Finding the Minimum-Weight k-Path[⋆]

Avinatan Hassidim[⋆⋆], Orgad Keller, Moshe Lewenstein[⋆⋆⋆], and Liam Roditty

Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel
{avinatan,kellero,moshe,liamr}@cs.biu.ac.il

Abstract. Given a weighted n-vertex graph G with integer edge-weights taken from a range $[-M, M]$, we show that the minimum-weight simple path visiting k vertices can be found in time $\tilde{O}(2^k \text{poly}(k) M n^2) = O^*(2^k M)$. If the weights are reals in $[1, M]$, we provide a $(1+\varepsilon)$-approximation which has a running time of $\tilde{O}(2^k \text{poly}(k) n^2 (\log\log M + 1/\varepsilon))$. For the more general problem of k-tree, in which we wish to find a minimum-weight copy of a k-node tree T in a given weighted graph G, under the same restrictions on edge weights respectively, we give an exact solution of running time $\tilde{O}(2^k \text{poly}(k) M n^3)$ and a $(1 + \varepsilon)$-approximate solution of running time $\tilde{O}(2^k \text{poly}(k) n^3 (\log\log M + 1/\varepsilon))$. All of the above algorithms are randomized with a polynomially-small error probability.

1 Introduction

Given an n-vertex graph $G = (V, E)$ and a parameter k, in the *k-path* problem we wish to find a simple path in G consisting of k vertices, if such exists. The k-path problem can be easily shown to be NP-complete: when $k = n$, it is exactly the Hamiltonian path problem. While a trivial $O^*(n^k)$ solution[1] is to try all $\binom{n}{k}$ combinations of k vertices, better can be obtained; Monien was the first to show an improvement [14], with an $O^*(k!)$ algorithm. In their seminal result, Alon, Yuster, and Zwick [2] introduced the *color-coding* technique. They used it to present a randomized $O^*((2e)^k)$ algorithm for this problem, which can be derandomized, replacing the $2e$ term with a large constant. Their result thus shows that the LOGPATH problem of determining whether a graph contains a path of length $\log n$ can be solved in polynomial time. Later, two independent results [11, 6] presented randomized $O^*(4^k)$ algorithms, again with larger constants when derandomized, having running times of $O^*(16^k)$ [11] and $O^*(12.5^k)$ [6].

While these results were combinatorial in nature, the next improvements used algebraic techniques: Koutis [12] presented an algorithm solving the problem in $O^*(2.83^k)$ time. His method was perfected by Williams [16], reducing the

⋆ A preliminary version of this paper has been presented in [10].

⋆⋆ Research supported by ISF grant 1241/12 and by GIF Young.

⋆⋆⋆ Research supported by BSF grant 2010437, a Google Research Award and GIF grant 1147/2011.

[1] Here and throughout, the O^* notation discards all factors that are polynomial in n, k, and $\log M$ from the running time. Similarly, the $\tilde{O}$ expressions discard polylogarithmic factors.

running time to $O^*(2^k)$. This had somewhat closed the gap between the k-path problem and the best method known for the specific case of finding a Hamiltonian path in a directed graph, which is $O^*(2^n)$ (though the latter is combinatorial in nature). For undirected graphs, recent results presented $O^*(1.657^n)$ [3] and later $O^*(1.657^k)$ [4, 1] running times for Hamiltonian path and k-path, respectively.

It is worthwhile to focus on Koutis' and Williams' techniques, as they are the basis to this paper. They reduce k-path and other problems to the problem of determining whether a given n-variate polynomial contains a k-multilinear-monomial (that is, a term which is the multiplication of k distinct variables) in its sum-product expansion. The problem is then solved by (roughly) evaluating this polynomial over random values taken from an adequate choice of an algebraic structure. In a later result [13] they both show that, in the evaluation framework they use, their technique for finding a k-multilinear-monomial is essentially optimal, as any choice of an algebraic structure for the polynomial evaluation would require that the elements in this structure have an $\Omega(2^k/k)$-sized representation.

One of the most natural generalizations coming to mind, is the *minimum-weight k-path* problem: in this scenario, the graph edges are weighted and we wish to find a k-path having minimum weight in the graph. In [16] this was referred to as the *short cheap tour* problem and mentioned that while the $O^*(4^k)$ methods can be easily extended to accommodate for this version, the algebraic methods do not seem to support such an extension, and left this as an open problem. We solve this problem for the specific case in which the edge weights are integers in the range $[-M, M]$, incurring a running time which also has a superlinear dependency on M. If the weights are reals in $[1, M]$ (or can be normalized to this range, as is the case if they are in the range $[\ell, h]$ for $0 < \ell < h$), we provide a $(1+\varepsilon)$-approximation which reduces this dependency to $\log\log M$. Notice that by this we conform to the important line of research in recent years, of discussing variants of distance problems in which edge-weights are integers taken from a bounded range, see e.g., [18, 7].

Another problem that generalizes k-path is presented in [13]: in the k-tree problem, given an n-vertex graph G and a k-node tree T, find a copy of T in G. For a similar generalization of this problem to *minimum-weight k-tree*, and under similar restrictions on the edge weights, we show similar exact and approximate results.

