Engineering Motif Search for Large Graphs[*]

Andreas Björklund[†] Petteri Kaski[‡] Łukasz Kowalik[§] Juho Lauri[¶]

Abstract

In the *graph motif problem*, we are given as input a vertex-colored graph H (the host graph) and a multiset of colors M (the motif). Our task is to decide whether H has a connected set of vertices whose multiset of colors agrees with M. The graph motif problem is NP-complete but known to admit parameterized algorithms that run in linear time in the size of H. We demonstrate that algorithms based on *constrained multilinear sieving* are viable in practice, scaling to graphs with hundreds of millions of edges as long as M remains small. Furthermore, our implementation is *topology-invariant* relative to the host graph H, meaning only the most crude graph parameters (number of edges and number of vertices) suffice in practice to determine the algorithm performance.

1 Introduction.

A basic question in algorithm engineering is to understand which graph problems admit algorithms that scale *in practice* to large graphs with hundreds of millions of edges and beyond. Our present focus is the following search problem (see §1.3 for motivation):

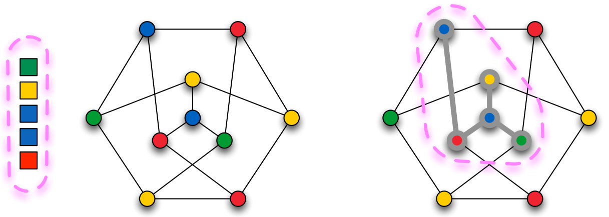
The Graph Motif Problem. We are given as input

(a) a multiset M of size k (the *query* or *motif*), and

(b) a vertex-colored graph H with n vertices and m edges (the *data* or *host graph*).

Our task is to decide whether H has a *connected set*[1] of k vertices whose multiset of colors agrees with the multiset M.

Example. The motif and the host (left); a connected set that matches the motif (right).



From an algorithm theory perspective, in particular from the perspective of *parameterized algorithms* [13], the graph motif problem is known to have a janusian nature: it is (i) NP-complete, but (ii) admits algorithms that run *in linear time* in the size of the host graph.[2] Hence even if a polynomial-time algorithm is perhaps not to be expected, such problems and associated algorithms present an opportunity in engineering terms: even if the theoretical worst-case running time scales exponentially, this exponential complexity can be isolated to a parameter k that is *independent* of the size of the graph. *Thus, for such problems there are no theoretical barriers that would prevent practical scalability, as long as the parameter k remains small, and sufficient engineering effort is put to the algorithm implementation.*

1.1 Our Contribution.

In this paper we demonstrate that (i) the graph motif problem admits practical algorithm implementations that scale to large graphs as long as k remains small, and (ii) our present theoretical basis for these algorithms, *constrained multilinear sieving*, is a design framework that translates to practical algorithms if sufficient engineering effort is put to the implementation. In particular, our effort is aimed at

[*]Supported in part by the Swedish Research Council, Grant VR 2012-4730 (A.B.), by the European Research Council, Starting Grant 338077 "Theory and Practice of Advanced Search and Enumeration" (P.K.), and by the National Science Centre of Poland, Grant UMO-2013/09/B/ST6/03136 (Ł.K.). We acknowledge use of computer resources funded by project "Science-IT" at Aalto University.

[†]Department of Computer Science, Lund University

[‡]Helsinki Institute for Information Technology HIIT & Department of Information and Computer Science, Aalto University

[§]Institute of Informatics, University of Warsaw

[¶]Helsinki Institute for Information Technology HIIT & Department of Information and Computer Science, Aalto University. *Current affiliation:* Department of Mathematics, Tampere University of Technology

[1]A set of vertices of H is *connected* if the set induces a connected subgraph of H.

[2]Indeed, a routine reduction from the Steiner tree problem shows that the decision version of the graph motif problem is NP-complete already when the motif M has two distinct colors. Yet, the complexity can be isolated to the motif M. Indeed, the decision problem admits a randomized algorithm that runs in time $O(2^k k^2 (\log k)^2 m)$ and admits parallelization with linear theoretical speedup over the exponential part of the running time [6, 7]. Thus, despite NP-completeness, the graph motif problem is solvable, at least in theory, in linear time in the number of edges m in the host graph. See §1.3.

a high-performance implementation on a single shared-memory multiprocessor.

For small values of k, our experiments (§4) show that the graph motif problem can be solved in practice on graphs up to a hundred million edges on a small-memory (single-processor) desktop computer and billions of edges on a large-memory (multiprocessor) compute node. Our implementation is empirically both linear-time and *topology-invariant* relative to the host graph H, that is, only the most crude graph parameters (number of edges m, number of vertices n) suffice in practice to determine the algorithm performance. The current experimental version of our implementation is available as open source [9].

1.2 Implementation Challenges. With scaling to large graphs as the objective, design and optimization for current shared-memory microarchitectures [27] presents at least the following engineering challenges:

(a) *Memory consumption.* Scalability to large graphs requires that the algorithm has a small random-access memory (working memory) footprint. Ideally, the working memory should be proportional to the number of vertices n in the graph, or less.

(b) *Graph traversal and the memory interface.* A recurrence that traverses the edges of a large input forces an essentially arbitrary pattern of accesses to main memory. This leads to inefficient use of the main memory interface *unless* the accesses are executed (i) one cache line at a time, utilizing all the words in a line, (ii) in parallel, utilizing all available execution cores[3], and (iii) by utilizing hardware and software prefetching to reduce latency.

(c) *Parallel execution and fast arithmetic.* Constrained multilinear sieving relies on arithmetic over fields of characteristic 2, requiring use of dedicated hardware/software techniques for fast implementation. Here a natural guideline is to rely on (i) vectorization (single-instruction-multiple-data parallelism) to implement low-level arithmetic at each vertex, and (ii) multiple execution cores (instruction-level parallelism) to parallelize across vertices.

We address these challenges as follows. First, we recall that constrained multilinear sieving represents the host graph H as an implicit multivariate polynomial $P = P_H$ (cf. §2.1) [6, 7]. The core of such a design is the algorithm that performs *evaluations* of P at specific points. To reduce memory consumption, we develop here two novel generating functions (§3.1 and §3.2) and their concrete evaluation algorithms that operate with $O(kn)$ and $O(n)$ words of working memory, respectively, whereas previous abstract designs [6, 7] required $O(km)$ memory. Second, to efficiently utilize the memory interface and the available instruction set extensions, we *vectorize* both the memory layout of the evaluation algorithms and the low-level arithmetic subroutines (§3.3 and §3.4). Third, to better saturate the memory interface and to benefit from multiple execution cores, we parallelize the recurrences that evaluate P so that each thread of execution is responsible for the recurrences of a group of vertices of H (§3.5). Fourth, to alleviate memory latency to main memory, we structure the memory layout to be conductive to hardware prefetching, and use software prefetching to expedite indirect, data-dependent accesses (§3.6). Finally, we engage in some routine data structure engineering to support extraction and listing of solutions (§3.7 and §3.8).

1.3 Motivation and Earlier Work. Pattern search on sequences and trees is a classical topic with a wealth of applications and algorithm designs. The graph motif problem is a natural and hard generalization of *jumbled pattern matching* problems on sequences (paths) and trees (e.g. [12, 22, 23, 30]) to graphs with arbitrary topology. Graph motif problems were introduced by Lacroix *et al.* [35] in the context of metabolic network analysis in bioinformatics, with further variants and extensions of the problem introduced by at least Dondi, Fertin, and Vialette [16], Björklund *et al.* [6, 7], and Pinter and Zehavi [43, 44, 49]. We restrict our present study to the basic k-sized motif problem.

The graph motif problem remains NP-complete even when the host H is a tree of maximum degree 3 and no color occurs more than once in the motif [20]. Fellows *et al.* [19] isolated the complexity of the problem to the parameter k (the motif size) by presenting a fixed-parameter $O^*(f(k))$-time[4] algorithm relying on color coding [1]. A subsequent race to improve the parameter dependency $f(k)$ ensued [2, 6, 7, 26, 32, 42], with the three most recent contributions consisting of the randomized $O^*(2.54^k)$-time algorithm of Koutis [32], the randomized $O^*(2^k)$-time algorithm of Björklund *et al.* [6, 7], and the deterministic $O^*(5.22^k)$-time algorithm of Pinter, Scachnai, and Zehavi [42]. There is complexity-theoretic evidence that algorithms with running time $O^*((2-\epsilon)^k)$ for any constant $\epsilon > 0$ do not exist [6, Theorem 6]. To our knowledge, we are the first to pursue a concrete algorithm implementation that scales to large graphs in a topology-invariant manner.

[3]A single execution core in general is not sufficient to saturate the memory interface, especially if multiple channels to main memory are available. Cf. Tables 1, 2, and 3.

[4]The $O^*(\cdot)$ notation suppresses a multiplicative factor that is polynomial in the input size.

Our present design framework, constrained multilinear sieving, originates from seminal work of Koutis [31], Williams [48], Koutis and Williams [33], and Koutis [32] in the context of group algebras. However, rather than work with group algebras we find it more convenient to pursue an implementation via multivariate polynomials and inclusion–exclusion sieving by polynomial substitution [4, 5, 6, 7].

Turning to our engineering contributions, our generating function designs can be traced back to Nederlof's [40] insight (in the context of the Steiner tree problem) that *branching walks* can be used to witness arbitrary connected sets of vertices in a host graph; Guillemot and Sikora [26] transported this insight to the graph motif problem. Our contribution is to modify the Guillemot–Sikora design and our own earlier designs [6, 7] to use more indeterminates so that less bookkeeping is required in dynamic programming, which enables us to reduce from $O(km)$ memory to $O(kn)$ memory, without increasing the running time.

