see [2, 12] for early works. The idea of using a linear amount of memory is due to Muthukrishnan [23]. Up to then, the ‘streaming’ term was associated with sub-linear memory, which is not enough for many graph problems [12]. To emphasize the difference, the streaming model with linear RAM (that we use) is also referred to as the *semi-streaming model* in the literature.

Since then, many kinds of graph problems have been addressed, such as shortest paths, spanning trees, connectivity, cuts, matching, and vertex cover. Several lower bounds are known. Most importantly for us, Feigenbaum et al. [13] proved that any BFS algorithm computing the first k layers with probability at least $2/3$, requires more than $k/2$ passes if staying within $\mathcal{O}(n \cdot \operatorname{poly}\log(n))$ memory (see Guruswami and Onak [16] for improved lower bounds). This constitutes a substantial hurdle when transferring existing algorithms into the streaming model. To the best of our knowledge, longest paths have not been addressed before in a streaming model.

It must be emphasized that streaming techniques also make sense when the graph is of size $c \cdot n \cdot \log(n)$ if a streaming algorithm can guarantee to stay within $c' \cdot n \cdot \log(n)$ for $c' < c$. Therefore, we give a memory guarantee for our algorithm using concrete constants.

1.3 Ongoing and Future Work

A major practical motivation for this work is the genome assembly problem. There, a graph is built based on the output of a sequencing machine, and long paths in this graph correspond to large sub-sequences of the genome. However, there the graphs are usually directed. Therefore, our next step will be the extension of our algorithm to directed graphs and the integration of it into existing genome assembly software.

In a separate line of research, we will try to give a streaming algorithm for the LPP (undirected or directed) with a proven worst-case guarantee on the length of the path and number of passes. At this time, it is unclear if any of the established theoretical techniques, such as color coding and Björklund-Husfeldt-type decompositions, are feasible in the streaming model.

Outline. We briefly describe three known RAM algorithms in Section 2. In Section 3, all the details of our algorithm are explained. In Section 4 we analyze its theoretical properties. The experimental studies takes place in Section 5 to Section 8.

2 Previous Algorithms

Trees. An algorithm for longest paths in trees was presented by Dijkstra around 1960; a proof of correctness can be found in [10]. It consists of two invocations of DFS, the first starting at an arbitrarily chosen vertex (e. g., chosen uniformly at random), and the second starting at a vertex that is in the final layer constructed by the first DFS.

Warnsdorf and Pohl-Warnsdorf. Warnsdorf’s rule was originally presented in 1823 and is a DFS that always picks a neighbor with a minimum number of unvisited neighbors. In case there are multiple such neighbors to choose from, Pohl gave a refinement [24, 25]: we restrict to those neighbors which themselves have a minimum-degree neighbor. Each vertex is used once as the starting point of the DFS, and the best path found is returned. This gives a total runtime of $\mathcal{O}(nm)$.

Pongrácz. This algorithm was announced in 2012 [26] and to the best of our knowledge has not been thoroughly studied since. We give a technically slightly modified description here. Given a start vertex r, using BFS we compute for each vertex v its distance to r. Then, starting at a randomly chosen v, we conduct a DFS that always picks an unvisited neighbor with maximum distance to r. Each vertex is used once as the start r, and the longest path found is returned. In the original version, for each r, also *each* v is tried (and not just one chosen randomly). In order to stay within $\mathcal{O}(nm)$, we decided to enumerate only one of the two possibilities: either r or v. In preliminary experiments, we found the choice given here (enumerate all r, pick one v randomly) to be superior. We leave a thorough study of the different variants of Pongrácz's algorithm for future work.

3 Description of Our Streaming Algorithm

Our algorithm works in two phases: (1) spanning tree construction, (2) spanning tree diameter improvement. Phase (1) is characterized by a parameter $\tau \in \mathbb{N}$ and a sequence $D = (D_1, \dots, D_{q_1})$ of degree limits, where $q_1 \geq 2$ and $D_{q_1} = \infty$. For each $i \in [\tau] = \{1, \dots, \tau\}$, a tree T_i is constructed. We start with the empty graph $T_i = (V, \emptyset)$ and then add edges to T_i over a number of q_1 passes. In each pass $p \in [q_1]$, we add an edge to T_i iff that does not create any cycle and it does not increase the maximum degree in T_i beyond D_p. Since $D_{q_1} = \infty$, we arrive at a spanning tree eventually (recall that we assume all our input graphs to be connected). The motivation for the degree limit is to favor path-like structures over clusters of edges. As an extreme example, consider a complete graph. Without degree restriction, it is possible that a spanning tree is constructed that is a star; whereas with a degree restriction of 2, we find a Hamilton path during the first pass.

In order to not just create the same tree τ times, in the first pass, we pick a number $r \in [m]$ uniformly at random (where $e_1, \dots, e_m$ is the stream of edges) and ignore any edges with an index smaller than r. Due to this offset for the first pass, it makes sense (but is not necessary) to use the same degree limit for the second pass. We will test $D = (2, \infty)$ and $D = (2, 2, 3, \infty)$ in experiments. By standard techniques (keeping track of the connected components), this algorithm can be implemented with a per-edge processing time of $\mathcal{O}(n)$: we can decide in $\mathcal{O}(1)$ if the current edge is to be inserted and if so, it takes $\mathcal{O}(n)$ to update connectivity information.