Paper Organization. In Section 2 we provide some preliminaries. Then, in Section 3 we show how to find the minimum degree coefficient of a polynomial containing a multilinear monomial in the other variables, which will serve as a subroutine in our algorithms. In Section 4, we first present an $\tilde{O}(2^k \text{poly}(k) M n^2)$ algorithm for computing the weight of the minimum-weight k-path when edge weights are integers in $[-M, M]$. In Section 5, we show how to find the path itself, incurring an $O(k \cdot \text{poly} \log n)$ multiplicative overhead for the above algorithm. In Section 6, for the case of real edge-weights in $[1, M]$, we provide a $(1+\varepsilon)$-approximation algorithm that reduces the dependency on M to $\log\log M$, by using a technique of careful adaptive scaling of the edge weights. The overall running time of this algorithm is $\tilde{O}(2^k \text{poly}(k) n^2 (\log\log M + 1/\varepsilon))$. In Section 7 we turn to the k-tree problem, and show similar results: we present an $\tilde{O}(2^k \text{poly}(k)Mn^3)$ algorithm for finding the minimum-weight k-tree when edge weights are integers in $[-M, M]$, and for the case the edge-weights are reals in $[1, M]$, provide a $(1+\varepsilon)$-approximation algorithm having running time $\tilde{O}(2^k \text{poly}(k)n^3(\log\log M + 1/\varepsilon))$.

2 Preliminaries

We follow Williams' notation [16]. Let $\mathbb{F}$ be a field and G be a multiplicative group. The group algebra $\mathbb{F}[G]$ is defined over the set of elements of the form

$$\sum_{g\in G} a_g g \tag{1}$$

where $a_g \in \mathbb{F}$ for all $g \in G$, i.e., on the set of sums of elements from G with coefficients from $\mathbb{F}$. Addition is computed component-wise as

$$\sum_{g\in G} a_g g + \sum_{g\in G} b_g g = \sum_{g\in G} (a_g + b_g) g \ , \tag{2}$$

multiplication is defined in the form of a convolution:

$$\left(\sum_{g\in G} a_g g\right)\left(\sum_{g\in G} b_g g\right) = \sum_{g,h\in G} a_g b_h gh = \sum_{g\in G}\left(\sum_{h\in G} a_h b_{h^{-1}g}\right) g \ , \tag{3}$$

(since G is a multiplicative group, the expression $h^{-1}g$ here replaces the expression of the type $g-h$ which is usually found in a convolution definition) and multiplication by a scalar $c \in \mathbb{F}$ as

$$c\left(\sum_{g\in G} a_g g\right) = \sum_{g\in G} c a_g g \ . \tag{4}$$

Let $0_{\mathbb{F}}, 1_{\mathbb{F}}$ be the addition and multiplication identities of $\mathbb{F}$, respectively. Let 1_G be the identity of G. It is easy to verify that $\mathbb{F}[G]$ is a ring where the addition identity element $0_{\mathbb{F}[G]} = \sum_{g\in G} 0_{\mathbb{F}} \cdot g$ is the element having all coefficients taken as $0_{\mathbb{F}}$, and the multiplication identity element $1_{\mathbb{F}[G]} = 1_{\mathbb{F}} \cdot 1_G = 1_G$. For ease of notation, hereafter 0 and 1 will denote $0_{\mathbb{F}[G]}$ and $1_{\mathbb{F}[G]}$, respectively.

Let z be a symbolic variable. Our computations are done on the set $(\mathbb{F}[G])[z]$ of univariate polynomials on z with coefficients in $\mathbb{F}[G]$. Notice that the set of polynomials with coefficients in a ring is a ring by itself. Throughout the paper, when referring to a *minimum-degree* coefficient or monomial in a polynomial from $(\mathbb{F}[G])[z]$, we always mean minimum degree w.r.t. z.

For our algorithm, we follow Williams and choose G to be $\mathbb{Z}_2^k$ (i.e., the set of binary vectors of dimension k) with multiplication between elements of $\mathbb{Z}_2^k$ defined as entry-wise addition modulo 2. It follows that 1_G is the k-dimensional all-zeros vector. Notice that for all $u, v \in \mathbb{Z}_2^k$, $u \cdot v = 1_G$ iff $u = v$. We also choose $\mathbb{F} = \mathrm{GF}(2^\ell)$ for $\ell = \log k + 3$. Notice that since $\mathbb{F} = \mathrm{GF}(2^\ell)$ has characteristic 2, it holds that for all $c \in \mathbb{F}$, $c + c = 0_{\mathbb{F}}$, and therefore that for all $v \in \mathbb{F}[G]$, $v + v = 0$.

