Finding Hamiltonian Cycle in Graphs of Bounded Treewidth: Experimental Evaluation

Michał Ziobro
Theoretical Computer Science Department, Faculty of Mathematics and Computer Science, Jagiellonian University, Kraków, Poland
michal.18.ziobro@student.uj.edu.pl

Marcin Pilipczuk
Institute of Informatics, University of Warsaw, Poland
malcin@mimuw.edu.pl

Abstract

The notion of treewidth, introduced by Robertson and Seymour in their seminal Graph Minors series, turned out to have tremendous impact on graph algorithmics. Many hard computational problems on graphs turn out to be efficiently solvable in graphs of bounded treewidth: graphs that can be sweeped with separators of bounded size. These efficient algorithms usually follow the dynamic programming paradigm.

In the recent years, we have seen a rapid and quite unexpected development of involved techniques for solving various computational problems in graphs of bounded treewidth. One of the most surprising directions is the development of algorithms for connectivity problems that have only single-exponential dependency (i.e., $2^{\mathcal{O}(t)}$) on the treewidth in the running time bound, as opposed to slightly superexponential (i.e., $2^{\mathcal{O}(t \log t)}$) stemming from more naive approaches. In this work, we perform a thorough experimental evaluation of these approaches in the context of one of the most classic connectivity problem, namely Hamiltonian Cycle.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms, Theory of computation → Graph algorithms analysis, Theory of computation → Dynamic programming

Keywords and phrases Empirical Evaluation of Algorithms, Treewidth, Hamiltonian Cycle

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.29

Funding Supported by the "Recent trends in kernelization: theory and experimental evaluation" project, carried out within the Homing programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

1 Introduction

The problem of finding Hamiltonian Cycle in graph is one of the oldest and best known $\mathcal{NP}$-complete problems. It was intensly studied together with its more generic optimization version Traveling Salesman Problem. Early and important result on this problem was dynamic algorithm invented independently by Bellman [2] and Held and Karp [16], running in time $O(2^n n^2)$. The exponential factor of this running time bound remains the best known for deterministic algorithms up to today, and a faster randomized Monte Carlo algorithm has been shown only very recently by Björklund [3]. Faster algorithms were also obtained for some special cases, like graphs with bounded degree [9, 4] or claw-free graphs [7].

An important class of graphs in which many combinatorial problems can be solved more efficiently, are graphs of bounded *treewidth*. Treewidth, introduced by Robertson and

© Michał Ziobro and Marcin Pilipczuk;
licensed under Creative Commons License CC-BY
17th International Symposium on Experimental Algorithms (SEA 2018).
Editor: Gianlorenzo D'Angelo; Article No. 29; pp. 29:1–29:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

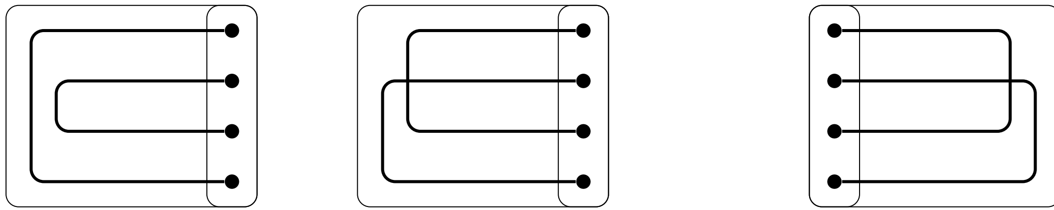

Figure 1 A separator S with two possible partial solutions on the left. Only the first one forms a Hamiltonian cycle with the partial solution on the right, despite that in all of them the vertices on the separator have degree 1.

Seymour in their Graph Minors project [20], measures how the input graph resembles a tree, or how can be covered be a set of bounded-sized bags organized in tree like structure which we call *tree decomposition*. It has proven to be very useful for dealing with $\mathcal{NP}$-hard problems; for example, given an n-vertex graph G and its tree decomposition of width t, one can solve the Maximum Independent Set problem in G in time $2^t \cdot t^{\mathcal{O}(1)} \cdot n$. We refer to [8] for more examples of algorithms on graphs of bounded treewidth.

Essentially every algorithm for graphs of bounded treewidth follows the paradigm of dynamic programming: it gradually (in a bottom-to-top fashion on the tree decomposition) builds partial solutions in subgraphs of the input graph. Using the fact that a bag in a tree decomposition is a separator, in many combinatorial problems it suffices to keep only a bounded (in the width of the decomposition) number of partial solutions in each step of the algorithm. To illustrate this concept, consider a separation (A, B) in a graph G with $S = A \cap B$ (i.e., $A, B \subseteq V(G)$ are two sets with $A \cup B = V(G)$ and no edge between $A \setminus B$ and $B \setminus A$), and think of a dynamic programming algorithm that processed already the graph $G[A]$, but has not yet touched $B \setminus A$. Observe that a partial solution $X \subseteq A$ to the Maximum Independent Set problem interacts with $B \setminus A$ only via the set S. Consequently, it suffices to store, for every $X_S \subseteq S$, an independent set $X \subseteq A$ of maximum possible size satisfying $X \cap S = X_S$. If the separator S is of size at most t, it leads to 2^t bound on the size of the memoization table in the dynamic programming algorithm.