Bitwise techniques such as *bit packing* to enable vectorization are classical techniques in algorithm engineering; see Knuth [29, §7.1.3] and Warren [47]. The use of processor registers to operate on multiple scalars simultaneously can be traced back to Lamport [36] and Fisher and Dietz [21]. Here we will apply such techniques to implement fast finite-field arithmetic, taking advantage of instruction set extensions (carryless multiplication [25] and vector instructions [28, Vol. 1, §14]) to produce a fast bit-packed implementation (cf. Gueron and Kounavis [24]). If instruction set extensions are not available, will rely on *bit slicing* (see Biham [3] and Rudra *et al.* [45]) for fast vectorized multiplication subroutines for small field sizes.

2 Preliminaries.

This section sets up terminology and background on existing algorithms. Let H be the given vertex-colored host graph with n vertices and m edges, and let M be the given motif of size k. Let us write $V(H)$ for the vertex set and $E(H)$ for the edge set of H. For a vertex $u \in V(H)$, we write $N_H(u)$ for the set of vertices adjacent to u in H. For graph-theoretic terminology we refer to [15].

2.1 Deciding the Existence of a Solution.

Our algorithm design is based on a randomized algebraic sieve that decides whether the given H has at least one connected set that agrees with the given M. This section reviews the preliminaries for constrained multilinear sieving [6, 7].

A high-level intuition is as follows. From the given input (H, M) we define (but never construct in explicit form) a generating polynomial $P = P_H$ that contains monomial witnesses for all connected sets in H. We then perform repeated random evaluations of P (using dedicated evaluation algorithms, to be described in §3.1 and §3.2) at specific points that depend both on M and on the colors at the vertices of H, and finally make the YES/NO decision based on the values we obtain from those evaluations.

We now proceed with the technical details of the sieve. All arithmetic takes place in a field of characteristic 2. Let us assume for convenience that $V(H) = [n] = \{1, 2, \ldots, n\}$. Let $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ be a tuple consisting of n *domain variables* and let $\boldsymbol{y} = (y_1, y_2, \ldots, y_s)$ be a tuple consisting of s *supporting variables.* Let $P(\boldsymbol{x}, \boldsymbol{y})$ be a multivariate polynomial whose every monomial $x_1^{d_1} x_2^{d_2} \cdots x_n^{d_n} y_1^{e_1} y_2^{e_2} \cdots y_s^{e_s}$ has total $\boldsymbol{x}$-degree $d_1 + d_2 + \ldots + d_n = k$ and whose coefficients are over a field of characteristic 2. (The precise definition of P will be postponed to §3.1 and §3.2.) A monomial $x_1^{d_1} x_2^{d_2} \cdots x_n^{d_n} y_1^{e_1} y_2^{e_2} \cdots y_s^{e_s}$ is *$\boldsymbol{x}$-multilinear* if for all $i \in [n]$ it holds that $d_i \in \{0, 1\}$.

Let C be a set of *colors.* Associate a color $c(i) \in C$ with each index $i \in [n]$. For each color $q \in C$, let us write $M(q)$ for the number of occurrences (the multiplicity) of q in the given motif M. We say that a monomial $x_1^{d_1} x_2^{d_2} \cdots x_n^{d_n} y_1^{e_1} y_2^{e_2} \cdots y_s^{e_s}$ is *properly colored* if for all $q \in C$ it holds that $M(q) = \sum_{i \in c^{-1}(q)} d_i$.

For each color $q \in C$ let S_q be a set of $M(q)$ *shades* of the color q, such that $S_q \cap S_{q'} = \emptyset$ if $q \neq q'$. For each index $i \in [n]$ and each shade $d \in S_{c(i)}$, introduce a variable $v_{i,d}$. Let L be a set of k *labels.* For each shade $d \in \bigcup_{q \in C} S_q$ and each label $j \in L$, introduce a variable $w_{d,j}$. Denote by $\boldsymbol{v}$ and $\boldsymbol{w}$ the tuples consisting of the introduced variables.

Lemma 2.1. (Constrained Multilinearity [6, 7]) *The polynomial $P(\boldsymbol{x}, \boldsymbol{y})$ has at least one monomial that is both $\boldsymbol{x}$-multilinear and properly colored if and only if the polynomial*

$$Q(\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y}) = \sum_{A \subseteq L} P\big(\boldsymbol{u}_A, \boldsymbol{y}\big) \tag{2.1}$$

is not identically zero, where the tuple $\boldsymbol{u}_A = (u_{1,A}, u_{2,A}, \ldots, u_{n,A})$ is defined for all $i \in [n]$, $j \in L$, and $A \subseteq L$ by

$$u_{i,A} = \sum_{j \in A} u_{i,j} \quad \text{and} \quad u_{i,j} = \sum_{d \in S_{c(i)}} v_{i,d} w_{d,j} \,. \tag{2.2}$$

The serendipity of Lemma 2.1 now arises from the fact we need not compute with polynomials in explicit form (as a sum of monomials) but rather we can compute in a homomorphic image (evaluation at a chosen point). This is enabled by the observation that a not-identically-zero polynomial of low degree does not have too many roots, so if we select a point uniformly at random, we are likely to witnesses with fair probability that Q is not identically zero:

LEMMA 2.2. (NUMBER OF ROOTS [14, 46, 50]) *A not-identically-zero polynomial $Q(z_1, z_2, \ldots, z_N)$ of total degree D with coefficients in the finite field $\mathbb{F}_q$ has at most Dq^{N-1} roots in $\mathbb{F}_q^N$.*

We can now summarize the decision algorithm. First, we select independent uniform random values for the variables $\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y}$ in a large enough finite field. Then, we evaluate Q at this selected random point, which translates via (2.1) into 2^k evaluations of $P(\boldsymbol{x}, \boldsymbol{y})$ at 2^k different points $\boldsymbol{x}$ determined from $\boldsymbol{v}, \boldsymbol{w}$ via (2.2). Once the evaluation is complete, we output YES if $Q(\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y})$ is nonzero; otherwise we output NO. The algorithm outputs a false negative with probability at most D/q, where $D = 2k - 1$ (this arises from the total degree of the generating polynomials to be described in §3.1 and §3.2) and q depends on our chosen field of characteristic 2; for example, we can take $q = 2^{64}$ (see §3.4 for implementation) to ensure a negligible probability for false negatives. The algorithm has no false positives.

2.2 Extracting One Solution.

We rely on prior recent work by a subset of the present authors [8] to extract individual solutions by using the decision algorithm as a black-box subroutine.

Suppose we are given a host graph H and a motif M of size k. Let us write $\mathcal{F}$ for the set of all connected sets $W \subseteq V(H)$ that have size k and whose colors agree with M. Observe that the decision algorithm from §2.1 enables us to decide whether $\mathcal{F}$ is empty, but does not directly enable us to determine a concrete witness $W \in \mathcal{F}$ for nonemptiness.

Using self-reducibility, we can turn the decision algorithm into an *interval oracle* that takes as input an interval $[L, U]$ with $L \subseteq U \subseteq V(H)$, and decides whether there is a $W \in \mathcal{F}$ with $L \subseteq W \subseteq U$. Indeed, to execute such an *interval query* $[L, U]$ on H and the motif M, we can proceed as follows: (i) delete from H all the vertices outside U, (ii) delete from M the multiset of colors of the vertices in L (or give a NO output if the colors in L do not appear in M with sufficient multiplicity to enable deletion), (iii) color the vertices in L with a new color that does not appear elsewhere in H, (iv) insert the new color with multiplicity $|L|$ into M, and (v) run the decision algorithm and report its decision as the answer to the interval query. In particular, one query to the decision algorithm suffices to execute one interval query.

We can then rely on techniques from combinatorial group testing [18] to transform the interval oracle into an algorithm that finds a solution if one exists. In more precise terms, by executing at most $2k(\log_2 \frac{n}{k} + 2)$ interval queries (where L is fixed and U varies, so the queries are in fact *inclusion* queries; cf. [8]), we can for given $[L, U]$ either (i) find a $W \in \mathcal{F}$ with $L \subseteq W \subseteq U$ or (ii) conclude that no such W exists; cf. [8, Algorithm 1 and Lemma 2.1] and [17]. Moreover, Björklund *et al.* [8] show that even if the interval oracle gives a false negative with probability at most $1/4$, there is an algorithm making $O(k \log n)$ queries *in expectation.*

2.3 Listing All Solutions.

Following ideas due to Lawler [37, §2] we can turn an interval extractor into a recursive *interval lister* that lists all $W \in \mathcal{F}$ in a given interval $[L, U]$. Indeed, first, run the interval oracle on the given interval $[L, U]$. If the interval oracle answers NO, stop. If the interval oracle outputs YES, run the interval extractor on $[L, U]$ to get a witness $W \in \mathcal{F}$ with $L \subseteq W \subseteq U$. Output W and let $W \setminus L = \{x_1, x_2, \ldots, x_t\}$. For $i = 1, 2, \ldots, t$, invoke the interval lister recursively with the interval $[L_i, U_i]$, where $L_i = L \cup \{x_1, x_2, \ldots, x_{i-1}\}$ and $U_i = U \setminus \{x_i\}$.

3 Implementation Engineering.

This section describes our engineering contributions.