3 Finding the Minimum Degree Coefficient Containing a Multilinear Monomial

Let $P(x_1, \ldots, x_n, z)$ be a polynomial where $x_1, \ldots, x_n$ are variables to be determined, but z is an indeterminate variable, and where all monomials in the sum-product expansion of P are of the form $x_{i_1} \cdots x_{i_k} \cdot z^d$. That is, all monomials are the product of k (not necessarily distinct) x-variables, finally multiplied by the term z^d for some integer d. P can be represented as an arithmetic circuit comprised of $+$-gates and $\times$-gates for which the inputs and the output are polynomials in the indeterminate z. We let the $+$-gates be of unbounded fan-in and the $\times$-gates be of fan-in 2. For the time being, we do not care about the exact implementation of these gates. We let $s(n)$ be the overall number of gates, and let $\mathrm{coeff}_z^d P(x_1, \ldots, x_n, z)$ be the d-th degree coefficient, w.r.t. z, of P. We claim the following:

Theorem 1. *Let $P(x_1, \ldots, x_n, z)$ be a polynomial represented as above, and let $d' = \min\{d \mid \mathrm{coeff}_z^d P(x_1, \ldots, x_n, z)$ contains a multilinear term$\}$ (if such exists). Then there is a randomized algorithms that computes d' with probability at least $1/5$, in time $\tilde{O}(2^k \mathrm{poly}(k) M \cdot s(n))$.*

3.1 Algorithm

Given the circuit representing P, define a new polynomial P' as follows: P' is identical to P except that the output of each multiplication gate g_i is multiplied by a random value $y_i \in \mathbb{F}$. Randomly pick n vectors $v_1, \ldots, v_n$ from $G = \mathbb{Z}_2^k$. Now compute the polynomial $P''(z) = P'(1+v_1, \ldots, 1+v_n, z)$. Let $\mathrm{coeff}_z^d P''(z)$ be the d-th degree term coefficient of $P''(z)$, and let $d' = \min\{d \mid \mathrm{coeff}_z^d P'(z)$ is not $0\}$ (if such exists). If d' exists, return it. Otherwise output "no multilinear term exists in P".

3.2 Proof of Correctness

Observe the sum-product expansion of P', and let $t = x(t) \cdot y(t) \cdot z^d$ be some monomial in it, where $x(t) = x_{i_1} \cdots x_{i_k}$ is a product of k (not necessarily distinct) x-variables, and $y(t)$ is a product of t y-variables (which are necessarily distinct by the definition of P'). For a monomial t, notice that if $x(t)$ is multilinear, it is square-free, since each variable x_i appears in it at most once. On the other hand, if $x(t)$ is non-multilinear, then it must contain some square x_j^2. In order to prove the algorithm correct, we need to show that w.h.p., (a) non-multilinear monomials vanish, (b) multilinear monomials do not vanish by their evaluation, and that (c) multilinear monomials are not eliminated when they are summed with other (same-degree) products. These notions are captured by the following propositions, which are similar to the ones in [16].

Proposition 1. *If $x(t)$ is non-multilinear, t vanishes.*

Proof. Assume $x(t)$ contains some square x_j^2. Since x_j was assigned with $1 + v_j$, it holds that $x_j^2 = (1 + v_j)^2 = 1_G^2 + 2 \cdot 1_G \cdot v_j + v_j^2 = 1_G + 2 \cdot 1_G \cdot v_j + 1_G = 2 \cdot 1_G + 2 \cdot 1_G \cdot v_j = 0 + 0 = 0$ where the third equality holds since for all $v \in G$, $v \cdot v = 1_G$, and the fifth equality holds since $\mathbb{F}$ has characteristic 2 and therefore for all $c \in \mathbb{F}$, $2c = 0_{\mathbb{F}}$. □

Let $J = \sum_{v \in G} v$ be the sum of all vectors from $G = \mathbb{Z}_2^k$ (addition here is the addition of $\mathbb{F}[G]$).

Proposition 2. *If $x(t)$ is multilinear, then if the vectors $v_{i_1}, \ldots, v_{i_k} \in \mathbb{Z}_2^k$ are linearly independent w.r.t. entry-wise addition modulo 2, then $x(t) = J$.*

Proof. If the k vectors $v_{i_1}, \ldots, v_{i_k} \in \mathbb{Z}_2^k$ are linearly-independent, then they form a basis $B = \{v_{i_1}, \ldots, v_{i_k}\}$ for $\mathbb{Z}_2^k$. Notice that $x(t) = \prod_{c=1}^{k}(1 + v_{i_c}) = \sum_{S \subseteq B} \prod_{v \in S} v$, i.e., $x(t)$ is the sum of every possible combination of vectors from B, multiplied together. Hence, the sum covers all vectors in the span of B, that is, $\sum_{S \subseteq B} \prod_{v \in S} v = \sum_{v \in \mathrm{span}(B)} v = \sum_{v \in \mathbb{Z}_2^k} v = J$. □

Corollary 1. *If $x(t)$ is multilinear, then with probability at least 0.28 t does not vanish.*

Proof. The values $v_{i_1}, \ldots, v_{i_k} \in \mathbb{Z}_2^k$ were chosen randomly and independently. It is known that a random $k \times k$ matrix of values from $\mathbb{Z}_2$ has full rank with probability at least 0.28 [5]. □

We have shown that with at least constant probability, multilinear terms do not vanish when they are assigned values as described. However, it still might happen that such multilinear terms will get eliminated when they are summed up with other multilinear terms. The next two propositions show that this can happen with at most constant probability.