In the Hamiltonian Cycle problem, the natural state space for the dynamic programming algorithm is a bit more complex. A partial solution in $G[A]$ would be a set of vertex-disjoint paths $\mathcal{P}$ that all have endpoints in S and together visit every vertex of $A \setminus B$. To complete the partial solution $\mathcal{P}$ to a Hamiltonian cycle H in G, it seems essential to remember not only which vertices of S are visited by $\mathcal{P}$ and which are the endpoints of paths in $\mathcal{P}$, but also how the paths of $\mathcal{P}$ pair up their endpoints in S (see Figure 1). This last piece of information leads to $2^{\theta(t \log t)}$ states for separator S of size t.

Up to late 2010, almost all known algorithms for combinatorial problems in graphs of bounded treewidth follow the naive approach outlined above, and researchers' effort focused mostly on speeding up computations in the so-called *join nodes* of the decomposition (see e.g. [22]).[1] In 2010, Lokshtanov, Marx, and Saurabh proved that many such algorithms have optimal dependency on treewidth [17] (under strong complexity assumptions) and provided a framework for proving similar lower bounds for complexities of the type $2^{\theta(t \log t)}$ [18]. However, providing such a tight lower bound for the connectivity problems such as Hamiltonian Cycle in graphs of bounded treewidth remained elusive.

[1] A join node of a decomposition corresponds to a node of the underlying tree of the tree decomposition of degree at least 3; intuitively, it corresponds to a bounded-size separator that splits the graph into more than 2 pieces, and in the dynamic programming algorithm one needs to merge information from at least two of such pieces.

Quite unexpectedly, a year after it turned out that there is a reason for this lack of progress, and a Monte Carlo algorithm with running time $4^t n^{\mathcal{O}(1)}$ for finding a Hamiltonian cycle in a graph with a given tree decomposition of width t has been reported [10]. The work [10] introduced a framework called Cut&Count that provided randomized single-exponential (i.e., with running time bound of the form $2^{\mathcal{O}(t)} n^{\mathcal{O}(1)}$) algorithms for many connectivity problems in graphs of bounded treewidth. The key idea of the Cut&Count method is to replace the original connectivity requirement with a different counting-mod-2 task, and ensure correctness via the Isolation Lemma [19].

In following years, a good understanding of the aforementioned improvement has been obtained by Bodlaender et al [6]. In the language of Hamiltonian Cycle, a linear algebra argument shows that it suffices only to keep 4^t partial solutions instead of the naive bound of $2^{\mathcal{O}(t \log t)}$; if the memoization table grows too large, an algorithm based on Gaussian elimination is able to prune provably unnecessary states. Cygan et al. [9] provided a better basis for the Gaussian elimination step and improved the bound for the number of states for Hamiltonian Cycle to $(2+\sqrt{2})^t$. Furthermore, in [9] a matching lower bound is shown. Due to the linear algebraic nature of the argument, this approach has been dubbed in the literature as the *rank-based approach*.

In [10], an involved fast convolution algorithm has been applied to obtain the $4^t n^{\mathcal{O}(1)}$ running time bound even in computations at join nodes. The need to execute Gaussian elimination in [6] and treat join nodes in a more direct fashion in both algorithms of [6, 9] yield worse theoretical running time bounds. Thus, the algorithm [10] remains theoretically fastest in graphs of bounded treewidth to this date.

Following a recent trend in multivariate algorithmics to experimentally evaluate parameterized algorithms (led by a growing popularity of the Parameterized Algorithms and Computational Experiments Challenge [12, 11]), in this work we thoroughly evaluate the aforementioned algorithms for Hamiltonian Cycle. A direct inspiration for our work is the work of Fafianie et al [13] that provided an experimental comparison of the naive and rank-based approaches for Steiner Tree (i.e., without considering the Cut&Count approach). That is, in this work we include Cut&Count and we compare the following four approaches.

naive The aforementioned naive approach with $2^{\mathcal{O}(t \log t)}$ bound on the number of states.

rank-based The approach of [6], that is, the naive approach with pruning of the state space leading to 4^t size bound.

rand-based with improved basis The approach of [9], that is, the rank-based approach with the improved basis yielding the size bound $(2+\sqrt{2})^t$.

Cut&Count The Cut&Count algorithm of [10].

As observed in [10], the application of the Isolation Lemma in the Cut&Count method yields a relatively high polynomial factor in the running time bound, but one can replace its usage with computations over a field of characteristic 2 and randomization via the Schwartz-Zippel lemma. This replacement leads again to linear dependency on the graph size in the running time bound. We follow this path. However, as has been overlooked in [10], the fast convolution algorithm at join nodes in the $4^t n^{\mathcal{O}(1)}$-time algorithm does not support computations over a field of characteristic 2, as it requires division by 2. Our theoretical contribution in this paper is a method around this obstacle, essentially showing that it is sufficient to perform the convolution over the ring of polynomials $\mathbb{Z}[x]$. This is described in Section 2.4 and leads to the following conclusion.

▶ **Theorem 1.1.** *There exists a Monte Carlo algorithm that, given an n-vertex graph G together with its tree decomposition of width t, solves* Hamiltonian Cycle *on G in time $4^t \cdot n \cdot (t \log n)^{\mathcal{O}(1)}$.*

In Section 2 we discuss implementation details of the algorithms. Section 3 discuss experiment setup and Section 4 discuss results. We conclude in Section 5.

2 Theory and implementation details

2.1 Tree decompositions

For more background on tree decompositions and dynamic programming algorithms using them, we refer to [8]. Here, we recall only the basic notions.