When all trees $T_1, \dots, T_\tau$ have been constructed, we unite them into a graph $U := (V, \bigcup_{i=1}^{\tau} E(T_i))$. This graph will in general contain cycles, but it has no more than τn edges. Since we construct U from trees, it is guaranteed to be connected and to span all the vertices of the input graph. In U, a long path P is constructed with a RAM algorithm; we use the Warnsdorf algorithm for this task. The final step of the first phase is to isolate P and then to build a spanning tree T around it using the same technique as for the trees $T_1, \dots, T_\tau$. Since we may assume that the constructions of $T_1, \dots, T_\tau$ are fed from the same passes, we thus have $2q_1$ passes for the first phase. We summarize phase (1) in Algorithm 1, which uses procedure `SpanningTree`, also given below. For a set X, we write $x :=_{\text{unif}} X$ to express that x is drawn uniformly at random from X.

When phase (1) is concluded, we determine a longest path P in the spanning tree T using the Dijkstra algorithm (Section 2). In phase (2), we try to modify this tree in order

```
Algorithm 1: Streaming Phase (1): Spanning Tree Construction
  Input: connected graph $G = (V, E)$ as a stream of edges, parameter $\tau$,
         degree limit sequence $D = (D_1, \dots, D_{q_1})$
  Output: spanning tree of $G$
1 foreach $i = 1, \dots, \tau$ do
2     $T_i \coloneqq (V, \emptyset)$;
3     SpanningTree $(T_i)$;
4 $U \coloneqq (V, \bigcup_{i=1}^{\tau} E(T_i))$;
5 find a long path $P$ in $U$ using Warnsdorf's algorithm;
6 $T \coloneqq (V, E(P))$;
7 SpanningTree $(T)$;
8 return $T$;
```

```
Procedure SpanningTree(T)
  Input: forest $T$ on $V$, possibly empty
  Output: spanning tree on $V$
1 $r \coloneqq_{\text{unif}} [m]$;
2 fast-forward the stream to position $r$;
3 for $p = 1, \dots, q_1$ do
4     while not at the end of the stream do
5         get next edge $vw$ from the stream;
6         if $T + vw$ is cycle-free and $\max\{\deg_T(v), \deg_T(w)\} < D_p$ then $T \coloneqq T + vw$;
7         if $|T| = n - 1$ then break;
8     rewind the stream to its beginning;
```

that it admits longer paths than P. A number of additional passes is conducted. In order to save time, we developed a criterion based on which we only consider a fraction of the edges during those passes. We explored the two options: (i) consider each edge independently with probability $\frac{n}{m+1}$ (resulting in only $\mathcal{O}(n)$ edges being considered); or (ii) skip an edge if both endpoints are on the so-far longest path P. After preliminary experiments, we decided for option (ii) due to better solution quality at a moderate runtime expense. A detailed comparison of (i) and (ii) is planned for the full version of this work; in our tables in Section 8, however we already give results for one variant of our algorithm using option (i).

For each edge e that is considered and that is not in T, we temporarily add e to T, creating a fundamental cycle C in $T' \coloneqq T + e$. We want to go back to a tree. To this end, we have to remove an edge from C. This edge is chosen so that among all possibilities, the resulting tree has maximum diameter.

It should not be assumed that an edge with both endpoints on P could not yield an improvement. Intuitively, relative to P it acts like a shortcut, but examples can be found where adding such an edge (and subsequently removing one edge from the fundamental cycle) improves the diameter of the tree. Still, criterion (ii) has shown to be effective in practice.

Phase (2) terminates after a preset number of passes q_2. We summarize phase (2) in Algorithm 2, where for any graph H, we denote $\ell(H)$ the length of a longest path in H.

Algorithm 2: Streaming Phase (2): Improvement

```
   Input: connected graph $G$ as a stream of edges, spanning tree $T$, pass limit $q_2$
   Output: a (long) path in $G$
1  compute longest path $P$ in $T$ with Dijkstra algorithm;
2  for $q_2$ times do
3      rewind the stream to its beginning;
4      while not at the end of the stream do
5          get next edge $e = vw$ from stream;
6          if $v \in V(P)$ and $w \in V(P)$ then discard and continue with next iteration;
7          $T' := T + e$;
8          compute fundamental cycle $C$ in $T'$;
9          $\ell^* := \max_{f \in E(C) \setminus \{e\}} \ell(T' - f)$;
10         if $\ell^* > |P|$ then
11             pick any $e'$ from the set $\{f \in E(C) \setminus \{e\}\,;\ \ell(T' - f) = \ell^*\}$;
12             $T := T' - e'$;
13             update $P$ with longest path in $T$;
14 return $P$;
```

4 Properties of Our Streaming Algorithm

If the cycle C is of length $\Omega(n)$, then a naive implementation requires $\Omega(n^2)$ to find an edge e' to remove (temporarily remove each edge on the cycle and invoke the Dijkstra algorithm). However, we have:

▶ **Theorem 1.** *Phase (2) can be implemented with per-edge processing time $\mathcal{O}(n)$.*

Proof. An $\mathcal{O}(n)$ bound is clear for all lines of Algorithm 2, except line 9 and line 11. Denote

$$\ell' := \max_{f \in E(C) \setminus \{e\}} \max \{|P|\,;\ P \text{ is path in } T' - f \text{ and } e \in E(P)\}$$

and let $R' \subseteq E(C) \setminus \{e\}$ be the set of edges where this maximum is attained. Then the following implications hold: $\ell' \leq |P| \implies \ell^* \leq |P|$ and $\ell' > |P| \implies \ell' = \ell^*$. This is because if a longest path in $T' - f$ is supposed to be longer than P, it must use e (since otherwise it would be a path in T). Hence it suffices to determine ℓ', and if $\ell' > |P|$, to find an element of R'.