3.1 A Memory-Efficient Generating Function.

We develop a novel generating polynomial $P(\boldsymbol{x}, \boldsymbol{y})$ that, together with multilinear sieving to cancel contributions from unwanted monomials, captures the connected sets in the host graph H. In particular, we will rely on the graph-theoretic notion of an *arborescence* to witness that a set of vertices in H is connected. Furthermore, we will *label* the edges of the arborescence to enable localization of the evaluation at vertices of the graph, which results in improved memory-efficiency compared with an earlier [6, 7] generating function that localizes at edges.

An *arborescence* A of *size* $\ell \geq 1$ *rooted at* vertex $u \in V(H)$ is a subtree of H that contains vertex u and whose $\ell - 1$ edges are oriented so that there is a (unique) path of directed edges in A from u to every other vertex $w \in V(A)$. We say that the arborescence *spans* the vertices in $V(A)$.

It is immediate that arborescences can be used to witness connected sets, that is, for every connected set $W \subseteq V(H)$ there is at least one arborescence in H that spans W.

A *tightly labeled* arborescence is an arborescence A of size ℓ rooted at u whose edges (if any) have been labeled with the integers $2, 3, \ldots, \ell$ so that (i) the maximum label of an edge is ℓ, (ii) the maximum label occurs on a unique edge e, (iii) the edge e is incident with the root u, and (iv) deleting e from A produces two tightly labeled arborescences. From the definition it follows immediately that an arborescence A admits exactly $\prod_{w\in V(A)} d_A^+(w)!$ tight labelings, where $d_A^+(w)$ is the out-degree of the vertex w in A.

Let us now introduce a generating function $P(\boldsymbol{x},\boldsymbol{y})$ whose $\boldsymbol{x}$-multilinear monomials will be in a one-to-one correspondence with the tightly labeled arborescences of size k in H.[5] Introduce (i) one indeterminate x_u for each vertex $u \in V(H)$, and (ii) one indeterminate $y_{\ell,uv}$ for each orientation $uv = (u,v)$ of each undirected edge $\{u,v\} \in E(H)$ and each integer label $\ell = 2,3,\ldots,k$. In total we thus have n domain variables in $\boldsymbol{x}$ and $s = 2(k-1)m$ supporting variables in $\boldsymbol{y}$.

Let us define the generating function inductively in terms of the size parameter ℓ. The base case at size $\ell = 1$ is defined for all $u \in V(H)$ by

$$P_{1,u}(\boldsymbol{x},\boldsymbol{y}) = x_u\,. \tag{3.3}$$

The recurrence that extends the size by one is defined for all $u \in V(H)$ and $\ell = 2,3,\ldots,k$ by

$$P_{\ell,u}(\boldsymbol{x},\boldsymbol{y}) = \sum_{v\in N_H(u)} y_{\ell,uv} \sum_{\substack{\ell_1+\ell_2=\ell\\ \ell_1,\ell_2\geq 1}} P_{\ell_1,u}(\boldsymbol{x},\boldsymbol{y})P_{\ell_2,v}(\boldsymbol{x},\boldsymbol{y})\,. \tag{3.4}$$

Finally, we sum over the vertices to get the desired generating polynomial

$$P(\boldsymbol{x},\boldsymbol{y}) = \sum_{u\in V(H)} P_{k,u}(\boldsymbol{x},\boldsymbol{y})\,. \tag{3.5}$$

It is immediate that every monomial of $P(\boldsymbol{x},\boldsymbol{y})$ has degree $2k-1$.

LEMMA 3.1. *The $\boldsymbol{x}$-multilinear monomials in $P(\boldsymbol{x},\boldsymbol{y})$ are in a one-to-one correspondence with the tightly labeled arborescences of size k in H.*

LEMMA 3.2. *Given values for the variables $\boldsymbol{x},\boldsymbol{y}$ as input, the value $P(\boldsymbol{x},\boldsymbol{y})$ can be computed using working memory for $(k-1)n$ field elements and at most $m(k+1)k$ field multiplications.*

Implementation Remark. The values x_u and $y_{\ell,uv}$ are used exactly once during the computation of $P(\boldsymbol{x},\boldsymbol{y})$ in (3.3) and (3.4). The value $y_{\ell,uv}$ is uniform random, so we can obtain it directly from the pseudorandom generator while evaluating (3.4) and without storing the vector $\boldsymbol{y}$ in memory. The value x_u is computed via (2.2).

[5] The generating function can, and in general does, have monomials that are not $\boldsymbol{x}$-multilinear, but contributions from such monomials will cancel out in the constrained multilinear sieve (Lemma 2.1).

3.2 Trading Time for Space.

Compared with the generating function in §3.1, we can further improve the memory footprint of the algorithm from $(k-1)n$ field elements to $2n$ field elements by using Lagrange interpolation to sieve out further unwanted monomials from a loosened generating function. In essence we are employing a further layer of algebrization to save space [38].

Introduce the same domain indeterminates x_u and supporting indeterminates $y_{\ell,uv}$ as in §3.1. Introduce one further indeterminate, z, which enables us to select the desired monomials from a compressed and loosened version of (3.4). In particular, our design will be such that the z-degree of each monomial agrees with the total $\boldsymbol{x}$-degree of the monomial.

Towards this end, let us set the $\ell = 1$ base case for all $u \in V(H)$ by

$$P_{1,u}(\boldsymbol{x},\boldsymbol{y},z) = x_u z\,. \tag{3.6}$$

A compressed and loosened version of (3.4) is now defined for all $u \in V(H)$ and $\ell = 2,3,\ldots,k$ by

$$P_{\ell,u}(\boldsymbol{x},\boldsymbol{y},z) = P_{\ell-1,u}(\boldsymbol{x},\boldsymbol{y},z)\Bigg(1+\sum_{v\in N_H(u)} y_{\ell,uv}P_{\ell-1,v}(\boldsymbol{x},\boldsymbol{y},z)\Bigg)\,. \tag{3.7}$$

Now observe that we can compress memory usage since to get the values for ℓ we only need the values for $\ell-1$, so $2n$ working memory suffices; cf. (3.4). Finally, sum over the vertices to get the generating polynomial

$$P(\boldsymbol{x},\boldsymbol{y},z) = \sum_{u\in V(H)} P_{k,u}(\boldsymbol{x},\boldsymbol{y},z)\,. \tag{3.8}$$

By construction, the monomials in $P(\boldsymbol{x},\boldsymbol{y},z)$ have z-degrees ranging between 1 and 2^{k-1}.

Let us now write $P_j(\boldsymbol{x},\boldsymbol{y})$ for $P(\boldsymbol{x},\boldsymbol{y},z)$ restricted to the $\boldsymbol{x},\boldsymbol{y}$-part of the monomials whose z-degree is exactly j. That is, we have $P(\boldsymbol{x},\boldsymbol{y},z) = \sum_{j=1}^{2^{k-1}} P_j(\boldsymbol{x},\boldsymbol{y})z^j$. In particular, we can recover $P_k(\boldsymbol{x},\boldsymbol{y})$ from $P(\boldsymbol{x},\boldsymbol{y},z)$ by evaluating $P(\boldsymbol{x},\boldsymbol{y},z)$ in $2^{k-1}+1$ distinct points $z = z_i$ and then using univariate Lagrange interpolation to recover the coefficient of the z-monomial of degree k.

An *ℓ-loosely labeled* arborescence is an arborescence A rooted at u whose edges (if any) have been labeled with integers from $2,3,\ldots,\ell$ so that (i) the maximum label of an edge occurs on a unique edge e, (ii) the edge e is incident with the root u, and (iii) deleting e from A produces two $(\ell-1)$-loosely labeled arborescences.

LEMMA 3.3. *The $\boldsymbol{x}$-multilinear monomials in $P_k(\boldsymbol{x},\boldsymbol{y})$ are in a one-to-one correspondence with the k-loosely labeled arborescences of size k in H.*

LEMMA 3.4. *Given values for the variables $\boldsymbol{x}, \boldsymbol{y}, z$ as input, the value $P(\boldsymbol{x}, \boldsymbol{y}, z)$ can be computed using working memory for $2n$ field elements and at most $(2m+n)k$ field multiplications.*

Implementation Remark. At most $2^{k+1}k(m+n)$ field multiplications followed by Lagrange interpolation on $2^{k-1}+1$ points suffice to recover the value $P_k(\boldsymbol{x}, \boldsymbol{y})$. Again we observe that each uniform random value $y_{\ell,uv}$ is used exactly once during the computation of $P_k(\boldsymbol{x}, \boldsymbol{y}, z)$ in (3.7).

3.3 Vectorized Arithmetic. We rely on vectorized finite-field arithmetic to implement constrained multilinear sieves for the generating functions in §3.1 and §3.2. This is (i) to enable parallel execution of arithmetic operations on independent operands using a single instruction stream, utilizing hardware vector extensions, if available, to increase the per-instruction (per-processor-clock) arithmetic throughput, and (ii) to coalesce memory accesses to vectors of consecutive memory words to enable efficient utilization of the memory interface. We find it convenient to use the following terms both in our present exposition and the source code.

A *limb* is a vector consisting of one or more field elements to enable efficient hardware execution of vectorized arithmetic in the sense of (i). For example, on a CPU supporting the AVX2 instruction set extension [28, Vol. 1, §14] it is convenient to bit-pack several field elements into a 256-bit limb that can then be manipulated using 256-bit AVX2 vector registers. For example, if we are working over GF(2^{64}), we can pack four 64-bit field elements into an AVX2 word.