For a graph G, a *tree decomposition* is a pair (T, β) where T is a tree and β assigns to every node $t \in V(T)$ a set $\beta(t) \subseteq V(G)$ called *a bag* with the following invariants: (i) for every $v \in V(G)$, the set $\{t \in V(T) | v \in \beta(t)\}$ is nonempty and connected in T, (ii) for every $uv \in E(G)$ there exists $t \in V(T)$ with $u, v \in \beta(t)$. The width of the tree decomposition is the maximum size of a bag, minus one, and the treewidth of a graph is the minimum possible width of its tree decomposition.

As in multiple previous results, it is convenient to describe dynamic programming algorithms on a special type of decompositions, called *nice*. A *nice tree decomposition* is a rooted tree decomposition for which the bag of the root is empty and every node is of one of the following types:

Leaf node is a node t with no children and $\beta(t) = \emptyset$.

Introduce vertex node is a node t with unique child t' and a vertex v such that $\beta(t) = \beta(t') \uplus \{v\}$.

Forget vertex node is a node t with unique child t' and a vertex $v \in \beta(t')$ such that $\beta(t) = \beta(t') \setminus \{v\}$.

Join node is a node t with exactly two children t_1 and t_2 with $\beta(t) = \beta(t_1) = \beta(t_2)$.

For a note $t \in V(T)$, we define $\gamma_{\downarrow}(t)$ to be the union of $\beta(t')$ over t' being descendants of t in T. Furthermore, let G_t be the graph $G[\gamma_{\downarrow}(t)] - E(G[\beta(t)])$ (i.e., we exclude the edges inside the bag $\beta(t)$).

Additionally, in our case it is convenient to precede every **forget vertex node** with a sequence of **introduce edge nodes**. That is, for a **forget node** t with child t' and forgotten vertex v, we take $E_{t,v}$ to be the set of edges of G that connect v with vertices of $\beta(t) \setminus \{v\}$, subdivide the edge tt' in $E(T)$ $|E_{t,v}|$ times, labelling the new nodes $\{t_e | e \in E_{t,v}\}$, and set $\beta(t_e) = \beta(t')$. The graphs G_{t_e} are defined as follows: if t'' is the unique child of t_e, then $G_{t_e} = G_{t''} \cup \{e\}$.

The intuition of this step is as follows: there is a significant difference between the graphs $G_{t'}$ and G_t, namely $E(G_t) = E(G_{t'}) \cup E_{t,v}$. We split this change into $|E_{t,v}|$ steps, adding edges one by one.

All our implementations start with preparing a nice tree decomposition with the **introduce edge nodes**.

2.2 Naive approach

Given a note t in a nice tree decomposition (T, β), a *partial solution* is a family $\mathcal{P}$ of vertex-disjoint paths in G_t such that (i) every vertex of $\gamma_{\downarrow}(t) \setminus \beta(t)$ is visited by some path in $\mathcal{P}$, and (ii) every path in $\mathcal{P}$ has both its endpoints in $\beta(t)$. For a partial solution $\mathcal{P}$ at note t, we define the following objects:

a bucket b is a function $b : \beta(t) \to \{0, 1, 2\}$ that assigns to every vertex $v \in \beta(t)$ its degree in the union of $\mathcal{P}$;

a pairing E is a family of disjoint two-element subsets of $b^{-1}(1)$ that pairs up the endpoints of the same path in $\mathcal{P}$.

The pair (b, E) is the *state* of $\mathcal{P}$. The crucial observation is that among partial solutions with the same state, it suffices to memoize only one. Note that for a given bucket b with $\ell = |b^{-1}(1)|$, there are $(\ell-1)\cdot(\ell-3)\cdot 3\cdot 1$ possible pairings, giving a $2^{\theta(|\beta(t)|\log|\beta(t)|)}$ bound on the number of different states.

With this observation, it is straightforward to design a dynamic programming algorithm that finds a Hamiltonian cycle in time $2^{\mathcal{O}(t\log t)}n$ given a tree decomposition of the input graph of width t. This is exactly the naive approach.

2.3 Rank based approach

The rank-based approach is strongly based on the naive one, with main change being a pruning on the number of possible pairings.

▶ **Theorem 2.1** ([6]). *For a fixed node t and bucket b, given a family $\mathcal{E}$ of pairings, one can find a subfamily $\mathcal{E}^* \subseteq \mathcal{E}$ of size at most $2^{|b^{-1}(1)|-1}$ with the following property: for every Hamiltonian cycle H in G, if $\mathcal{P}$ is its intersection with G_t and (b, E) is the state of $\mathcal{P}$, and $E \in \mathcal{E}$, then there exists $E^* \in \mathcal{E}^*$ such that for every partial solution $\mathcal{P}^*$ with state (b, E^*), the graph $(H - E(\mathcal{P})) \cup E(\mathcal{P}^*)$ is a Hamiltonian cycle as well.*

Furthermore, given b and $\mathcal{E}$, one can assign to every $E \in \mathcal{E}$ a $2^{|b^{-1}(1)|-1}$-length 0-1 vector v_E such that the family $\mathcal{E}^*$ is defined as the indices of any maximal independent (over $\mathbb{F}_2$) subfamily of $\{v_E | E \in \mathcal{E}\}$.