Denote $C = (v_1, \ldots, v_k)$ the fundamental cycle for some $k \in \mathbb{N}$ written so that $e = v_1 v_k$. When computing ℓ', we can restrict to paths in T' of the form

$$(\ldots, v_s, v_{s-1}, \ldots, v_1, v_k, v_{k-1}, \ldots, v_t, \ldots) \tag{1}$$

for $1 \leq s < t \leq k$, where v_s is the first and v_t is the last common vertex, respectively, of the path and C. For each i, let T_i be the connected component of v_i in $T - E(C)$, i. e., T_i is the part of T that is reachable from v_i without using the edges of C. Denote $\ell(T_i)$ the length of a longest path in T_i that starts at v_i and denote $c_i := \ell(T_i) + i - 1$ and $a_i := \ell(T_i) + k - i$. Then a longest path entering C at v_s and leaving it at v_t, as in (1), has length exactly $c_s + a_t$. Hence we have to determine a pair (s, t) such that $c_s + a_t$ is maximum (this maximum value is ℓ'); we call such a pair an *optimal pair*. If the so determined value ℓ' is not greater than $|P|$, then nothing further has to be done (the edge e cannot give an improvement). Otherwise, having constructed our optimal pair (s, t), we pick an arbitrary edge (e.g., uniformly at random) from $\{v_i v_{i+1}\ ;\ s \leq i < t\}$, which are the edges between v_s and v_t on C. We show that the following algorithm computes the value ℓ' and an optimal pair in $\mathcal{O}(n)$.

```
1  compute $c_1, \dots, c_{k-1}$ and $a_2, \dots, a_k$ using DFS;
2  $M := 0$; $L := 0$;
3  for $i = 1, \dots, k-1$ do
4      if $c_i > M$ then
5          $M := c_i$;
6          $s := i$;
7      if $M + a_{i+1} > L$ then
8          $L := M + a_{i+1}$;
9          $t := i + 1$;
10 return $(s, t)$;
```

The total of computations in line 1 can be done by DFS in $\mathcal{O}(n)$, and the loop in $\mathcal{O}(k) \leq \mathcal{O}(n)$. We prove that the final (s, t) is optimal. For fixed t, the best possible length $c_s + c_t$ is obtained if t is combined with an $s < t$ where $c_s \geq c_j$ for all $j < t$. In the algorithm, for each t (when $t = i + 1$ in the loop) we combine a_t with the maximum $\max_{j<t} c_j$ (stored in the variable M). Thus, when the algorithm terminates, $L = \ell'$ and $c_s + c_t = \ell'$. ◀

▶ **Corollary 2.** *Our streaming algorithm (with the two phases as in Algorithm 1 and Algorithm 2) can be implemented with a per-edge processing time of $\mathcal{O}(n)$.*

We turn to the memory requirement. Denote b the amount of RAM required to store one vertex or one pointer (e.g., $b = 32$ bit or $b = 64$ bit) and call $n \cdot b$ one *unit*.

▶ **Theorem 3.** *Our streaming algorithm (with the two phases as in Algorithm 1 and Algorithm 2) conducts at most $2q_1 + q_2$ passes. Moreover, the algorithm can be implemented such that the RAM requirement is at most $(\max\{4\tau, 2\tau + 4\} \cdot n + c) \cdot b$ with a constant c.*

Proof. The construction of each of the initial trees $T_1, \dots, T_\tau$ can be fed from the same passes, so we obtain those τ trees within at most q_1 passes. After isolating the path P, we need at most q_1 more passes to get back to a spanning tree. A bound of q_2 for phase (2) is obvious. We turn to the memory requirement.

Phase (1). All the adjacency lists of one tree together require 2 units, plus 1 unit for an array of pointers to each of the lists. We need 1 additional unit per tree to store connectivity information during tree construction. This amounts to 4τ units for the main data structures at any time so far, plus a few extra bits required for bookkeeping (loop variables, etc) that are covered by the constant c. The graph U can be stored in $2\tau + 1$ units (adjacency lists plus pointer array). For the Warnsdorf algorithm, we need 1 unit to store DFS information (e.g., store the DFS tree as a predecessor relation) and 1 unit for the best path so far. To determine the next vertex to visit (according to Warnsdorf's rule), we need $\Delta b \leq nb$ bits, where Δ is the maximum degree in the graph. This amounts to $2\tau + 4$ units for the main data structures for the Warnsdorf algorithm on U. The rest of phase (1) is clearly covered by this as well.

Phase (2). We need 3 units to store the tree, 1 unit to store the longest path so far, 1 unit for the fundamental cycle, and 1 unit for DFS. This amounts to 6 units during this phase plus bookkeeping, which is covered by the stated bound. ◀

▶ Remark. On a graph with average degree d, the Warnsdorf, Pohl-Warnsdorf, and Pongrácz algorithms each requires at least $(d+3) \cdot nb$ bits of memory.