A *line* is a vector consisting of one or more limbs to make efficient use of the memory interface in the sense of (ii). For example, on a microarchitecture with 512-bit cache lines it is convenient to bit-pack limbs into a line so that a 512-bit line results. For example, two 256-bit AVX2 words make a 512-bit line.

Implementation Outline. Observe that the constrained multilinear sieve (2.1) requires us to evaluate the generating function $P(\boldsymbol{x}, \boldsymbol{y})$ at 2^k points $(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{u}^A, \boldsymbol{y})$ indexed by the 2^k subsets $A \subseteq L$ of the k-element set L; cf. (2.2). Serendipitously, the recurrences in §3.1 and §3.2 are *oblivious* to their given input $(\boldsymbol{x}, \boldsymbol{y})$, that is, precisely the same sequence of arithmetic operations will be executed regardless of the input $(\boldsymbol{x}, \boldsymbol{y})$. Thus, we can compute the 2^k values so that the evaluation algorithm considers groups of g values at a time.

In our implementation we replace every scalar operation (that is, operation on individual field elements) in the recurrences §3.1 and §3.2 with a vector operation that works on a line consisting of g independent scalars. Such vectorization increases the working memory of the algorithm for field elements by a multiplicative factor g, but gives running time gain in (i), (ii), and furthermore, (iii) amortizes the cost of memory accesses to the adjacency list representation of the host graph H when evaluating (2.1). In practice we select the line size g to be a power of 2 so that the g scalars together occupy about one 512-bit cache line. If $2^k > g$ we use an outer loop that executes $2^k/g$ vectorized evaluations to compute the sum (2.1).

3.4 Arithmetic on Limbs. Let us now turn to low-level implementation of arithmetic on limbs. We study two field sizes, 64 bits and 8 bits, together with several implementation strategies for each field size. Here we give only a rough overview; further details can be found in the accompanying source code.

Limbs for 64-Bit Scalars. We study two implementations for vectorized arithmetic in GF(2^{64}). The first is a baseline implementation that uses an unrolled multiplication loop operating on 64-bit words. The second implementation relies on the PCLMULQDQ [25] and AVX2 [28, Vol. 1, §14] instruction set extensions to execute multiplication in limbs of four 64-bit words that together occupy a 256-bit AVX2 vector register.

Limbs for 8-Bit Scalars. We study three different implementations for vectorized arithmetic in GF(2^8). First, we prepare a look-up table implementation (we tabulate the discrete logarithm and its inverse for each nonzero field element) to multiply individual field elements. Second, we use *bit-slicing* and software simulation of a (simplified) Mastrovito [39] multiplier to multiply w field elements in parallel in 141 word operations, where w is the available word length in terms of number of bits in the processor registers. The advantage of this approach is that no instruction set extensions are necessary to get $w = 32$ (on a 32-bit microarchitecture) or $w = 64$ (on a 64-bit microarchitecture). Third, we pack 32 field elements into one 256-bit AVX2 register, and then use vector instructions to multiply in parallel.

3.5 Parallelization over Vertices. Our implementation makes use of shared-memory multiprocessing through the OpenMP API [41] via the `omp parallel for` construct with default scheduling.

The constrained sieve relies on essentially three parallel blocks in the body of the outer loop. The first block parallelizes the computation of the values (2.2) over $i \in [n]$ for each line of size g. The second block parallelizes the evaluation of the generating function (§3.1 or §3.2) over the vertices $u \in V(H)$. In particular, the loop over $u \in V(H)$ when evaluating (3.4) (respectively, (3.7)) is parallelized into disjoint equal-sized segments for each fixed ℓ, with synchronization on transition from ℓ to $\ell+1$. The third block evaluates the final sum (3.5) (respectively, (3.8)) in parallel in disjoint equal-sized segments of vertices.

The first two parallel blocks both rely on pseudorandom values (that is, the values of the variables $\boldsymbol{v}$, $\boldsymbol{w}$, and $\boldsymbol{y}$), which we supply with a pseudorandom generator whose state can be fast-forwarded into any position in the pseudorandom sequence.[6]

To obtain further parallel speedup, we parallelize also the parts of the program that parse the input graph and take induced subgraphs for purposes of extraction and listing.[7]

3.6 Hardware and Software Prefetching. Our implementation is structured to benefit from automatic hardware prefetching [27] of values from main memory to in-processor cache memory. Such design considerations include using an adjacency list representation for the host graph H so that the recurrences (3.4) and (3.7) traverse the adjacency lists of vertices in linear order, which triggers automatic prefetching inside an adjacency list and across adjacency lists of consecutive vertices, which are stored as one contiguous array in memory. Similarly, the arrays that store the values $P_{\ell_1,u}(\boldsymbol{x},\boldsymbol{y})$ and $P_{\ell,u}(\boldsymbol{x},\boldsymbol{y},z)$ are organized in memory so that the loops scan the arrays at consecutive cache lines, one cache line at a time.

To implement the recurrences (3.4) and (3.7), in the inner loops we need to access the values $P_{\ell_2,v}(\boldsymbol{x},\boldsymbol{y})$ and $P_{\ell-1,v}(\boldsymbol{x},\boldsymbol{y},z)$ associated with the neighbours $v \in N_H(u)$ of a vertex $u \in V(H)$. For large graphs such indirect, data-dependent (that is, dependent on v where the value v must itself be loaded from the adjacency list of u) accesses to values in main memory present a performance bottleneck because the automatic prefetchers in processor hardware are not able to prefetch indirect accesses from main memory to the processor. To reduce execution stalls in inner loops caused by memory latency, we use software prefetching as follows.

When the inner loop in (3.4) runs over the values $\ell_2 = \ell - \ell_1$ for a fixed $v \in N_H(u)$, we issue a prefetch for the value $P_{\ell_2,v'}(\boldsymbol{x},\boldsymbol{y})$ for the *next* value $v' \in N_H(u)$ in the adjacency list of u.

The inner loop in (3.7) runs over the values $v \in N_H(u)$. When we are currently computing with the value $P_{\ell-1,v}(\boldsymbol{x},\boldsymbol{y},z)$, we issue a prefetch for the value $P_{\ell-1,v'}(\boldsymbol{x},\boldsymbol{y},z)$ so that v' is four positions ahead of v in the adjacency list of u, assuming such a v' is available.

In both cases above we use the temporal prefetch instruction `prefetcht0` [28, Vol. 2B, 4-198].

3.7 Data Structures for Extraction. Our implementation relies on an adjacency list representation for the input graph H and its subgraphs. In particular, to extract a witnesses $W \in \mathcal{F}$, we must be able to recursively form induced subgraphs of H (with relabeling of vertices to consecutive integers) in the adjacency list representation to enable fast execution of the decision algorithm. We accomplish this with the following implementation considerations.

First, we maintain the invariant that the adjacency list of each vertex is sorted in the increasing order. Second, we implement the interval extractor (§2.2) so that each of the at most $k+1$ sets of vertices stored in the queue of the extractor (see [8, Algorithm 1]) is a set of consecutive vertices $i, i+1, \ldots, j$. This implies that each interval query that we execute can be formed by deleting a set of consecutive vertices from the current working graph to produce the query graph.

To construct the adjacency lists of the query graph from those of the working graph, it suffices to run a combination of binary search and linear copying of contiguous segments of adjacency lists. This construction can in fact be executed in parallel using multiple threads by (i) first computing in parallel the degree sequence of the query graph, using the known degrees of the working graph and binary search to determine the contiguous segment that must be cut out from each adjacency list (indeed, the adjacency lists are sorted and we are cutting out a set of consecutive vertices), (ii) computing cumulative degrees with parallel prefix sum, and (iii) constructing the query graph in parallel by copying disjoint segments of the working graph using the cumulative degrees computed in (ii) for positioning the writes. Whenever the decision oracle gives a YES answer on the query graph, the query graph becomes the current working graph and the old working graph is discarded and the sets in the extractor queue relabeled to reflect the new working graph. We also retain information how to embed the new graph back to the old graph so that the eventual witness W that we discover can be mapped back to the original input. This is straightforward be-

[6]We use as a pseudorandom generator the exclusive-or of the states of two 64-bit linear-feedback shift registers (LFSRs), one LFSR whose state is shifted towards more significant bits and another LFSR whose state is shifted towards less significant bits, where the state of both LFSRs can be fast-forwarded with square-and-multiply exponentiation in the polynomial quotient ring defined by the tap polynomial.

[7]Our current implementation carries out no load balancing across the segments of vertices $u \in V(H)$ allocated to each thread. This will result in poor parallel speedup if the total number of incidences (total vertex degree) across the segments is not balanced. We plan to address this implementation caveat in a future version. In the meantime, it is advisable to preprocess the input by randomly relabeling (permuting) the vertices before the input is handed out to the algorithm.

cause by our design each set in the extractor queue is a set of consecutive vertices in the original graph.

3.8 Data Structures for Listing. Our implementation of the recursive listing algorithm (§2.3) uses a software stack to keep track of the recursion to enable scaling to large inputs and a convenient iterator interface to the listing algorithm.

We control the size of the software stack with the invariant that we branch in the order $i = t, t-1, \ldots, 1$ and retain each discovered witness W in the stack only if this is required by subsequent recursive branching, that is, only if we branch with $i \geq 2$ (see §2.3). Indeed, when we branch with $i = 1$ there are no subsequent branches originating from W, and hence the $i = 1$ branch can be rewinded by simply inserting x_1 back to the working set to recover U from U_1. In particular, it suffices to store only x_1 instead of W in the stack to undo the branch. Furthermore, all recursive branches with $i \geq 2$ increase the size of L by at least one, so no path in the recursion tree from the root to a leaf contains more than k such branches. It follows that the size of the software stack is bounded from above by $n + O(k^2)$ words, which enables memory-efficient scaling to large n.