Proposition 3. *If the variables $v_{i_1}, \ldots, v_{i_k} \in \mathbb{Z}_2^k$ are linearly dependent w.r.t. entry-wise addition modulo 2, then t vanishes.*

Proof. Recall that $x(t) = \sum_{S \subseteq \{v_{i_1}, \ldots, v_{i_k}\}} \prod_{v \in S} v$. If the k vectors $v_{i_1}, \ldots, v_{i_k} \in \mathbb{Z}_2^k$ are linearly-dependent, then there exists a set $T \subseteq \{v_{i_1}, \ldots, v_{i_k}\}$ such that $\prod_{v \in T} = 1_G$. Since, as mentioned, for $u, v \in G$ it holds that $uv = 1_G$ iff $u = v$, we get that for all $S' \subseteq T$, $\prod_{v \in S'} v = \prod_{v \in T \setminus S'} v$ and therefore for a subset $S \subseteq \{v_{i_1}, \ldots, v_{i_k}\}$, $\prod_{v \in S \cap T} v = \prod_{v \in T \setminus S} v$. Define the mapping $f(S) = (S \setminus T) \cup (T \setminus S)$. Notice that $S \neq f(S)$ and that as $f(f(S)) = S$, this mapping is bijective. It follows that a specific value r can only occur an even number of times in the summation, because if it occurs in the summation as $r = \prod_{v \in S} v$, then it must also occur again as $r = \prod_{v \in (S \setminus T) \cup (T \setminus S)} v$. Since $2r = 0_{\mathbb{F}} \cdot r$ as $\mathbb{F}$ has characteristic 2, all terms are eliminated in the summation. □

Recall that $P'(x_1,\ldots,x_n,z)$ is a polynomial in z and therefore can be viewed as

$$P(x_1,\ldots,x_n,z) = \sum_{d=0}^{kM} \sum_{\substack{t \\ \deg_z(t)=d}} y(t)\cdot x(t)\cdot z^d \ . \tag{5}$$

Let d' be the minimum degree (w.r.t. z) in P' of a term t for which $x(t)$ is multilinear and let

$$\mathrm{coeff}_z^{d'} P'(x_1,\ldots,x_n,z) = \sum_{\substack{t \\ \deg_z(t)=d'}} y(t)\cdot x(t) \tag{6}$$

be its corresponding coefficient. Our goal is to show that with at least constant probability, $\mathrm{coeff}_z^{d'} P'$ does not vanish when it is evaluated.

Proposition 4. $\mathrm{coeff}_z^{d'} P''(z)$ *is not zero with probability at least* $1/5$.

Proof. By Propositions 2 and 3, it holds that

$$\mathrm{coeff}_z^{d'} P''(z) = J \cdot \sum_{\substack{t \\ \deg_z(t)=d' \text{ and } t \text{ survived}}} y(t) \ .$$

Let

$$Q = \sum_{\substack{t \\ \deg_z(t)=d' \text{ and } t \text{ survived}}} y(t) \ .$$

Q is a degree-k polynomial in the variables y_j. With probability at least 0.28 at least one minimum-weight term t had survived and therefore Q is not identically zero. In this case, by the Schwartz-Zippel lemma [15, 17, 8], when assigning random values from $\mathrm{GF}(2^\ell)$ to the y_j variables, Q evaluates to zero with probability at most $k/2^\ell = 1/8$. Therefore Q (and hence, $\mathrm{coeff}_z^{d'} P'(z)$) does not vanish with probability at least $0.28 \cdot 7/8 > 1/5$. □

3.3 Running Time Analysis

The running time of the algorithm is dominated by the evaluation of the $s(n)$ gates in P''s representation, where each gate performs an arithmetic operation over the polynomial ring $(\mathbb{F}[G])[z]$. Therefore, we need to account for the the cost of each such operation. Notice that for any arithmetic operation in $(\mathbb{F}[G])[z]$ performed by our algorithm, the maximum degree (w.r.t. z) of the operand polynomials and resulting polynomial, is at most kM. We can therefore focus on the set R of polynomials in $(\mathbb{F}[G])[z]$ with degree at most kM. By treating the polynomials in R as periodic with period kM (since there will be no carry or overflow to greater degrees), R continues to be a ring. Let T be the upper-bound on the time required for an arithmetic operation in R; trivially, $T = \Omega(2^k \cdot kM \log|\mathbb{F}|)$. It follows that the algorithm requires $O(s(n)\cdot T)$ time, and it remains to compute T.

Addition Gate. Addition of two polynomials can be easily done component-wise in time $O(kM \cdot 2^k \cdot \log|\mathbb{F}|) = O(2^k \mathrm{poly}(k)M)$.

Multiplication Gate. Multiplication is trickier and is done by employing a multidimensional fast Fourier transform-type approach.[2] We now describe the multiplication process in more detail.