Proof. Since those algorithms perform special variants of DFS (and Pongrácz also BFS), we cannot restrict them to sequential access and thus we have to load the instance into RAM as adjacency lists.[2] Hence, $d+1$ units are required to store the graph. Two more units must be allotted to store DFS information and the longest path found so far, in the case of Pongrácz need one more unit for the distance information. ◀

▶ **Corollary 4.** *Not counting the additive constant c from Theorem 3, the RAM algorithms require $\frac{d+3}{\max\{4\tau, 2\tau+4\}}$ times more RAM than our streaming algorithm, on a graph with average degree d. For $\tau = 2$, this ratio is $\frac{d+3}{8}$.*

5 Test Instances

Connected Random. We denote this model by $\mathcal{G}^*(n,p)$. A graph is constructed by starting with a random tree on n vertices (via a randomly chosen Prüfer sequence) and then adding further edges as in $\mathcal{G}(n,p)$. The average degree in such a graph is slightly larger than np due to the $n-1$ initial tree edges.

Chains. Parameters for a chain graph are n, p, and k, with n being a multiple of k. We create k graphs $G_1, \ldots, G_k$, the *clusters*, according to $\mathcal{G}^*(n,p)$, each on n/k vertices. Then we insert an edge $v_i w_i$ with randomly chosen $v_i \in V(G_i)$ and $w_i \in V(G_{i+1})$ for each $1 \leq i < k$, making sure that $w_i \neq v_{i+1}$. Such graphs pose a particular challenge to DFS-based LPP algorithms, since if the DFS visits the connecting point to the next cluster (w_i or v_i) too early, it will eventually miss out on a large number of vertices in the current cluster.

Preferential Attachment and Small World. Preferential attachment graphs are created as per the Barabási-Albert model [3]: parameters are $n, n_0, d \in \mathbb{N}$, where n is total the number of vertices, n_0 is the size of the initial tree, and in each step the new vertex is connected by d new edges. This model guarantees connectedness. Small world graphs are created as per the Watts-Strogatz model [31], with a small modification. Parameters are $n, d \in \mathbb{N}$, with d even, and $0 \leq \beta \leq 1$. We start with a ring lattice where each vertex is connected to each $d/2$ vertices on either side, then each edge vw with v and w not being next to each other on the ring is replaced with a random edge vu with probability β (the *rewiring probability*). Our modification (not to rewire certain edges) guarantees that the result is Hamiltonian (and in particular connected).

These two models were chosen since they yield very different degree distributions: for preferential attachment, we have a power-law and there exist a few *hubs*, i. e., vertices with high degree. In the small world model on the other hand, vertices tend to have similar degree.

[2] This is unless we invoke external-memory techniques, which is unexplored for the LPP at this time.

Hyperbolic Geometric. Hyperbolic geometric graphs are a very interesting new class of graphs, for which efficient generators were recently given by von Looz, Staudt, Meyerhenke, and Prutkin [30]. They are constructed in hyperbolic space of constant negative curvature. Vertices correspond to points that are randomly inserted into this space, and an edge between two vertices is inserted if the corresponding points are within a certain distance from each other. This model has been shown to exhibit many features of complex real-world networks. We refer to [20, 30] for details. Parameters are number of vertices n, average degree d, and the exponent γ of the power-law degree distribution. We use the generator implementation from [30]. Connectedness is ensured by initializing the graph with a random tree.

6 Experimental Setup

Each algorithm was implemented in C++14. Each graph stream is realized as a `std::vector` of pairs of 32 bit integers. We keep those vectors in RAM for the sake of faster running times and hence more experiments conducted – but it is guaranteed that we access those vectors only sequentially and all other data structures are $\mathcal{O}(n)$. Our implementation also allows to process graphs stored in a file on disk, without copying the contents of the file into RAM (it is accessed via a `std::ifstream`). Using the *Valgrind* tool *Massif*,[3] we verified that RAM consumption of our algorithm is indeed independent of the number of edges. For each instance, the stream of edges is randomized once and the order does not change between passes or between the invocations of the algorithms. Each implementation concludes immediately when a Hamilton path is found.

For each random graph model under consideration, we test three settings: $n = 16{,}000$ and nominal average degree $d = 14$ (*sparse*); $n = 16{,}000$ and nominal average degree $d = \sqrt[3]{n}$ (*dense*); and $n = 100{,}000$ and nominal average degree $d = 10$ (*large*). (Note that chain and hyperbolic graphs will have a slightly larger average degree than the given d due to the additional tree that is used to guarantee connectedness.) The *dense* graphs have $\Omega(n^{4/3})$ edges and are thus beyond the theoretical RAM capacity of the semi-streaming model. More on the practical side, note that by Corollary 4 (not counting the small additive constant), even for average degree $d = 14$, the RAM algorithms require more than two times more memory than ours when configured with $\tau \leq 2$. Due to lack of space we skip the details for *sparse* and *dense* small world graphs, and we only use a selection of algorithms for the *large* graphs. The study of larger and more instances is deferred to the full version, due to time constraints.

We run Warnsdorf, Pohl-Warnsdorf, Pongrácz, the simple randomized DFS, and different variants of our algorithm on 100 randomly generated instances for each parameter set (only 50 instances for *large* graphs in order to save time) and record the length of the path that is found and the running time. Variants of our algorithm are denoted in the form $\tau/q_1/q_2$, where τ is the number of trees in the beginning, q_1 is the maximum number of passes used to construct a spanning tree using degree limiting, and q_2 is the number of improvement passes. In order to save time, for fixed τ and q_1, we obtain results for $\tau/q_1/0$ up to $\tau/q_1/q_2$ by running $\tau/q_1/q_2$ and recording intermediate results.

Solution quality is analyzed in terms of *relative solution quality.* For an instance I and algorithm A, denote $\ell(A, I)$ the length of the path delivered by A on I. Then we define $\rho(A, I) := \frac{\ell(A,I)}{\max_{A'} \ell(A',I)} \in [0, 1]$, where A' runs over all algorithms under investigation. That

[3] `http://valgrind.org/docs/manual/ms-manual.html`

is the result of A divided by the best result on any of the algorithms. Clearly, one algorithm per instance will always have relative solution quality 100%.