In other words, for a fixed bucket b, it is sufficient to keep only $2^{|b^{-1}(1)|-1}$ pairings, and pruning unnecessary pairings can be done via Gaussian elimination on a matrix with $|\mathcal{E}|$ rows and $2^{|b^{-1}(1)|-1}$ columns over the field $\mathbb{F}_2$ (the two-element field modulo 2).

In [9], Theorem 2.1 is improved with a different construction of vectors v_E that are of length $2^{|b^{-1}(1)|/2-1}$. Furthermore, [9] showed how to use the special structure of the vectors v_E to avoid Gaussian elimination at **introduce/forget vertex/edge nodes**, yielding $(2+\sqrt{2})^p p^{\mathcal{O}(1)} n$-time algorithms for graphs with a given *path* decomposition of width p (i.e., without any **join nodes**).

We implement the rank-based approach both with the vector construction of [6] and the improved one of [9]. Both implementations use Gaussian elimination, as it is not known how to avoid it at join nodes.

In the implementation, the core of the naive and rank-based approaches is the same. We use two variants of the implementations: keep track of partial solutions (so that the entire Hamiltonian cycle can be returned in the end) or, in order to save space, just remember a flag and a Hamiltonian cycle is found via self-reducibility. All implementations perform the same computations specific to the node type, which are straightforward in all cases. At join nodes, the algorithm first sorts the partial solutions by buckets and then tries to match the partial solutions only for buckets that fit each other (e.g., do not exceed the bound of 2 on a degree of a vertex).

After succesfully computing the set of partial solutions for a current node the algorithm runs a reduce function. In the naive approach, it only deletes the duplicates by sorting set of partial solutions and checking if the two consecutive are same or not. In rank-based approach it divides all partial solutions into buckets (same as during processing the **join node**). For each bucket it computes the necessary matrix and performs Gaussian elimination on it to get a representative set of partial solutions.

To limit the effect of self-reducibility in case of the decision-only variant, we employ a problem-specific strategy. That is, we discover the Hamiltonian cycle edge-by-edge. For a path P in G with at least two edges, we can discover if G contains a Hamiltonian cycle containing P by deleting from G all edges of $E(G) \setminus E(P)$ that are incident to internal vertices of P, and run the decision algorithm on the obtained subgraph. Given a path P, we extend it one by one by doing a binary search over the next edge incident to an endpoint of P. This gives $\mathcal{O}(n \log \Delta)$ calls to the decision version of the problem for graphs with n vertices and maximum degree Δ.

2.4 Cut&Count approach

The main idea of the Cut&Count approach [10] is to replace search for a Hamiltonian cycle with counting the following objects: a cycle cover of the graph (i.e., a subset of edges where every vertex is of degree exactly two) with an assignment of every cycle to either left or right. In this manner, a fixed cycle cover with c cycles is counted 2^c times; if we additionally force one fixed vertex to be always on the left side, we get 2^{c-1} instead. That is, every Hamiltonian cycle is counted once, and every other cycle cover is counted an even number of times.

In [10], the Isolation Lemma [19] is employed to essentially reduce to the case when we solve instances with a unique Hamiltonian cycle. Then, the parity of the count described above indicates whether the graph contains a Hamiltonian cycle. However, this approach introduces a large polynomial overhead in the running time bound: first, because of the need for self-reducibility to discover the cycle (which we handle as in the previous section) and, second, because of the use of Isolation Lemma that adds an additional "weight" dimension to the dynamic programming memoization tables.

For the second overhead, as discussed [10], it can be remedied by, instead of using the Isolation Lemma, pick a field $\mathbb{F}$ of characteristic 2 (i.e., a field of size 2^p for some integer p), associate with each edge $e \in E(G)$ a variable x_e, associate with each cycle cover a monomial being a product of the variables associated with the edges used in the cycle cover, and compute the sum of the monomials over all cycle covers and all left/right assignment, using a random assignment of values from $\mathbb{F}$ to variables x_e. Then, if $\mathbb{F}$ is large enough (larger than the maximum degree of the monomial, which is n), the Schwarz-Zippel lemma ensures that with good probability the result is nonzero if and only if the graph has a Hamiltonian cycle (i.e., there is a small probability of a false negative).

In our implementation, we follow this path, using a field of size 2^{64}. This size is large enough so that the failure probability is neglible. On the other hand, there exists an efficient implementation of operations on this field using the PCLMULQDQ processor instruction for multiplication. Our implementation of the field operations follow [5].

Furthermore, as discussed in the introduction, the choice of computations over $GF(2^{64})$ rather than arguably simpler counting algorithms via the Isolation Lemma resulted also in technical problems in handling **join nodes**. As observed in [10], a natural and direct approach to a **join node** with bag of size t runs in time $9^t t^{\mathcal{O}(1)}$. In [10], this is speeded up by an involved fast convolution approach, reducing the 9^t factor to 4^t. At heart of this approach lies an algorithm to quickly compute the following convolution.

Let $f, g : \mathbb{Z}_4^m \to R$ for some ring R and integer m. We define $f * g : \mathbb{Z}_4^m \to R$ as

$$(f * g)(x) = \sum_{y \in \mathbb{Z}_4^m} f(y) g(x - y).$$

Here, the addition in $\mathbb{Z}_4^m$ is done coordinatewise. [10] developed a FFT-like approach to computing the above convolution, yielding the following.