4 Experiments.

This section documents our experiments.

4.1 Hardware and Software. We use three configurations in our experiments.

Small-Memory Configuration. The small-memory configuration is a Fujitsu Esprimo E920 E90+ equipped with a 3.20-GHz Intel Core i5-4570 CPU (Haswell microarchitecture with SSE 4.1/4.2 and AVX 2.0 instruction set extensions, 4 cores, no hyperthreading, 6 MiB last-level cache, two channels to main memory) and 16 GiB of main memory (4×4 GiB DDR3-1600 Hynix HMT451U6AFR8C-PB, non-ECC, unbuffered). The operating system is Linux 3.13.0-35-generic (Ubuntu 4.8.2-19ubuntu1) and the C compiler is `gcc` 4.8.2.

Large-Memory Configuration. The large-memory configuration is a HP ProLiant SL230s Generation8 equipped with two 2.80-GHz Intel Xeon E5-2680 v2 CPUs (Ivy Bridge microarchitecture with AVX instruction set extension, 10 cores, hyperthreading disabled, 25 MiB last-level cache, four channels to main memory per processor) and 256 GiB of main memory (16×16 GiB DDR3-1866 HP 712383-081, ECC, registered). The operating system is Linux 2.6.32-431.23.3.el6.x86_64 and the C compiler is `gcc` 4.4.7.

Fat-Memory Configuration. The fat-memory configuration is a HP Proliant DL580 Generation 7 equipped with four 2.67-GHz Intel Xeon X7542 CPUs (Nehalem microarchitecture, 6 cores, no hyperthreading, 18 MiB last-level cache, one channel to main memory) and 1 TiB of main memory (64×16 GiB DDR3-1066 *unknown vendor/model number*, synchronous). The operating system was Linux 2.6.32-431.23.3.el6.x86_64 and the C compiler was `gcc` 4.4.7.

Our software is implemented with the C programming language (C99), using the OpenMP API [41] for parallelization. The source is compiled on each configuration using the `gcc` options `-march=native` and `-O5` to enable architecture-specific optimizations and instruction set extensions. The running times in our experiments are measured via the OpenMP wall-clock time interface (`omp_get_wtime`) and the memory usage is tracked by placing a wrapper around C standard library memory allocation interface (`malloc`, `free`).

4.2 Input Graphs. We use both synthetic and natural graph topologies in our experiments. We start with synthetic and then proceed to natural topologies.

Regular. For nonnegative integers d, n with dn even, we use the *configuration model* [11, §2.4] to create a random d-regular n-vertex graph. We do *not* reject configurations with loops (1-cycles in the configuration) or couplings (2-cycles).

Power Law. For nonnegative integers D, n, w and $\alpha < 0$, we use the configuration model to create a random n-vertex graph such that the following hold (approximately): (i) the sum of vertex degrees is Dn; (ii) the distribution of vertex degrees is supported at w distinct values with uniform geometric spacing; and (iii) within the support of the degree distribution, the frequency of vertices of degree d is proportional to d^α. That is, we create a nonnegative integer distribution that approximately meets (i,ii,iii) and then draw a uniform random configuration with these degrees. Again we do not reject configurations with loops or couplings.

Clique. For nonnegative integers D, n, we (i) start with an empty n-vertex graph, (ii) select uniformly at random without replacement $t = \lfloor \sqrt{Dn} \rfloor$ out of the n vertices, and (iii) place a clique on these t vertices.

Natural Graphs. For natural graph topologies we rely on the Koblenz Network Collection [34]. The following specific natural graphs from the Koblenz collection are considered in our experiments: Wiki (en) [`wikipedia_link_en`], Wiki (it) [`wikipedia_link_it`], Wiki (fr) [`wikipedia_link_fr`], Wiki (pt) [`wikipedia_link_pt`], DBpedia [`dbpedia-all`], and Berkeley [`web-BerkStan`]; click on the text in brackets to follow the hyperlink. To balance parallel workload (see §3.5), we randomly relabel each natural graph before it is input to the algorithm.

To guarantee a unique match to the motif we use a monochromatic M such that H has exactly k vertices whose color agrees with the motif.[8] For natural graphs, we select one vertex uniformly at random and expand it to k first vertices traversed by depth first search. For synthetic graphs, we select k random vertices and insert a random $(k-1)$-edge path that spans these k vertices.

4.3 Baseline Performance. Tables 1, 2, and 3 document the baseline performance of our experimental configurations. The performance of finite-field arithmetic is measured using our limb implementations (§3.4) to execute 2^{30} independent scalar multiplications. The memory interface is measured by operating on a 4-gibibyte (2^{32} bytes) array of 64-bit words. The large-memory and the fat-memory configurations do not have hardware support for GF(2^{64}) multiplication.

4.4 Vectorization and Parallelism. Our first set of experiments studies the speedup obtainable from the individual techniques in §3. These experiments are presented in Figure 1 and Figure 2.

4.5 Scalability. Our second set of experiments studies the scalability of the algorithm as a function of (a) the number of edges m, and (b) the motif size k. More specifically, we study the *decision time*, that is, the running time of the decision algorithm when it is executed on an input in the adjacency list format.

Figures 3 and 4 display how the decision time and memory consumption grow as we increase the number of edges m geometrically. In both cases we observe the running time grows linearly with little variance between the independent inputs.

Figure 5 displays the decision time as we increase the motif size k in steps of one. We observe exponential scaling with very small variance except at running times substantially below one second.

4.6 Topology Invariance. Our next set of experiments studies the sensitivity of the algorithm to the topology of the host graph. Figure 6 displays the algorithm performance for three different synthetic topologies and several natural topologies. We observe the decision time remains essentially invariant across topologies; only the clique topology is a clear outlier in the positive sense. The decreased running time for the clique topology appears to be caused by the fact that the graphs contain many isolated vertices and comparatively few high-degree vertices, whose recurrence data hence fits in cache memory.

[8]The constrained multilinear sieve (Lemma 2.1) is essentially oblivious to the colors in M and H. Thus, a monochromatic M in a bichromatic H is representative of worst-case performance.

4.7 Extraction versus Decision. Our previous experiments considered the time to decide whether a solution exists. The next set of experiments studies the overhead required to extract a solution after its existence has been discovered.

Figure 7 displays both the time to extract a solution and the decision time. We observe extraction has more variance than decision and that the ratio of the two can be empirically bounded from above by approximately $2k$; cf. [8, Lemma 2.2 and 3.2].

4.8 Beyond One Billion Edges. We also had limited access to a compute node with a one-tebibyte fat-memory configuration (§4.1). To provide data across a more broad spectrum of configurations, we report on miscellaneous experiments (Table 4) for this configuration. In particular we observe from Table 4 that our algorithm implementation scales to graphs with up to 20 billion edges on the fat-memory configuration. A 20-billion edge graph stored as a list of edges takes 40 billion words, which requires 149 GiB (for 32-bit words) or 298 GiB (for 64-bit words) of storage. Our current implementation uses 64-bit indexing.

It is perhaps interesting to observe from Table 4 that it takes a substantial amount of time simply to *generate* a random input with billions of vertices (cf. Column "Input" and Column "Decision" in Table 4).

References

[1] N. Alon, R. Yuster, and U. Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

[2] N. Betzler, M. R. Fellows, C. Komusiewicz, and R. Niedermeier. Parameterized algorithms and hardness results for some graph motif problems. In *Proc. CPM'08*, volume 5029 of *LNCS*, pages 31–43, 2008.

[3] E. Biham. A fast new DES implementation in software. In E. Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.

[4] A. Björklund. Determinant sums for undirected hamiltonicity. In *Proc. FOCS'10*, pages 173–182, 2010.

[5] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Narrow sieves for parameterized paths and packings. *CoRR*, abs/1007.1161, 2010.

[6] A. Björklund, P. Kaski, and Ł. Kowalik. Constrained multilinear detection and generalized graph motifs. *CoRR*, abs/1209.1082, 2012.

[7] A. Björklund, P. Kaski, and L. Kowalik. Probably optimal graph motifs. In N. Portier and T. Wilke,

Table 1: Baseline performance of the small-memory configuration.

Benchmark	Single core	All 4 cores
Finite-field arithmetic		
Scalar multiplication with a $8 \times \mathrm{GF}(2^{64})$ bit-packed line	0.82 GHz	3.1 GHz
Scalar multiplication with a $32 \times \mathrm{GF}(2^{8})$ bit-sliced line	1.7 GHz	6.3 GHz
Memory interface		
Read from linear addresses (consecutive 64-bit words)	16 GiB/s	22 GiB/s
Write to linear addresses (consecutive 64-bit words)	8.7 GiB/s	9.8 GiB/s
Read from random addresses (full cache lines)	2.4 GiB/s	6.8 GiB/s
Read from random addresses (individual 64-bit words)	0.81 GiB/s	1.8 GiB/s

Table 2: Baseline performance of the large-memory configuration.