7 Data and Discussion

Tables with detailed experimental data can be found in section 8. The column labeled 'ℓ' gives statistics (mean value μ and standard deviation σ) for the lengths of the paths found and is intended as a general orientation in which range our solutions are located. The column labeled 'wins' counts how many times this algorithm delivered the best solution, i. e., how many times it achieved relative solution quality $\rho = 100\%$. Detailed statistics are given for the relative performance in the following columns: mean value, standard deviation, minimum, 5th and 10th percentile, and median. We use percentile notation everywhere: P_0 for the minimum, P_5 and P_{10} for the 5th and 10th percentile, and P_{50} for the median. In the final two columns, we give the running time in seconds. The algorithm marked with a star (2/4/3*) uses the randomized criterion for skipping edges in the improvement phase, whereas all other variants of our algorithm use the path criterion as stated in Algorithm 2. In the following, we distill the data from the tables into several observations and conclusions.

- The fact that the simple randomized DFS algorithm (denoted 'DFS' in the tables) delivers clearly inferior results in many cases is an indication that at least those instances are not 'too easy'.
- Warnsdorf and Pohl-Warnsdorf are generally the best, except for chain graphs. For many of the instances, they find a Hamilton path, and then they are very fast, sometimes below one second. Note that this advantage could easily be removed by making the graphs non-Hamiltonian, e. g., by connecting two additional vertices as leafs to the same vertex.
- Warnsdorf and Pohl-Warnsdorf are close to each other in terms of solution quality, but unsurprisingly the former is faster.
- In terms of the average path length $\mu(\ell)$, for each set of parameters there is one algorithm that delivers at least $0.84 \cdot n$, i. e., 84% of a Hamilton path. It follows that a good relative performance also means a good absolute performance.
- Our strongest variant, 2/4/3, with the exception of preferential attachment graphs, always delivers a relative solution quality of at least 71%. For preferential attachment, we record a minimum of 49% in Table 3. In terms of the 5th percentile, i. e., after removing the 5% worst cases, and omitting preferential attachment graphs, our minimum relative solution quality is 83%. In terms of the 10th percentile and including preferential attachment graphs, we still have at least 71%. In terms of mean and median, we have at least 83%.
- Regarding running time, we compare our variant 2/4/3 with Warnsdorf, which is the fastest RAM algorithm, not counting the simple randomized DFS. Clearly, we cannot compete in cases where Warnsdorf finds a Hamilton path within a second, but as remarked before, this advantage of Warnsdorf could easily be removed by making the graph non-Hamiltonian. Apart from those cases, in the *sparse* and *dense* sets, the biggest difference is for *sparse* hyperbolic graphs, where Warnsdorf only needs about 56% of our running time on average. For *dense* chains, we are faster than Warnsdorf. For the *large* set, our variant 2/4/1 has similar running times as Warnsdorf, while delivering at least 71% in terms of P_{10}, and when excluding preferential attachment graphs it delivers 82% in terms of P_5. More than one improvement pass here only gives incremental gain, so in order to save time on large graphs, the variant 2/4/1 is recommended over 2/4/2 or 2/4/3.
- Using $\tau = 2$ has a clear advantage over $\tau = 1$, in particular compare 1/2/0 with 2/2/0 in terms of ℓ in Table 1 and Table 2.

- The degree-limiting technique yields substantial improvements. For $q_1 = 2$ (i. e., for variants of the form $\tau/2/q_2$), we use the sequence $D = (2, \infty)$, i. e., in the first pass we limit the degree to 2 and in the second pass we have no limit. In the configuration with $q_1 = 4$ we use $D = (2, 2, 3, \infty)$. Comparing for example 1/2/0 with 1/4/0 with respect to ℓ in Table 2 for preferential attachment and hyperbolic graphs, we see that 1/2/0 delivers roughly $50 - 60\%$ length on average compared to 1/4/0. Comparing 2/2/3 with 2/4/3 in particular with respect to P_0, P_5, and P_{10} for preferential attachment graphs in Table 1, we see that $q_1 = 4$ brings an improvement even on top of the improvement gained by using $\tau = 2$ and by the improvement phase.
- The improvement phase (phase (2)) can bring further improvements, in particular with respect to P_0. This is seen for example by comparing 2/4/0 with 2/4/3 for preferential attachment and hyperbolic graphs in Table 1.
- Comparing the runtimes of 2/2/3 and 2/4/3 over all tables, we find that consistently the former is slower, while delivering inferior solutions. The same goes for 1/4/3 and 2/4/3; here the difference in running time is very high for preferential attachment graphs. This shows that a lack of effort in phase (1) can make phase (2) substantially slower. An explanation is that more improvement steps have to be carried out.
- Comparing 2/4/3 with 2/4/3*, we find the former being consistently better in terms of solution quality, but requiring up to roughly 30% more time.
- Our biggest advantage (using 2/4/3) over the other algorithms is for chain graphs.

In particular, we conclude from those observations that none of the three features (namely using multiple trees in the beginning, degree-limiting, and improvement) should be missed. The combination of all those features makes our algorithm competitive.

8 Tables of Experimental Data

On the following pages please find tables of results for the experiments as discussed in Section 7. By μ we denote the mean value and by σ the standard deviation. By P_i we denote the ith percentile, in particular P_0 is the minimum and P_{50} is the median. By ℓ we denote the path length and by ρ the relative performance. Running times t (last two columns) are in seconds. For further explanations, please see Section 7.