▶ **Lemma 2.2** ([10]). *Given $f, g : \mathbb{Z}_4^m \to \mathbb{Z}$, the convolution $f * g$ can be computed in $4^m m^{\mathcal{O}(1)}$ operations on $\mathbb{Z}$ on values of the order of $2^{\mathcal{O}(m)}$ times larger than the maximum absolute value of the input functions.*

However, the proof of the above lemma involves division by a factor of 4^m, making it inapplicable directly to $R = GF(2^{64})$. To circumvent this obstacle, we developed a new variant of Lemma 2.2, building on the internal structure of the field $GF(2^{64})$. Recall that a field $GF(2^p)$ can be defined as the ring $\mathbb{Z}[x]$ divided by the ideal generated by 2 and an irreducible (in $\mathbb{F}_2[x]$) polynomial Q of degree p.

▶ **Lemma 2.3.** *Let $p \geq 1$ and assume that the elements of field $GF(2^p)$ are given as polynomials from $\mathbb{F}_2[x]$ of degree less than p, and multiplication in $GF(2^p)$ is done modulo a known polynomial Q of degree p. Given two function $f, g : \mathbb{Z}_4^m \to GF(2^p)$, the convolution $f * g$ can be computed in time $4^m(pm)^{\mathcal{O}(1)}$.*

Proof. We follow the same algorithm as in the proof of Lemma 2.2 from [10], but treating the values of f and g as elements of $\mathbb{Z}[x]$, not $GF(2^{64})$. This allows the necessary division steps in the algorithm, and an inspection of the proof of [10] shows that the algorithm operates on $\mathcal{O}(m)$-bit integers and polynomials of degree $\mathcal{O}(p)$. Then, at the very end, we reduce every resulting polynomial modulo 2 and modulo Q to obtain elements of $GF(2^{64})$. ◀

However, in the above we need to depart from the efficient implementation of operations in $GF(2^{64})$, and explicitly operate on polynomials in $\mathbb{Z}[x]$ of larger degree. While theoretically sound, this is expected to give a large overhead in experiments. Consequently, we test two variants of the Cut&Count algorithm: the one using a naive approach to the join nodes in time $9^t t^{\mathcal{O}(1)}$ and the one using Lemma 2.3.

To conclude the proof of Theorem 1.1, we observe that to ensure correctness with constant probability, the Cut&Count algorithm of [10] requires field $GF(2^p)$ with $p = \Omega(\log n)$.

3 Setup

3.1 Hardware and code

All of the computations were performed on a PC with an Intel Core i5-6500 processor and 16 GB of random-access memory. The operating system used during the experiments was Arch Linux. All implementations has been done in C++, the code is available at [1, 23].

3.2 Data sets

To evaluate our algorithms we decided to use well known set of Hamiltonian Cycle instances from Flinders Hamiltonian Cycle Project [15] consisting of 1001 instances. To find tree decompositions of small with, we first applied our implementation of the minimum fill-in heuristic (cf. [14]). The heuristic returned tree decompositions of width at most 8 for 623 instances, and indicated that 30 more instances may have treewidth within ranges allowing usage of our Hamiltonian Cycle algorithms.

We took the aforementioned 623 instances as our main benchmark. For sake of optimizing hyperparameters of our algorithms, we sampled a subset of 30 elements.

To the aforementioned 30 instances with larger but potentially tractable treewidth, we applied the heuristic of Ben Strasser [21] that won the second place in 2017 PACE Challenge [12]. This resulted in another 19 instances with tree decompositions of width between 17 and 29. Out of these instances, 15 turned out to be tractable by our algorithms.

Furthermore, we also sampled 7 random instances in the following way: starting from a Hamiltonian cycle C, we added a number of random edges with endpoints close on the cycle C (so that the treewidth is bounded). These instances are meant to generate many partial solutions at separator, and are expected to give large advantage to rank-based approaches.

To sum up, we operate on five data sets, all but the last being subsets of the Flinders Hamiltonian Cycle Project [15]:

set A is the whole set of graphs with small treewidth recognized by our heuristic (623 instances, treewidth at most 8),

set B is a subsample of A (30 instances, treewidth at most 8),

set C is the set of larger treewidth graphs with decompositions found by [21] (19 instances, treewidth beetwen 17 and 29).

set D is the subset of the set C that turned out to be tractable by our implementations (15 instances, treewidth beetwen 17 and 29).

set E is a set of 7 random graphs sampled as described above.

All instances from [15] are available through their webpage. At [1] we provide a list of the used instances in each set, the set E, and the used tree decompositions for sets C and D.

3.3 Fine-tuning the frequency of Gaussian elimination

As discussed in the introduction, in the rank-based approach the theoretical running time bound is worse than the one of Cut&Count approach partially due to the need of applying Gaussian elimination on the set of partial solutions. It is expected that the Gaussian elimination would also take substantial part of time resources in experiments.

In theory, the Gaussian elimination step is applied whenever the size of the set of partial solutions exceeds theoretical guarantees. However, in practice it seems reasonable that sometimes it pays off to apply this computationally expensive step less often; that is, allow the set of partial solutions to grow significantly beyond the theoretical bounds, and once in a while trim it at bulk with a single Gaussian elimination step. This intuition has been supported by the results of Fafianie et al [13] for the case of Steiner Tree.

Consequently, we start our experiments with fine-tuning the frequency of Gaussian elimination in both rank-based approaches we study. Since the width of the tree decomposition can play substantial role in deciding the optimal frequency, we do it separatedly for sets B and sets C.