Benchmark	Single core	All 20 cores
Finite-field arithmetic		
Scalar multiplication with a $8 \times \mathrm{GF}(2^{64})$ bit-packed line	N/A	N/A
Scalar multiplication with a $32 \times \mathrm{GF}(2^{8})$ bit-sliced line	1.6 GHz	26 GHz
Memory interface		
Read from linear addresses (consecutive 64-bit words)	12 GiB/s	42 GiB/s
Write to linear addresses (consecutive 64-bit words)	6.9 GiB/s	16 GiB/s
Read from random addresses (full cache lines)	1.3 GiB/s	16 GiB/s
Read from random addresses (individual 64-bit words)	0.3 GiB/s	4.4 GiB/s

Table 3: Baseline performance of the fat-memory configuration.

Benchmark	Single core	All 24 cores
Finite-field arithmetic		
Scalar multiplication with a $8 \times \mathrm{GF}(2^{64})$ bit-packed line	N/A	N/A
Scalar multiplication with a $32 \times \mathrm{GF}(2^{8})$ bit-sliced line	0.83 GHz	19 GHz
Memory interface		
Read from linear addresses (consecutive 64-bit words)	3.7 GiB/s	18 GiB/s
Write to linear addresses (consecutive 64-bit words)	3.9 GiB/s	9.0 GiB/s
Read from random addresses (full cache lines)	0.74 GiB/s	7.5 GiB/s
Read from random addresses (individual 64-bit words)	0.15 GiB/s	1.9 GiB/s

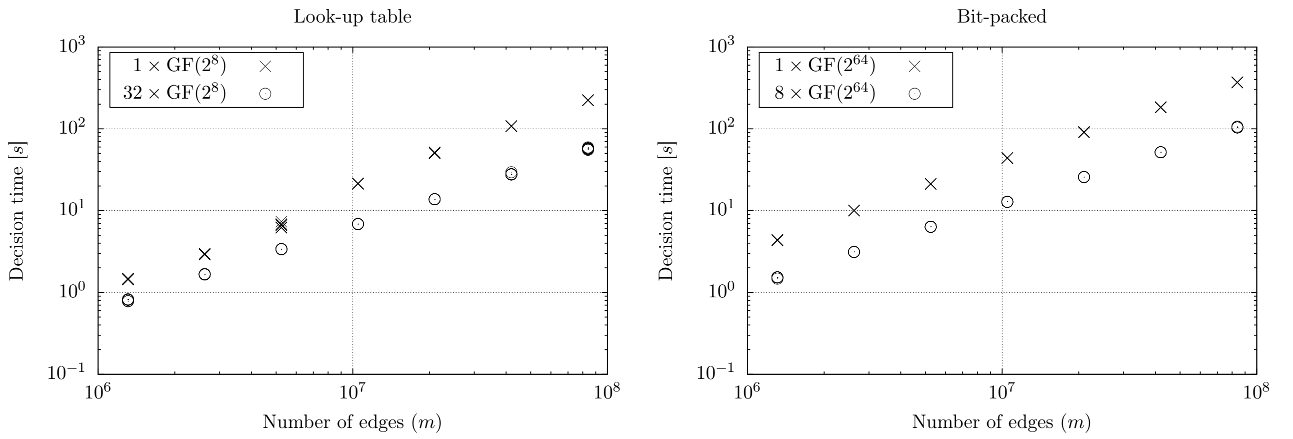


Figure 1: Performance gain from vectorization on the small-memory configuration. We display the decision time of 10 independent d-regular random graphs for each $n = 2^{18}, 2^{19}, \ldots, 2^{24}$ and $d = 10$ fixed. Motif size $k = 5$ fixed. At the largest size $n = 2^{24}$ the speedup from vectorization is at least 3.7 for $\mathrm{GF}(2^8)$ and at least 3.5 for $\mathrm{GF}(2^{64})$.

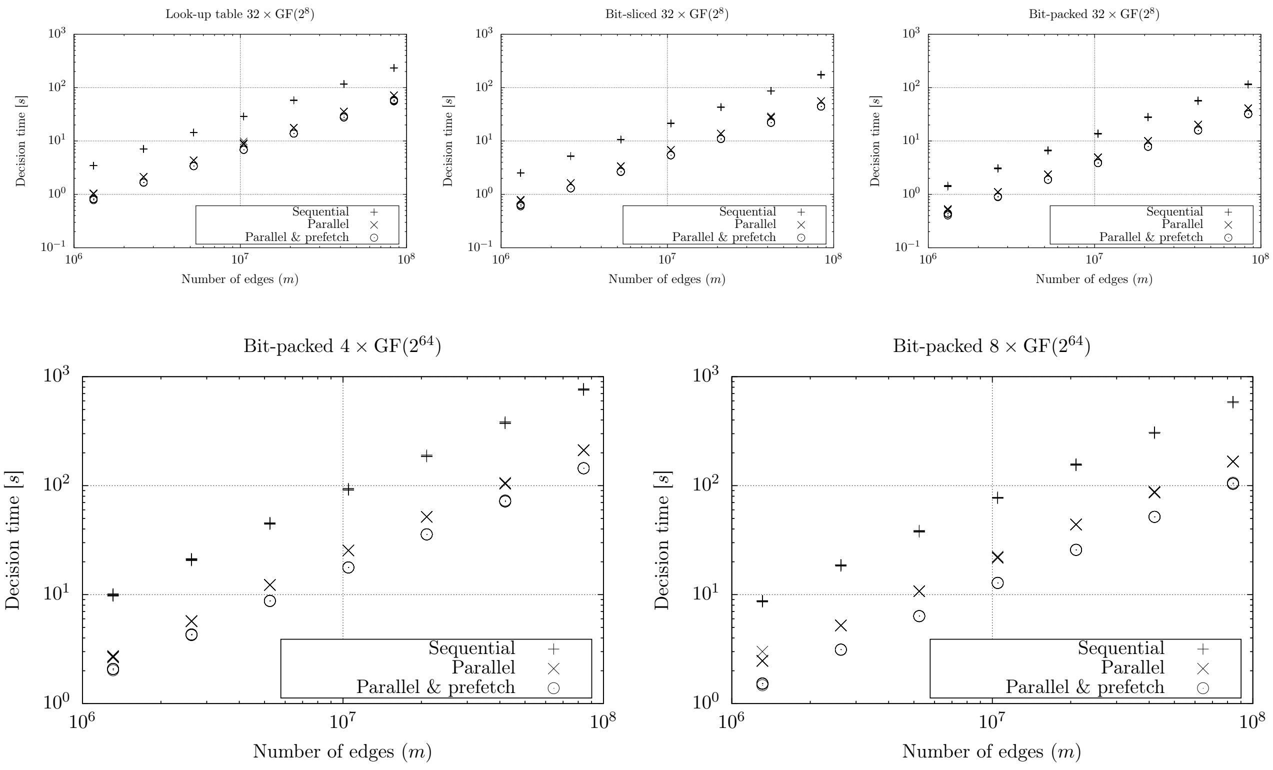


Figure 2: The effect of introducing parallelization and software prefetching on different arithmetic implementations on the small-memory configuration. Top row: three implementations for vectorized $32 \times \mathrm{GF}(2^8)$ arithmetic (look-up table, bit-slicing, bit-packing). Bottom row: a $4 \times \mathrm{GF}(2^{64})$ implementation (left), and a $8 \times \mathrm{GF}(2^{64})$ implementation (right). We display the decision time of 10 independent d-regular random graphs for each $n = 2^{18}, 2^{19}, \ldots, 2^{24}$ and $d = 10$ fixed. Motif size $k = 5$ fixed. At the largest size $n = 2^{24}$ the speedup factor from parallelization and software prefetching is at least 3.5 for bit-packed $32 \times \mathrm{GF}(2^8)$ and at least 5.4 for bit-packed $8 \times \mathrm{GF}(2^{64})$.

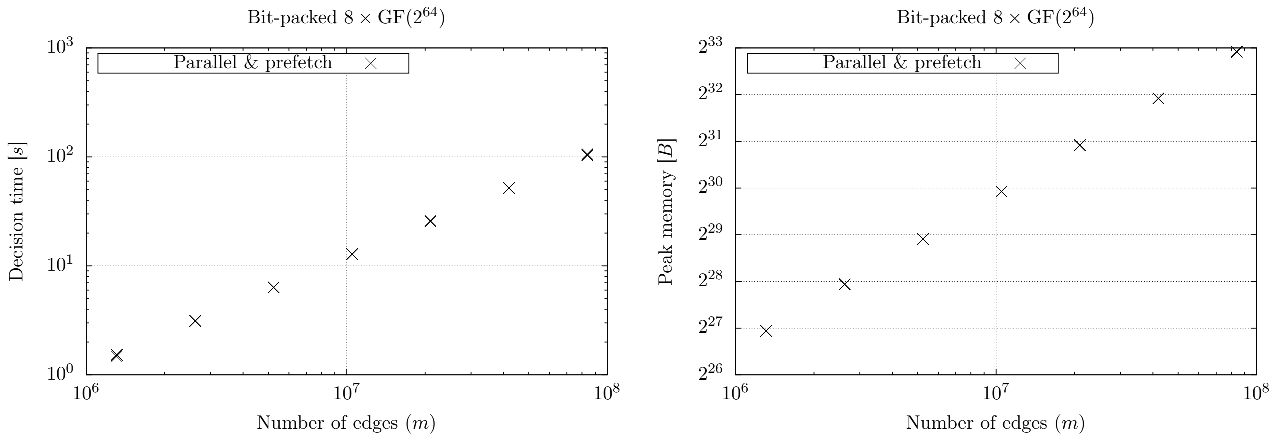


Figure 3: Scalability of the algorithm as the number of edges m increases. We display the decision time (left) and the memory consumption (right) for 10 independent d-regular random graphs for each $n = 2^{18}, 2^{19}, \ldots, 2^{24}$ and $d = 10$ fixed. Motif size $k = 5$ fixed. Small-memory configuration. All axes have logarithmic scale.