Acknowledgments. We thank the German Research Foundation (DFG) for financial support through Grant SR7/15-1 *Algorithmic Foundations for Genome Assembly* within Priority Programme 1736 *Algorithms for Big Data*. We thank Ole Kliemann for substantially improving our C++ implementation and helpful discussions.

Table 1 *Sparse Set:* $n = 16{,}000$ and $d = 14$.

		ℓ			ρ in %						t	
graph class	algo	μ	σ	wins	μ	σ	P_0	P_5	P_{10}	P_{50}	μ	σ
chain $k = 125$ $l = 128$ $p = 0.11$ $n = 16{,}000$ $\lvert E\rvert \approx 127{,}044$	1/2/0	3,032	645	0	20	4	11	12	13	21	26	2
	1/2/3	13,435	606	0	89	4	79	80	82	91	185	34
	1/4/0	3,947	222	0	26	1	22	24	24	26	23	0
	1/4/3	14,150	55	0	94	1	93	93	93	94	141	4
	2/2/0	10,985	1,336	0	73	9	17	58	64	75	46	4
	2/2/3	14,522	373	7	96	2	91	92	93	97	129	18
	2/4/0	11,683	617	0	77	4	63	68	73	78	48	4
	2/4/3	15,062	122	93	100	0	98	100	100	100	103	3
	2/4/3*	14,346	283	0	95	2	86	90	94	96	92	4
	Pon	14,518	52	0	96	1	95	95	95	96	110	4
	War	10,598	304	0	70	2	66	67	68	70	86	2
	PW	10,539	306	0	70	2	65	67	68	70	131	3
	DFS	9,255	165	0	61	1	59	60	60	61	38	1
pref. attach. $n_0 = 7$ $d = 14$ $n = 16{,}000$ $\lvert E\rvert = 111{,}957$	1/2/0	464	137	0	3	1	1	2	2	3	35	6
	1/2/3	6,653	1,212	0	42	8	27	27	28	43	437	23
	1/4/0	748	105	0	5	1	3	4	4	5	29	0
	1/4/3	8,281	122	0	52	1	50	50	51	52	394	8
	2/2/0	12,712	2,350	0	79	15	15	43	58	85	47	3
	2/2/3	13,000	1,859	0	81	12	36	53	65	86	169	71
	2/4/0	13,743	1,825	0	86	11	18	66	76	90	46	4
	2/4/3	14,060	1,100	0	88	7	55	73	79	91	133	44
	2/4/3*	13,817	1,265	0	86	8	51	69	75	90	104	3
	Pon	13,565	28	0	85	0	84	84	85	85	113	4
	War	16,000	0	100	100	0	100	100	100	100	0	0
	PW	16,000	0	100	100	0	100	100	100	100	0	0
	DFS	12,385	27	0	77	0	77	77	77	77	41	1
hyperbolic $d = 14$ $\gamma = 3$ $n = 16{,}000$ $\lvert E\rvert \approx 128{,}185$	1/2/0	586	185	0	4	1	1	2	2	4	33	22
	1/2/3	9,791	826	0	61	5	49	50	54	62	337	38
	1/4/0	968	144	0	6	1	4	5	5	6	25	0
	1/4/3	10,987	145	0	69	1	67	67	67	69	290	8
	2/2/0	12,822	1,808	0	80	11	37	47	62	84	46	4
	2/2/3	14,099	1,013	0	88	6	67	71	77	90	130	41
	2/4/0	13,732	941	0	86	6	56	72	78	88	46	5
	2/4/3	14,646	509	0	92	3	77	84	86	93	108	19
	2/4/3*	14,303	587	0	89	4	69	82	85	91	97	3
	Pon	14,373	106	0	90	1	87	89	89	90	117	5
	War	15,997	5	92	100	0	100	100	100	100	61	38
	PW	15,998	4	94	100	0	100	100	100	100	83	52
	DFS	12,908	22	0	81	0	80	80	81	81	40	1

Table 2 *Dense Set:* $n = 16{,}000$ and $d = \sqrt[3]{n}$.