In the next experiments, we use the optimum found frequencies for the algorithms based on both rank-based approaches. Note that the frequencies may differ between the low-treewidth regime (sets A and B) and the medium-treewidth regime (set C).

3.4 Comparison of the approaches

Having found the optimal frequency of the Gaussian elimination in the rank-based approaches, we run all four algorithms on every test in sets A, B, and C and compare results. In set C, every run has a timeout of 30 minutes. In set A, the timeout equals 10 minutes.

4 Results

In our experiments, it quickly became apparent that the variant of the naive and rank-based approach that stores all partial solutions (i.e., no self-reducibility) is significantly faster for small treewidth (sets A and B), while the self-reducibility one is faster for larger treewidth

Table 1 Fine-tuning results for test set B. Note that the second and third columns correspond to compression guarantees of the two studied algorithms, respectively.

ℓ	$2^{\ell-1}$	$2^{\ell/2-1}$	τ	Total running time on set B (SS.ms)	
				rank-based 4^t	rank-based $(2+\sqrt{2})^t$
4	8	2	3	1910.968	1318.385
			4	1827.949	1569.542
6	32	4	5	1901.457	1264.803
			7	1915.583	1286.522
			9	1960.515	1298.849
			11	1890.034	1316.813
			13	1876.483	1339.439
			15	1889.843	1401.620
			17	1923.244	1425.338
8	128	8	9	1896.748	1269.761
			18	1899.633	1290.696
			36	1996.629	1274.545
			72	1925.507	1268.261
			144	1863.837	1283.934

(sets C, D, and E). Thus, in what follows, we used the first one for experiments on small treewidth graph and the latter for larger treewidth graphs.

4.1 Fine-tuning the frequency of Gaussian elimination

4.1.1 Small treewidth

Recall that in sets A and B, the maximum size of the bag in the decomposition is 9. Consequently, in every state (b, E) the size of $b^{-1}(1)$ is at most 8 (as it must be even). The treatment of the states with $|b^{-1}(1)| \in \{0, 2\}$ does not use any of the involved rank-based techniques. Thus, we decided to separatedly fine-tune the frequency of applying the Gaussian elimination step to buckets with $|b^{-1}(1)|$ of size 4, 6, and 8 each. More formally, for $\ell \in \{4, 6, 8\}$ we fix a threshold τ and, for fixed bucket b with $|b^{-1}(1)| = \ell$ apply the Gaussian elimination step to the states (b, E) only if the number of these states is at least τ. While experimenting with one ℓ, the threshold for another sizes remains fixed. We perform tests on set B and report the total time used to find Hamiltonian cycles in all instances. The results are presented in Table 1.

From the results, it seems that lowering the frequency of Gaussian elimination does not help neither of the approaches, and evidently worsened the case for the improved base algorithm and $\ell = 4, 6$. The only exception seemed to be the case $\ell = 6$ and $\tau = 13$ for the worse base algorithm.

We see a number of good explanations for that. First, we think that case $\ell = 8$ appeared very rarely in the computations, and thus the impact of fine-tuning it has neglible effect in the overall result.

For the remaining cases, note that the matrices passed to the Gaussian elimination have at most 32 columns in the case of the first algorithm, and only 4 columns in the second. Thus, the Gaussian elimination step is very cheap in this regime of ℓ. Consequently, one does not gain much from lowering the frequency, while evidently losing by needing to maintain bigger memoization tables. This explains the worsening of the second algorithm for $\ell = 4, 6$

Table 2 Fine-tuning results for test set D. The Gaussian elimination step is applied to buckets b with $\ell = |b^{-1}(1)|$ and at least $\alpha \cdot 2^{\ell/2-1}$ states (b, E).

α	Total running time on set D (SS.ms)		α	Total running time on set D (SS.ms)	
	rank-based 4^t	rank-based $(2+\sqrt{2})^t$		rank-based 4^t	rank-based $(2+\sqrt{2})^t$
0.5	7363.078	2021.516	32	1802.851	1778.647
1	2704.165	1801.278	64	1797.416	1807.470
2	1925.618	1768.000	128	1794.877	1801.913
4	1813.478	1779.293	256	1801.104	1822.113
8	1792.872	1788.217	512	1795.312	1818.508
16	1806.994	1783.919	1024	1800.863	1800.698

and increasing τ.

However, one would expect that the first algorithm would also worsen with the increase of τ, but this is not supported by data. To explain this behavior, note that the values of τ used here are lower than the theoretical guarantees of the algorithm. Consequently, the pruning of the memoization tables in the first algorithm seem to give very little in these cases.

In other words, the pruning capabilities of the vectors v_E used by the first algorithm are much weaker for low values of ℓ than the capabilities of the second algorithm. This is most striking in the case $\ell = 4$: there are 3 pairings of a 4-element set; the first algorithm keeps all of them if present, while the second one notices that one is redundant and deletes it.

To sum up, the data indicates that decreasing the frequency of the Gaussian elimination step does not help for small values of ℓ, while the first algorithm with the worse pruning capabilities does not offer much pruning in this regime of values of ℓ.

4.1.2 Larger treewidth

For fine-tuning in graphs of larger treewidth, we use set D. Here, we propose slightly different threshold behavior: we fix a parameter α and, for fixed bucket b with $\ell = |b^{-1}(1)|$, we apply the Gaussian elimination step if the number of states (b, E) exceed $\alpha \cdot 2^{\ell/2-1}$ (i.e., α is a multiplicative parameter relative to the pruning size guarantee of the second algorithm). The results are gathered in Table 2.