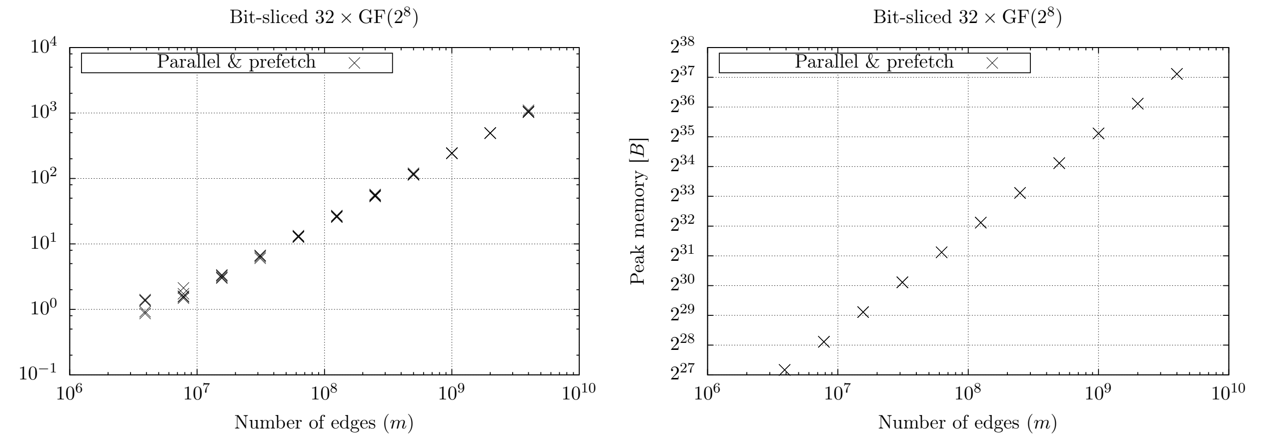


Figure 4: Scalability of the algorithm as the number of edges m increases. We display the decision time (left) and the memory consumption (right) for 5 independent d-regular random graphs for each $n = 4 \cdot 10^8, 2 \cdot 10^8, 1 \cdot 10^8, \ldots, 390625$ with $d = 20$ fixed. The largest value of n is $n = 400$ million. Motif size $k = 5$ fixed. Large-memory configuration. All axes have logarithmic scale.

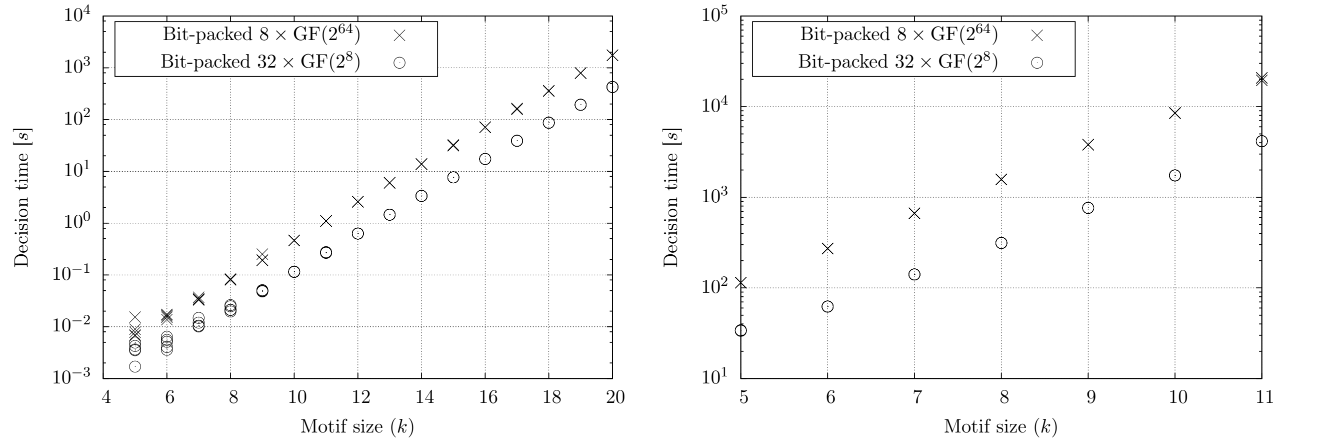


Figure 5: Scalability of the algorithm as the motif size k increases. For each value of k we display the decision time for 5 independent d-regular random graphs with $d = 20$ fixed. Left: $n = 1000$ and $m = 10000$. Right: $n = 10$ million and $m = 100$ million. Small-memory configuration. Vertical axes have logarithmic scale.

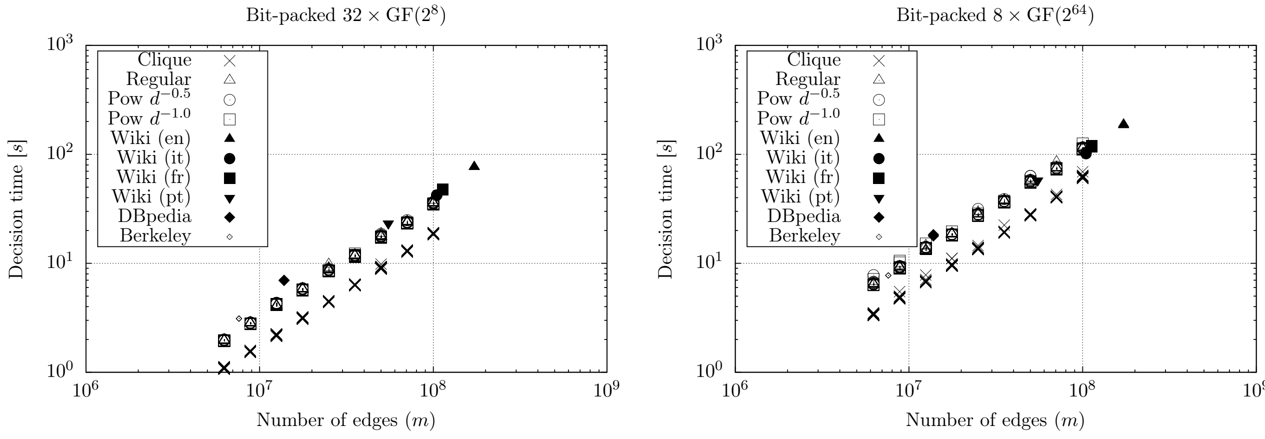


Figure 6: Experiments with different graph topologies. We display the decision time for (a) 50 independent synthetic graphs for each value of n, with n decreasing in multiplicative steps of $1/\sqrt{2}$; and (b) 3 independent random relabelings for each natural graph. *Cliques* start with $n = 10$ million, and $m = 100$ million with $D = 20$ fixed. *Regular graphs* start with $n = 10$ million, and $m = 100$ million with $d = 20$ fixed. *Power-law graphs* start with $n = 10$ million, $D = 20$, $w = 100000$, and both $\alpha = -0.5$ and $\alpha = -1.0$. Motif size $k = 5$ fixed. Small-memory configuration. All axes are logarithmic.

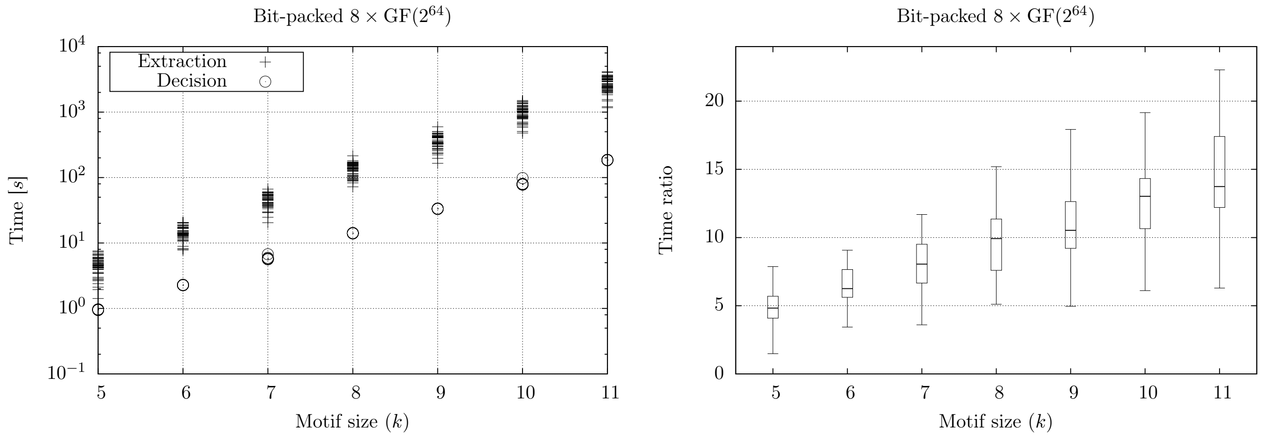


Figure 7: Time to extract one solution versus time to decide the existence of a solution as the motif size k increases. For each value of k we display the extraction time for 50 independent extractions and 10 decisions for d-regular random graphs with $n = 100000$, $m = 1$ million, and $d = 20$ fixed. Left: extraction time and decision time plotted on a logarithmic axis. Right: extraction time divided by the median decision time; plotted are the minimum, 1st quartile, median, 3rd quartile, and maximum ratios so obtained.