The multiplication process will be easier to describe on the ring $\mathbb{F}[\mathbb{Z}_2^k \times [kM]]$ which is isomorphic to R, as will be shown immediately. Given a vector $v = (v_1, \ldots, v_k) \in \mathbb{Z}_2^k$ and an integer $d \in [kM]$, let $(v; d)$ denote the vector $(v_1, \ldots, v_k, d) \in \mathbb{Z}_2^k \times [kM]$. A polynomial $p \in R$ can be uniquely described as a sum $\sum_{v,d} a_{(v;d)} \cdot (v; d)$ of at most $N = 2^k kM$ summands, where each $a_{(v;d)} \in \mathbb{F}$ is the coefficient of v appearing in $\mathrm{coeff}_z^d p$ (i.e., if $\mathrm{coeff}_z^d p = \sum_{v \in G} b_v v$, then $a_{(v;d)} = b_v$). Our definition of multiplication over $G = \mathbb{Z}_2^k$ can be naturally extended to $\mathbb{Z}_2^k \times [kM]$: multiplication still corresponds to entry-wise addition, only that now addition is done modulo 2 for dimensions $1, \ldots, k$ and modulo kM for dimension $k+1$. With that in mind, our definitions of addition, multiplication, and identity elements for R are extended appropriately, thus forming the ring $\mathbb{F}[\mathbb{Z}_2^k \times [kM]]$. The bottom line is that now any $p \in R$ can be viewed as a sum of elements with coefficients taken from a multidimensional array indexed by values from $\mathbb{Z}_2^k \times [kM]$ and that multiplication is still a convolution, an important fact to be used later.

Moving to $\mathbb{F} = \mathrm{GF}(2^\ell)$, being a finite field, all elements in $\mathbb{F}$ can be represented in the usual manner as a degree-ℓ polynomials with coefficients in $\mathbb{Z}_2 = \mathrm{GF}(2)$ and operations that are done modulo some predefined irreducible polynomial of degree ℓ (this irreducible polynomial can even be found naïvely as $\ell = \log k + 3$). For the purpose of using FFT, we treat polynomials in $\mathbb{Z}_2[x]$ as if they were actually in $\mathbb{C}[x]$, i.e., the set of univariate polynomials over the complex numbers. At the end of the multiplication process, we will appropriately convert polynomials in $\mathbb{C}[x]$ back to $\mathrm{GF}(2^\ell)$ as will be described shortly.

By the above arguments, given two polynomials $p, q \in R$ to be multiplied, they can be taken as the sums $\sum_{v,d} p_{(v;d)} \cdot (v; d)$ and $\sum_{v,d} q_{(v;d)} \cdot (v; d)$, respectively, where $p_{(v;d)}, q_{(v;d)} \in \mathbb{C}[x]$ for each $v \in \mathbb{Z}_2^k$ and $d \in [kM]$. As the multiplication corresponds to a convolution, by the convolution theorem, it holds that $p * q = \mathrm{DFT}^{-1}(\mathrm{DFT}(p) \cdot \mathrm{DFT}(q))$, where $*$ denotes a convolution, $\cdot$ denotes pointwise multiplication, and DFT denotes the $(k+1)$-dimensional discrete Fourier transform for values indexed by vectors of type $(v_1, \ldots, v_k, d) \in \mathbb{Z}_2^k \times [kM]$. Let $D(\ell)$ denote the time required for an arithmetic operation on degree-ℓ polynomials in $\mathbb{C}[x]$—including converting them back to $\mathrm{GF}(2^\ell)$ by division by an irreducible polynomial—and notice that $D(\ell) = O(\ell^2) = O(\mathrm{poly}\log k)$ as multiplication and division here are quadratic by nature. Then the above DFT operations can be computed efficiently in time $O(N \log N \cdot D(\ell)) = \tilde{O}(2^k k^2 M)$ by using the multidimensional FFT algorithm. Once we have computed $\mathrm{DFT}(p)$

[2] Here, as opposed to Williams [16], the Walsh-Hadamard transform is not an adequate choice anymore due to the existence of the variable z which can have a degree up to kM.

and $\mathrm{DFT}(q)$, thus obtaining for each of them N values in $\mathbb{C}[x]$ (indexed as well by vectors in $\mathbb{Z}_2^k \times [kM]$), we point-wise multiply them, obtaining an array $w = \mathrm{DFT}(p) \cdot \mathrm{DFT}(q)$, and compute $\mathrm{DFT}^{-1}(w)$, again by using FFT on multidimensional coefficients in $\mathbb{C}[x]$. Finally, we reduce $\mathbb{C}[x]$ terms (which are actually in $\mathbb{Z}[x]$, as convolution over integer values returns integer values) by dividing them by the irreducible polynomial used before and the appropriate modulo operations.

We conclude that multiplication of polynomials in R can be performed in time $\tilde{O}(2^k \mathrm{poly}(k) M)$, and therefore the overall running time of the algorithm is $\tilde{O}(2^k \mathrm{poly}(k) M \cdot s(n))$.

4 Finding the Weight of the Minimum-Weight *k*-Path

Given a weighted, directed or undirected graph $H = (V, E, w)$ on the vertex-set $V = \{1, \ldots, n\}$, with integer edge-weights in $[-M, M]$, we first show how to compute the weight of the minimum-weight k-path with high probability, by reducing the problem to finding the minimum-degree coefficient containing a multilinear monomial in a polynomial $P(x_1, \ldots, x_n, z)$.