		ℓ			ρ in %						t	
graph class	algo	μ	σ	wins	μ	σ	P_0	P_5	P_{10}	P_{50}	μ	σ
chain $k = 125$ $l = 128$ $p = 0.21$ $n = 16{,}000$ $\lvert E\rvert \approx 222{,}351$	1/2/0	3,749	860	0	24	6	11	13	15	26	25	2
	1/2/3	14,633	438	0	94	3	86	87	89	95	200	52
	1/4/0	4,666	314	0	30	2	24	27	27	30	23	1
	1/4/3	15,079	34	0	97	0	97	97	97	97	144	4
	2/2/0	11,646	755	0	75	5	57	62	70	77	49	5
	2/2/3	15,248	251	12	98	2	92	95	96	99	139	26
	2/4/0	11,864	867	0	77	5	40	67	69	78	50	6
	2/4/3	15,489	67	88	100	0	99	100	100	100	112	5
	2/4/3*	14,715	135	0	95	1	91	93	94	95	99	5
	Pon	15,034	39	0	97	0	96	96	97	97	165	10
	War	10,420	313	0	67	2	63	64	65	67	126	5
	PW	10,380	315	0	67	2	62	64	65	67	208	6
	DFS	9,348	142	0	60	1	58	59	59	60	52	3
pref. attach. $n_0 = 13$ $d = 26$ $n = 16{,}000$ $\lvert E\rvert = 207{,}843$	1/2/0	639	176	0	4	1	2	2	2	4	32	5
	1/2/3	8,783	1,238	0	55	8	34	39	43	56	721	67
	1/4/0	1,037	151	0	6	1	4	5	5	6	27	1
	1/4/3	10,576	113	0	66	1	64	65	65	66	604	16
	2/2/0	14,248	1,587	0	89	10	26	72	83	92	50	3
	2/2/3	14,534	969	0	91	6	59	80	85	93	182	80
	2/4/0	14,878	720	0	93	5	65	84	91	94	51	4
	2/4/3	15,102	422	0	94	3	80	88	93	95	139	33
	2/4/3*	15,004	450	0	94	3	73	88	91	95	111	4
	Pon	14,754	18	0	92	0	92	92	92	92	165	13
	War	16,000	0	100	100	0	100	100	100	100	0	0
	PW	16,000	0	100	100	0	100	100	100	100	0	0
	DFS	13,923	14	0	87	0	87	87	87	87	55	4
hyperbolic $d = 26$ $\gamma = 3$ $n = 16{,}000$ $\lvert E\rvert \approx 224{,}369$	1/2/0	737	242	0	5	2	1	2	3	5	29	5
	1/2/3	11,818	852	0	74	5	60	62	67	75	458	59
	1/4/0	1,304	188	0	8	1	6	6	7	8	25	1
	1/4/3	12,885	122	0	81	1	78	79	80	81	369	13
	2/2/0	13,867	1,090	0	87	7	48	69	80	89	49	4
	2/2/3	14,998	480	0	94	3	81	87	90	95	139	38
	2/4/0	14,299	554	0	89	3	64	84	89	90	50	4
	2/4/3	15,237	210	0	95	1	88	93	95	96	119	16
	2/4/3*	14,727	423	0	92	3	77	87	89	93	101	4
	Pon	14,971	77	0	94	0	91	93	93	94	169	10
	War	16,000	0	100	100	0	100	100	100	100	0	0
	PW	16,000	0	100	100	0	100	100	100	100	0	0
	DFS	13,769	19	0	86	0	86	86	86	86	55	3

Table 3 *Large Set:* $n = 100{,}000$ and $d = 10$.

		ℓ			ρ in %						t	
graph class	algo	μ	σ	wins	μ	σ	P_0	P_5	P_{10}	P_{50}	μ	σ
chain $k = 1{,}000$ $l = 100$ $p = 0.01$ $n = 100{,}000$ $\lvert E\rvert \approx 599{,}468$	2/4/0	69,738	8,809	0	82	8	35	69	72	84	2,345	387
	2/4/1	84,335	3,661	0	99	0	98	99	99	99	3,997	395
	2/4/2	84,884	3,511	0	100	0	100	100	100	100	5,334	603
	2/4/3	84,909	3,500	50	100	0	100	100	100	100	6,591	833
	War	68,212	2,071	0	80	4	76	77	77	79	3,587	192
pref. attach. $n_0 = 5$ $d = 10$ $n = 100{,}000$ $\lvert E\rvert = 499{,}979$	2/4/0	80,190	8,064	0	82	8	49	58	71	86	2,612	778
	2/4/1	80,380	7,681	0	82	8	51	59	71	86	4,385	1,308
	2/4/2	80,510	7,459	0	82	8	53	60	71	86	5,974	1,957
	2/4/3	80,606	7,322	0	83	7	54	60	71	86	7,465	2,660
	War	97,685	52	50	100	0	100	100	100	100	4,110	588
hyperbolic $d = 10$ $\gamma = 3$ $n = 100{,}000$ $\lvert E\rvert \approx 599{,}680$	2/4/0	84,202	4,582	0	85	5	60	76	82	87	2,641	358
	2/4/1	88,147	3,772	0	89	4	68	82	86	91	3,978	591
	2/4/2	88,426	3,519	0	90	4	70	82	87	91	5,015	858
	2/4/3	88,494	3,400	0	90	3	71	83	87	91	5,963	1,076
	War	98,710	84	50	100	0	100	100	100	100	3,910	294
small world $d = 10$ $\beta = 0.3$ $n = 100{,}000$ $\lvert E\rvert = 500{,}000$	2/4/0	86,928	4,924	0	89	5	68	80	83	91	2,441	401
	2/4/1	90,810	4,038	0	93	4	76	86	89	95	3,457	531
	2/4/2	91,171	3,758	0	94	4	78	86	89	95	4,275	837
	2/4/3	91,253	3,590	0	94	4	79	87	89	95	5,072	1,148
	War	97,212	44	50	100	0	100	100	100	100	3,357	223

References

1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995. doi:10.1145/210332.210337.

2 Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, USA, January 2002 (SODA 2002)*, pages 623–632, 2002. URL: http://dl.acm.org/citation.cfm?id=545464.

3 Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. doi:10.1126/science.286.5439.509.

4 Cristina Bazgan, Miklos Santha, and Zsolt Tuza. On the approximation of finding a(nother) Hamiltonian cycle in cubic Hamiltonian graphs. *Journal of Algorithms*, 31(1):249–268, 1999. Conference version at STACS 1998. doi:10.1006/jagm.1998.0998.

5 Andreas Björklund. Determinant sums for undirected Hamiltonicity. *SIAM Journal on Computing*, 43(1):280–299, 2014. Conference version at FOCS 2010. doi:10.1137/110839229.