The results indicate that a mild increase of the threshold (i.e., $\alpha = 2$) increases the speed of the second algorithm, while further increase of the threshold slowly worsenes the bounds. For the first algorith, the sweet spot seems to be slightly later, and further increase of the threshold does not necessarily worsen the algorithm.

The gain from mild increase in the case of both algorithms can be explained by the fact that for larger values of ℓ, the Gaussian elimination step starts to be costly. In the case of the first algorithm, we think that its pruning capabilities are limited for the Hamiltonian cycle problem, and thus further increase of the threshold does not change much.

To sum up, both algorithms definitely slow down if the Gaussian elimination step is done too frequently. The data showed optimum values $\alpha = 8$ for the first algorithm and $\alpha = 2$ for the second.

4.2 Comparison

As discussed in Section 2.4, we have implemented two variants of the Cut&Count algorithm: the one that uses the fast convolution at **join nodes** (Lemma 2.3) and the one that does it more naively in time bounded by $9^t t^{\mathcal{O}(1)}$.

Table 3 Total running times for test set A (timeout 10 minutes per instance). The Cut&Count program did not finish within alloted time on 124 instances, the remainder solved all test cases. In the first row, we show total running time on all 499 tests solved by all programs.

	naive	rank-based 4^t	rank-based $(2+\sqrt{2})^t$	Cut&Count
499 tests finished by all	5993.633	7383.249	5919.392	46650.101
all 623 tests	11532.153	13675.58	10278.827	-

Table 4 Running times for test sets C and E. Hyphen means timeout (30 minutes). For the set C, the "tw" column indicates the width of the used tree decomposition (found by the algorithm of Strasser [21]). Tests where all algorithms were timeouted are not presented here.

test	$\|V(G)\|$	$\|E(G)\|$	tw	naive	rank-based 4^t	rank-based $(2+\sqrt{2})^t$	Cut&Count
0074	462	756	28	38.737	109.655	110.040	-
0109	606	933	17	.063	.086	.085	.611
0110	606	925	17	.066	.089	.090	.471
0144	804	1256	21	.190	.253	.231	205.128
0145	804	1252	21	.137	.187	.186	3.549
0172	1002	1575	25	1.156	1.298	.554	-
0173	1002	1579	25	.459	.598	.475	215.115
0199	1200	1902	29	13.513	15.419	3.369	-
0200	1200	1902	26	3.673	6.900	1.544	-
0253	1578	2688	29	93.343	167.458	167.440	-
0268	1644	2767	25	36.449	70.157	69.111	-
0271	1662	2770	29	28.149	33.145	33.208	-
0272	1662	2863	25	554.271	1260.329	1230.722	-
0290	1770	3020	25	57.901	83.781	82.386	-
0298	1806	3071	23	10.035	18.611	18.492	-
E0001	360	566		371.775	-	64.390	-
E0002	600	886		204.197	-	28.882	-
E0003	700	1139		-	-	711.778	-
E0007	360	655		1575.475	-	328.191	-

We found out that the one with the fast convolution behaves very slowly even on small tests. This can be easily explained by the hidden complexity of ring computations inside Lemma 2.3. Consequently, while theoretically sound, we dropped it from further experiments and considered only the Cut&Count algorithm without the fast convolution.

For test set A, we have used a timeout of 10 minutes per instance. A CSV file with full results can be found on the project website [1]. Table 3 presents summary; the Cut&Count algorithm did not finish in time for 124 tests, and thus we compare its running time on the other 499 tests. For sets C and E, full results are in Table 4 (for set E only naive and improved rank-based algorithms were executed).

The first corollary from the results is that the Cut&Count approach does not turn out to be practical, and is heavily outperformed by other approaches. We see some good explanations for that. First, all other approaches are "positive-driven": they keep only values in their memoization tables that correspond to found partial solutions, and in many cases there can be much fewer such partial solutions that the worse-case theoretical bound. In particular, these approaches can implicitly use some hidden structure of the input graph, such as planarity. The Cut&Count approach, on the other hand, relies on computing coefficients for partial cycle covers, and – even with our positive-driven implementation that keeps only nonzero elements – keeps track of much more partial solutions that the other approaches.

This effect is even stronger if one tries to use fast convolution at **join nodes**: the convolution fills up the entire table of 4^t values being polynomials, even if the input functions were sparse.

Second, the Cut&Count approach solves only a decision version of the problem, yielding large overhead from some self-reducibility application, while all other algorithms return the Hamiltonian cycle in question straight away.

For the other approaches, it is noticeable that the first rank-based approach (with 4^t guarantee on the size of the memoization tables) is clearly outperformed by the naive approach. That is, the cost of the Gaussian elimination step does not pay back in savings of the size of memoization tables. This can be explained as already discussed in the previous section: the vectors used in this algorithm are too weak to effectively prune the memoization tables, which is particularly visible on buckets b with small $\ell = |b^{-1}(1)|$.

Results from small treewidth graphs (set A) show also that the improved rank-based approach outperforms the naive one by roughly 10%. For larger treewidth (set C), the situation is more complicated: on some tests the rank-based approach outperforms the naive one by significant factor (0172, 0199, 0200), while sometimes it is opposite (0074, 0272). As expected, the artificially generated random instances gave big advantage to the rank-based approach.