We can assume that the edge weights are actually in $[0, M]$, otherwise we re-define $w(i,j) \leftarrow w(i,j) + M$ for each $(i,j) \in E$ and then $M \leftarrow 2M$: as this process incurs a penalty of $(k-1)M$ for each k-path, it maintains the order relation on k-path weights. Define a *k-walk* to be a walk in the graph comprised of k (not necessarily distinct) vertices, and let $I = \langle i_1, \ldots, i_k \rangle$ be some arbitrary k-walk in H. With a slight abuse of notation, we will also use I to denote the set of edges participating in the walk.

We define a collection $\{B_c\}_{c=1}^{k-1}$ of polynomial matrices B_c as follows:

$$B_c[i,j] = \begin{cases} x_i \cdot z^{w(i,j)} & \text{if } (i,j) \in E, \\ 0 & \text{if } (i,j) \notin E; \end{cases} \tag{7}$$

where each x_i is a variable that corresponds to vertex i and z is a symbolic variable. We define the polynomial P as follows: $P(x_1, \ldots, x_n, z) = \mathbf{1} \cdot B_1 \cdots B_{k-1} \cdot \boldsymbol{x}$, where $\mathbf{1}$ is the n-dimensional all-ones vector and $\boldsymbol{x}$ is the vector $(x_1, \ldots, x_n)$. Re-writing P as its sum-product expansion we get:

$$P(x_1, \ldots, x_n, z) = \sum_{\substack{I \\ I = \langle i_1, \ldots, i_k \rangle \text{ is a walk in } H}} \left(\prod_{c=1}^{k-1} B_c[i_c, i_{c+1}] \right) x_{i_k} \ , \tag{8}$$

that is, P is an aggregate sum over all k-walks in H, where each walk $I = \langle i_1, \ldots, i_k \rangle$ is represented by the product of its corresponding components in $B_1, \ldots, B_{k-1}$, finally multiplied by x_{i_k} which corresponds to the final vertex of the walk. By substituting the $B_c[i_c, i_{c+1}]$'s for their values, and re-arranging the walk's product such that the x_i terms appear first, and then the z term, it follows that

$$P(x_1, \ldots, x_n, z) = \sum_{\substack{I \\ I = \langle i_1, \ldots, i_k \rangle \text{ is a walk in } H}} x^I \cdot z^{w(I)} \ , \tag{9}$$

where $x^I = x_{i_1} \cdots x_{i_k}$ and $w(I) = \sum_{e \in I} w(e)$ is the weight of walk I. In other words, each k-walk is represented in P as a product of the k variables representing the vertices visited by the walk, finally multiplied by z to the power of the walk's weight. For a k-simple-path I that visits each vertex at most once, x^I is multilinear. Therefore, the minimum-degree (w.r.t. z) monomial having a multilinear x^I corresponds to the minimum-weight k-path. By running the algorithm from the previous section on P, we can find its weight. We now have the following theorem:

Theorem 2. *The weight of the minimum-weight k-path can be found with high probability in $\tilde{O}(2^k \mathrm{poly}(k) M n^2) = O^*(2^k M)$.*

Proof. Correctness follows from the above discussion. As for the running time, the circuit representation of the polynomial P is comprised of $k-1$ vector-by-matrix multiplication, and one vector-by-vector multiplication, and therefore $s(n) = O(k \cdot n^2)$. The overall running time is thus $\tilde{O}(2^k \mathrm{poly}(k) M n^2) = O^*(2^k M)$, by Theorem 1. □

5 Finding the Actual Path

Let $G = (V, E, w)$ be a weighted graph with integer edge-weights in $[-M, M]$. Given the algorithm from the previous section, we show that it is possible to find the minimum-weight k-path itself with only $O(k \mathrm{poly} \log n)$ multiplicative overhead w.r.t. the previous algorithm and with a polynomially small error probability.

We denote by $\mathcal{A}$ the algorithm from the previous section, amplified by running $O(\log n)$ iterations of it and choosing the minimal result, such that its error probability is bounded by $1/n^{c'}$ for some constant c'. The algorithm for finding the actual path uses $\mathcal{A}$ as a sub-routine.

We first run $\mathcal{A}(G, k)$ on the graph. Let d be the value returned by it, i.e., the weight of the minimum-weight k-path. If $|V| > 10k$, repeat the following procedure $\Theta(\log n)$ times:[3] remove each of the graph vertices with probability $1/k$. If $\Omega(|V|/k)$ vertices were removed, run $\mathcal{A}$ on the resulting graph and k. If the algorithm had returned a result $d' = d$, then keep the vertices discarded indefinitely and stop, otherwise return them back to the graph. If after the $\Theta(\log n)$ iterations no vertices were discarded indefinitely, output “Fail”.