Table 4: Miscellaneous experiments with the fat-memory configuration. The graphs are obtained from the d-regular generator with $d = 20$ (top) and $d = 40$ (bottom). Motif size $k = 5$ fixed. The column "Input" gives the time to generate (with an auxiliary generator process) and pipe the input graph (in binary edge list format) into the main memory space of our solver process; "Preprocess" gives the time to transform from an arbitrary-order edge list into an adjacency list; "Decision" gives the decision time of our algorithm; "Total" gives the total execution time of the solver process; and "Peak memory" gives the peak memory usage of the solver process. All reported times are in wall-clock time, using all 24 cores of the fat-memory configuration.

Vertices	Edges	Input	Preprocess	Decision	Total	Peak memory
2 000 000 000	20 000 000 004	2330 s	1937 s	3163 s	7452 s	693 GiB
1 000 000 000	10 000 000 004	1057 s	987 s	1545 s	3599 s	347 GiB
500 000 000	5 000 000 004	492 s	407 s	770 s	1673 s	174 GiB
250 000 000	2 500 000 004	237 s	183 s	376 s	799 s	87 GiB
125 000 000	1 250 000 004	112 s	90 s	182 s	386 s	44 GiB
62 500 000	625 000 004	55 s	43 s	88 s	187 s	22 GiB
1 000 000 000	20 000 000 004	2040 s	1830 s	2915 s	6805 s	623 GiB
500 000 000	10 000 000 004	939 s	816 s	1430 s	3196 s	312 GiB
250 000 000	5 000 000 004	467 s	409 s	704 s	1586 s	156 GiB
125 000 000	2 500 000 004	221 s	182 s	343 s	749 s	78 GiB
62 500 000	1 250 000 004	109 s	88 s	165 s	363 s	39 GiB

editors, *STACS*, volume 20 of *LIPIcs*, pages 20–31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[8] A. Björklund, P. Kaski, and Ł. Kowalik. Fast witness extraction using a decision oracle. In A. S. Schulz and D. Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 149–160. Springer, 2014.

[9] A. Björklund, P. Kaski, Ł. Kowalik, and J. Lauri. `motif-search-v1.0-experimental`: ALENEX15 release. DOI:10.5281/zenodo.12548, 2014. `https://github.com/pkaski/motif-search`.

[10] H. L. Bodlaender and G. F. Italiano, editors. *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, volume 8125 of *Lecture Notes in Computer Science*. Springer, 2013.

[11] B. Bollobás. *Random Graphs*, volume 73 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, second edition, 2001.

[12] F. Cicalese, T. Gagie, E. Giaquinta, E. S. Laber, Z. Lipták, R. Rizzi, and A. I. Tomescu. Indexes for jumbled pattern matching in strings, trees and graphs. In O. Kurland, M. Lewenstein, and E. Porat, editors, *String Processing and Information Retrieval - 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings*, volume 8214 of *Lecture Notes in Computer Science*, pages 56–63. Springer, 2013.

[13] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer. To appear.

[14] R. A. DeMillo and R. J. Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7:193–195, 1978.

[15] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, Heidelberg, fourth edition, 2010.

[16] R. Dondi, G. Fertin, and S. Vialette. Maximum motif problem in vertex-colored graphs. In *Proc. CPM'09*, volume 5577 of *LNCS*, pages 221–235, 2009.

[17] D. Z. Du and F. K. Hwang. Competitive group testing. In *On-line Algorithms (New Brunswick, NJ, 1991)*, volume 7 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 125–134. Amer. Math. Soc., Providence, RI, 1992.

[18] D.-Z. Du and F. K. Hwang. *Combinatorial Group Testing and Its Applications*, volume 12 of *Series on Applied Mathematics*. World Scientific Publishing Co. Inc., 2000.

[19] M. R. Fellows, G. Fertin, D. Hermelin, and S. Vialette. Sharp tractability borderlines for finding connected motifs in vertex-colored graphs. In *Proc. ICALP'07*, volume 4596 of *LNCS*, pages 340–351, 2007.

[20] M. R. Fellows, G. Fertin, D. Hermelin, and S. Vialette. Upper and lower bounds for finding connected motifs in vertex-colored graphs. *J. Comput. Syst. Sci.*, 77(4):799–811, 2011.

[21] R. J. Fisher and H. G. Dietz. Compiling for SIMD within a register. In S. Chatterjee, J. Prins, L. Carter, J. Ferrante, Z. Li, D. C. Sehr, and P. Yew, editors, *Languages and Compilers for Parallel Computing, 11th International Workshop, LCPC'98, Chapel Hill, NC, USA, August 7-9, 1998, Proceedings*, volume 1656 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 1999.

[22] T. Gagie, D. Hermelin, G. M. Landau, and O. Weimann. Binary jumbled pattern matching on

trees and tree-like structures. In Bodlaender and Italiano [10], pages 517–528.

[23] E. Giaquinta and S. Grabowski. New algorithms for binary jumbled pattern matching. *Inf. Process. Lett.*, 113(14-16):538–542, 2013.

[24] S. Gueron and M. Kounavis. Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. *Information Processing Letters*, 110(14):549–553, 2010.

[25] S. Gueron and M. E. Kounavis. *Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode - Rev 2.02*. Intel Corporation, April 2014. [Link].

[26] S. Guillemot and F. Sikora. Finding and counting vertex-colored subtrees. In *Proc. MFCS'10*, volume 6281 of *LNCS*, pages 405–416, 2010.

[27] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, September 2014. [Link].

[28] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*, September 2014. [Link].

[29] D. E. Knuth. *The Art of Computer Programming*, volume 4A, Combinatorial Algorithms, Part 1. Addison-Wesley, Upper Saddle River, NJ, 2011.

[30] T. Kociumaka, J. Radoszewski, and W. Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. In Bodlaender and Italiano [10], pages 625–636.

[31] I. Koutis. Faster algebraic algorithms for path and packing problems. In *Proc. ICALP'08*, volume 5125 of *LNCS*, pages 575–586, 2008.

[32] I. Koutis. Constrained multilinear detection for faster functional motif discovery. *Inf. Process. Lett.*, 112(22):889–892, 2012.

[33] I. Koutis and R. Williams. Limits and applications of group algebras for parameterized problems. In *ICALP (1)*, volume 5555 of *LNCS*, pages 653–664, 2009.

[34] J. Kunegis. KONECT: the Koblenz network collection. In L. Carr, A. H. F. Laender, B. F. Lóscio, I. King, M. Fontoura, D. Vrandecic, L. Aroyo, J. P. M. de Oliveira, F. Lima, and E. Wilde, editors, *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 1343–1350. International World Wide Web Conferences Steering Committee / ACM, 2013. .

[35] V. Lacroix, C. G. Fernandes, and M.-F. Sagot. Motif search in graphs: Application to metabolic networks. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(4):360–368, 2006.

[36] L. Lamport. Multiple byte processing with full-word instructions. *Commun. ACM*, 18(8):471–475, 1975.

[37] E. L. Lawler. A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem. *Management Sci.*, 18:401–405, 1971/72.

[38] D. Lokshtanov and J. Nederlof. Saving space by algebraization. In L. J. Schulman, editor, *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 321–330. ACM, 2010.

[39] E. Mastrovito. *VLSI Architectures for Computations in Galois Fields*. PhD thesis, Department of Electrical Engineering, Linköping University, 1991.

[40] J. Nederlof. Fast polynomial-space algorithms using Möbius inversion: Improving on Steiner tree and related problems. In *Proc. ICALP'09*, volume 5555 of *LNCS*, pages 713–725, 2009.

[41] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.0 – July 2013*. [Link].

[42] R. Y. Pinter, H. Shachnai, and M. Zehavi. Deterministic parameterized algorithms for the graph motif problem. In E. Csuhaj-Varjú, M. Dietzfelbinger, and Z. Ésik, editors, *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II*, volume 8635 of *Lecture Notes in Computer Science*, pages 589–600. Springer, 2014.

[43] R. Y. Pinter and M. Zehavi. Partial information network queries. In T. Lecroq and L. Mouchard, editors, *Combinatorial Algorithms - 24th International Workshop, IWOCA 2013, Rouen, France, July 10-12, 2013, Revised Selected Papers*, volume 8288 of *Lecture Notes in Computer Science*, pages 362–375. Springer, 2013.

[44] R. Y. Pinter and M. Zehavi. Algorithms for topology-free and alignment network queries. *J. Discrete Algorithms*, 27:29–53, 2014.

[45] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi. Efficient rijndael encryption implementation with composite field arithmetic. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2001.

[46] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[47] H. S. Warren. *Hacker's Delight*. Addison-Wesley, Upper Saddle River, NJ, second edition, 2012.

[48] R. Williams. Finding paths of length k in $O^*(2^k)$ time. *Inf. Process. Lett.*, 109(6):315–318, 2009.

[49] M. Zehavi. Parameterized algorithms for module motif. In K. Chatterjee and J. Sgall, editors, *Mathematical Foundations of Computer Science 2013 - 38th International Symposium, MFCS 2013, Klosterneuburg, Austria, August 26-30, 2013. Proceedings*, volume 8087 of *Lecture Notes in Computer Science*, pages 825–836. Springer, 2013.

[50] R. Zippel. Probabilistic algorithms for sparse polynomials. In *Proc. International Symposium on Symbolic and Algebraic Computation*, volume 72 of *LNCS*, pages 216–226, 1979.