A natural question is why we see only 10% increase despite significant asymptotic gain in the analysis ($2^{\mathcal{O}(t \log t)}$ vs $(2+\sqrt{2})^t$). Apart from the obvious answers to this questions (the values of t we are studying are low for asymptotic analysis), we would like to point out another, problem-specific reason. The difference between the naive approach and the rank-based one is only within handling states for one fixed bucket b, and there are up to 3^t different buckets. Iterating over all non-empty buckets is a common part of both approaches, and can be responsible for most of their running time.

To sum up, the only approach competitive with the naive approach is the improved rank-based approach with the $(2+\sqrt{2})^t$ guarantee on the size of memoization tables. However, its gain is limited, and there are multiple cases where the use of Gaussian elimination steps is not helpful at all.

5 Conclusions

We have experimentally evaluated multiple known approaches to solve Hamiltonian Cycle in graphs of bounded treewidth. The results show that the Cut&Count approach is impractical, while the improved rank-based approach of [9] consistently outperforms the more generic one of [6]. Furthermore, the latter seem to help little and is outperformed by the naive solution.

The comparison between the naive solution and the improved rank-based one of [9] is more intricate. On graphs of small treewidth, the second one outperforms the first one by 10% margin. For larger treewidth, the results are rather indecisive.

The results indicate potential in the improved rank-based algorithm of [9] and point to the need of further theoretical study of this approach. In [9], the authors show how to perform pruning without the need of Gaussian elimination at **introduce/forget nodes**. The question of matching the $(2+\sqrt{2})^t t^{\mathcal{O}(1)}$ running time bound for **join nodes** remains open, and a positive answer to this question may lead to significantly faster implementation. Also, we did not try to mix the Gaussian elimination steps at **join nodes** with the other steps at **introduce/forget nodes**.

Finally, we found it quite remarkable that 638 out of 1001 instances of Flinders Hamiltonian Cycle Challenge [15] (i.e., our sets A and D) could be solved with the naive bounded treewidth routine on a personal computer, while 623 out of them (our set A) have one-digit treewidth.

References

1. Recent trends in kernelization: theory and experimental evaluation — project website. 2018. http://kernelization-experiments.mimuw.edu.pl.
2. Richard Bellman. Combinatorial processes and dynamic programming. Technical report, RAND CORP SANTA MONICA CA, 1958.
3. Andreas Bjorklund. Determinant sums for undirected hamiltonicity. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 173–182. IEEE, 2010.
4. Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Trimmed moebius inversion and graphs of bounded degree. *Theory of Computing Systems*, 47(3):637–654, 2010.
5. Andreas Björklund, Petteri Kaski, Lukasz Kowalik, and Juho Lauri. Engineering motif search for large graphs. In *2015 Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 104–118. SIAM, 2014.
6. Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Inf. Comput.*, 243:86–111, 2015. doi:10.1016/j.ic.2014.12.008.
7. Hajo Broersma, Fedor V Fomin, Pim van't Hof, and Daniël Paulusma. Fast exact algorithms for hamiltonicity in claw-free graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 44–53. Springer, 2009.
8. Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
9. Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Fast hamiltonicity checking via bases of perfect matchings. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 301–310. ACM, 2013.
10. Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michal Pilipczuk, Joham MM van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 150–159. IEEE, 2011.
11. Holger Dell, Thore Husfeldt, Bart MP Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A Rosamond. The first parameterized algorithms and computational experiments challenge. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 63. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
12. Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration.
13. Stefan Fafianie, Hans L Bodlaender, and Jesper Nederlof. Speeding up dynamic programming with representative sets: an experimental evaluation of algorithms for steiner tree on tree decompositions. *Algorithmica*, 71(3):636–660, 2015.
14. Serge Gaspers, Joachim Gudmundsson, Mitchell Jones, Julián Mestre, and Stefan Rümmele. Turbocharging treewidth heuristics. In Jiong Guo and Danny Hermelin, editors, *11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24-26, 2016, Aarhus, Denmark*, volume 63 of *LIPIcs*, pages 13:1–13:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.IPEC.2016.13.
15. M. Haythorpe. FHCP challenge set, 2015. http://fhcp.edu.au/fhcpcs.
16. Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
17. Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Known algorithms on graphs on bounded treewidth are probably optimal. In Dana Randall, editor, *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San*

Francisco, California, USA, January 23-25, 2011, pages 777–789. SIAM, 2011. doi:10.1137/1.9781611973082.61.

18 Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Slightly superexponential parameterized problems. In Dana Randall, editor, *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 760–776. SIAM, 2011. doi:10.1137/1.9781611973082.60.

19 Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987. doi:10.1007/BF02579206.

20 Neil Robertson and Paul D Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

21 Ben Strasser. Computing tree decompositions with flowcutter: PACE 2017 submission. *CoRR*, abs/1709.08949, 2017. arXiv:1709.08949.

22 Johan M. M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009, 17th Annual European Symposium, Copenhagen, Denmark, September 7-9, 2009. Proceedings*, volume 5757 of *Lecture Notes in Computer Science*, pages 566–577. Springer, 2009. doi:10.1007/978-3-642-04128-0_51.

23 Michał Ziobro and Marcin Pilipczuk. Finding Hamiltonian Cycle in graphs of bounded treewidth: Experimental evaluation. code repository, 2018. https://github.com/stalowyjez/hc_tw_experiments.