The above procedure is repeated as long as $|V| > 10k$. Once $|V| \le 10k$, we perform an ordinary self reduction: each time we remove a different vertex and query $\mathcal{A}$ with the resulting graph and k; if the result stays the same, we keep this vertex discarded, otherwise, we return it to the graph. Once $|V| = k$, we return the edge-set E as the resulting path. This algorithm's pseudo-code is given as Algorithm 1.

[3] For the sake of brevity, we do not give full details of the underlying constants that are required.

Algorithm 1: Finding the minimum-weight k-path.

```
1  $d \leftarrow \mathcal{A}(G, k)$
2  while $|V(G)| > 10k$ do
3      for $\Theta(\log n)$ times do
4          $G' \leftarrow$ a copy of $G$ in which each vertex is removed with probability $1/k$
5          if at least $\Omega(|V(G)|/k)$ were removed and $\mathcal{A}(G', k) = d$ then
6              $G \leftarrow G'$
7              Go to the while loop
8      return "Fail"
9  foreach remaining vertex $v \in V(G)$ and until $|V(G)| = k$ do
10     $G' \leftarrow G \setminus v$  /* $G \setminus v$ is $G$ with $v$ and its incident edges removed */
11     if $\mathcal{A}(G', k) = d$ then $G \leftarrow G'$
12 return $E(G)$
```

Error Probability. Let P be the minimum-weight k-path in G, and assume $k \geq 3$, otherwise the problem is trivial. Let T be the set of vertices removed from G in an iteration of the for loop. The probability T does not include any of the vertices of P is $(1-1/k)^k \geq 1/4$. Now assume it does not, in that case it holds that $E[|T|] = \frac{|V(G)|-k}{k} \geq \frac{9|V(G)|}{10k}$, and that $Var[|T|] = (|V(G)|-k)(1/k)(1-1/k) < |V(G)|/k$. According to Chebyshev's inequality, $|T| = \Omega(|V(G)|/k)$ with probability of at least a constant. It follows that the probability to pick T that does not hit any of the vertices in P and at the same time is $\Omega(|V(G)|/k)$ is at least a constant $\alpha > 0$. We define this event as a "success". Since we perform at most $\Theta(\log n)$ trials at each iteration of the while loop, the probability of failing in all of them is $(1-\alpha)^{\Theta(\log n)}$ which can be made at most $1/n^c$ for some constant c. By using the union-bound over the $O(k \ln n)$ (as will be explained immediately) iterations of the while loop, we get a polynomially-small error probability of at most $k \ln n/n^c$. Since the probability to fail any invocation of $\mathcal{A}$ is less than $1/n^{c'}$, by a similar union-bound argument the probability to fail in any of the calls to $\mathcal{A}$ is $O(k \log^2 n/n^{c'})$. We obtain an overall polynomially-small error probability.

Running Time. Each non-failed iteration of the while loop in Algorithm 1 discards $\Omega(|V(G)|/k)$ vertices and therefore reduces the number of vertices in the graph by a multiplicative factor of $(1 - \Omega(1/k))$. As this happens until $|V(G)| \leq 10k$, $O(k \ln n)$ iterations are enough for getting the number of vertices to $10k$. As each iteration invoked $\mathcal{A}$ at most $O(\log n)$ times, the $O(k\text{poly}\log n)$ multiplicative factor follows for this stage of the algorithm. As the for-each loop incurs only $O(k) < O(k\text{poly}\log n)$ calls to $\mathcal{A}$, the running-time analysis follows.

6 Approximation

The main drawback of the previous algorithm is that its running time has a superlinear dependency in M, the bound on an edge weight. If the weights are in

$[1, M]$ (or can be normalized to this range), we show that if we settle for a $(1+\varepsilon)$-approximation algorithm to the problem, this dependency can be brought down to $\log\log M$, by using a technique of careful adaptive scaling of the edge weights, thus bringing the overall running time to $\tilde{O}(2^k \mathrm{poly}(k) n^2(\log\log M + 1/\varepsilon))$. Our techniques are in the spirit of the FPTAS of Ergün et al. [9] for the restricted shortest path problem. We start with the following proposition:

Proposition 5. *Given a graph G with integer edge-weights in $[0, M]$, a parameter k, and a value B, it is possible to find an exact solution to the minimum-weight k-path problem of weight at most B, if such exists, or to return that no such solution exists, in time $\tilde{O}(2^k \mathrm{poly}(k) B n^2) = O^*(2^k B)$ and polynomially-small error probability.*[4]

Proof. The algorithm is identical to the previous one, except that as a first step, edges of weight greater than B are deleted from the graph, and that when multiplying two polynomials in $(\mathbb{F}[G])[z]$ of degree at most B, we truncate from the resulting polynomial any term of degree greater than B, thus keeping all polynomials throughout the algorithm at degree of at most B. As every polynomial multiplication now takes $\tilde{O}(2^k \mathrm{poly}(k) B)$ time, the running time analysis follows. $\square$

We denote with $\mathcal{B}$ the algorithm that finds an exact solution to the k-path problem of weight at most B, if such exists, or returns that no such solution exists. We will use it as a sub-routine in our approximation algorithm.