6 Andreas Björklund and Thore Husfeldt. Finding a path of superlogarithmic length. *SIAM Journal on Computing*, 32(6):1395–1402, 2003. doi:10.1137/S0097539702416761.

7 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Narrow sieves for parameterized paths and packings, 2010. URL: http://arxiv.org/abs/1007.1161.

8 Andreas Björklund, Thore Husfeldt, and Sanjeev Khanna. Approximating longest directed paths and cycles. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming, Turku, Finland, July 2004 (ICALP 2004)*, pages 222–233, 2004. doi:10.1007/978-3-540-27836-8_21.

9 Hans L. Bodlaender. On linear time minor tests with depth-first search. *Journal of Algorithms*, 14:1–23, 1993. Conference version at WADS 1989. doi:10.1006/jagm.1993.1001.

10 R.W. Bulterman, F.W. van der Sommen, G. Zwaan, T. Verhoeff, A.J.M. van Gasteren, and W.H.J. Feijen. On computing a longest path in a tree. *Information Processing Letters*, 81(2):93–96, 2002. doi:10.1016/S0020-0190(01)00198-3.

11 Tomás Feder, Rajeev Motwani, and Carlos Subi. Approximating the longest cycle problem in sparse graphs. *SIAM Journal on Computing*, 31(5):1596–1607, 2002. doi:10.1137/S0097539701395486.

12 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharath Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348:207–216, 2005. Conference version at ICALP 2004. doi:10.1016/j.tcs.2005.09.013.

13 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharath Suri, and Jian Zhang. Graph distances in the data-stream model. *SIAM Journal on Computing*, 38:1709–1727, 2008. doi:10.1137/070683155.

14 Harold N. Gabow. Finding paths and cycles of superpolylogarithmic length. *SIAM Journal on Computing*, 36(6):1648–1671, 2007. doi:10.1137/S0097539704445366.

15 Harold N. Gabow and Shuxin Nie. Finding long paths, cycles and circuits. In *Proceedings of the 19th International Symposium on Algorithms and Computation, Gold Coast, Australia, December 2008 (ISAAC 2008)*, pages 752–753, 2008. doi:10.1007/978-3-540-92182-0_66.

16 Venkatesan Guruswami and Krzysztof Onak. Superlinear lower bounds for multipass graph processing. *Electronic Colloquium on Computational Complexity*, 2014. Conference version at CCC 2013. URL: http://eccc.hpi-web.de/report/2013/002/.

17 David Karger, Rajeev Motwani, and G.D.S. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18:82–98, 1997. doi:10.1007/BF02523689.

18 Fatemeh Keshavarz-Kohjerdia, Alireza Bagherib, and Asghar Asgharian-Sardroudb. A linear-time algorithm for the longest path problem in rectangular grid graphs. *Discrete Applied Mathematics*, 160(3):210–217, 2012. `doi:10.1016/j.dam.2011.08.010`.

19 Ioannis Koutis. Faster algebraic algorithms for path and packing problems. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Reykjavik, Iceland, July 2008 (ICALP 2008)*, pages 575–586, 2008. `doi:10.1007/978-3-540-70575-8_47`.

20 Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Physical Review E*, 82, 2010. `doi:10.1103/PhysRevE.82.036106`.

21 Minko Markov, Mugurel Ionuţ Andreica, Krassimir Manev, and Nicolae Ţăpuş. A linear time algorithm for computing longest paths in cactus graphs. *Serdica Journal of Computing*, 6(3), 2012. URL: `http://serdica-comp.math.bas.bg/index.php/serdicajcomputing/article/view/158`.

22 Burkhard Monien. How to find long paths efficiently. *Annals of Discrete Mathematics*, 25:239–254, 1985. URL: `https://digital.ub.uni-paderborn.de/hs/content/titleinfo/42079`.

23 Muthu Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):67 pages, 2005. URL: `http://algo.research.googlepages.com/eight.ps`.

24 Ira Pohl. A method for finding Hamilton paths and knight's tours. *Communications of the ACM*, 10(7):446–449, 1967. `doi:10.1145/363427.363463`.

25 Ira Pohl and Larry Stockmeyer. Pohl-Warnsdorf – revisited. In *Proceedings of the International Conference on Intelligent Systems and Control, Honolulu, Hawaii, USA, August 2004 (ISC 2004)*, 2004. URL: `https://users.soe.ucsc.edu/~pohl/Papers/Pohl_Stockmeyer_full.pdf`.

26 Lajos L. Pongrácz. A greedy approximation algorithm for the longest path problem in undirected graphs, 2012. URL: `http://arxiv.org/abs/1209.2503v2`.

27 Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 2011.

28 Ryuhei Uehara and Yushi Uno. On computing longest paths in small graph classes. *International Journal of Foundations of Computer Science*, 18(5), 2007. `doi:10.1142/S0129054107005054`.

29 Sundar Vishwanathan. An approximation algorithm for finding long paths in Hamiltonian graphs. *Journal of Algorithms*, 50(2):246–256, 2004. Conference version at SODA 2000. `doi:10.1016/S0196-6774(03)00093-2`.

30 Moritz von Looz, Christian L. Staudt, Henning Meyerhenke, and Roman Prutkin. Fast generation of complex networks with underlying hyperbolic geometry, 2015. URL: `http://arxiv.org/abs/1501.03545`.

31 Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998. `doi:10.1038/30918`.

32 Ryan Williams. Finding paths of length k in $O^*(2^k)$ time. *Information Processing Letters*, 109:315–318, 2009. `doi:10.1016/j.ipl.2008.11.004`.