Define $k' = k - 1$ (the number of edges in a k-path), and let OPT be the minimum-weight k-path. Our approximation algorithm starts by defining an upper and a lower bound, U and L, respectively, to the weight of OPT. At first, $U = k'M$ and $L = k'$. It then iteratively fine-tunes U and L to the point where the ratio U/L is less than or equal to 2, while maintaining the invariant that $L \leq w(OPT) \leq U$. This fine tuning is done as follows.

At each iteration we let the value $X = \sqrt{LU}$ be the geometric mean of L and U, and define the value $\delta = (L/U)^{1/3} - \sqrt{L/U}$ which will serve as a scaling coefficient. Notice that $\delta > 0$ as $U > L$. We then scale-down the edge weights by a factor of $\delta U/k'$, thus defining a new weight $w'(i,j) = \left\lfloor \frac{w(i,j)}{\delta U/k'} \right\rfloor$ for each edge (i,j), and let $G' = (V, E, w')$ be the graph with the new weights. Ideally, we would like to test whether the weight of the optimal solution is less than or greater than X by calling $\mathcal{B}(G', k, \frac{X}{\delta U/k'})$; here notice that the value $\frac{X}{\delta U/k'}$ is the scaled-down equivalent of X in G'. However, while the scaling guarantees that this test can be done without incurring a high running time cost, it also introduces a loss of precision due to the floor function in the scaling: define $w_{\text{eff}}(i,j) = (\delta U/k')w'(i,j)$ as the effective weight $w'(i,j)$ simulates, then we have that $w_{\text{eff}}(i,j) \leq w(i,j) \leq w_{\text{eff}}(i,j) + \delta U/k'$, and therefore for a k-path P, we have that $w_{\text{eff}}(P) \leq w(P) \leq w_{\text{eff}}(P) + \delta U$. Therefore, in the case $w'(OPT) > \frac{X}{\delta U/k'}$ we have that $w(OPT) \geq w_{\text{eff}}(OPT) > X$, but if $w'(OPT) \leq \frac{X}{\delta U/k'}$ (and therefore

[4] B does not have to be an integer, but the effect in this case is as if $\lfloor B \rfloor$ is used.

Algorithm 2: Approximation algorithm.

```
1  $k' \leftarrow k - 1$
2  $L \leftarrow k'$
3  $U \leftarrow k'M$
4  while $U > 2L$ do
5      $X \leftarrow \sqrt{LU}$
6      $\delta \leftarrow (L/U)^{1/3} - \sqrt{L/U}$
7      Define $w' \colon E \rightarrow \mathbb{N}$ such that $w'(i,j) = \left\lfloor \frac{w(i,j)}{\delta U/k'} \right\rfloor$
8      $G' \leftarrow (V, E, w')$
9      if $\mathcal{B}(G', k, \frac{X}{\delta U/k'})$ returns a result then
10         $U \leftarrow X + \delta U$
11     else
12         $L \leftarrow X$
13 Define $w' \colon E \rightarrow \mathbb{N}$ such that $w'(i,j) = \left\lfloor \frac{w(i,j)}{\varepsilon L/k'} \right\rfloor$
14 $G' \leftarrow (V, E, w')$
15 return $\mathcal{B}(G', k, \frac{U}{\varepsilon L/k'})$
```

$w_{\text{eff}}(OPT) \leq X$) then all we can assert is that $w(OPT) \leq X + \delta U$. Therefore, a k-path returned by a call to $\mathcal{B}(G', k, \frac{X}{\delta U/k'})$ has weight at most $X + \delta U$ (and not X) w.r.t. the original graph. According to the outcome of the call to $\mathcal{B}(G', k, \frac{X}{\delta U/k'})$, we redefine U and L: if $\mathcal{B}(G', k, \frac{X}{\delta U/k'})$ returned a result, we set $U \leftarrow X + \delta U$; otherwise we set $L \leftarrow X$.

When the main loop is done (convergence is shown to exist below), we again redefine a new weight function: $w'(i,j) = \left\lfloor \frac{w(i,j)}{\varepsilon L/k'} \right\rfloor$ for each edge (i,j), the graph $G' = (V, E, w')$, and return the result of a call to $\mathcal{B}(G', k, \frac{U}{\varepsilon L/k'})$. The full algorithm pseudo-code is given as Algorithm 2.

Running Time. We first show that the main loop performs $O(\log \log M)$ iterations. Let L_i, U_i be the respective values of L, U at the start of iteration i; we will show that $U_{i+1}/L_{i+1} \leq (U_i/L_i)^{2/3}$. At the end of each iteration i, we have that either $L_{i+1} \leftarrow L_i$ and $U_{i+1} \leftarrow X + \delta U_i$, or that $L_{i+1} \leftarrow X$ and $U_{i+1} \leftarrow U_i$, where $X = \sqrt{L_i U_i}$ and $\delta = (L_i/U_i)^{1/3} - \sqrt{L_i/U_i}$. In the former case we have that

$$\frac{U_{i+1}}{L_{i+1}} = \frac{X + \delta U_i}{L_i} = \frac{\sqrt{L_i U_i} + \left(\left(\frac{L_i}{U_i} \right)^{1/3} - \sqrt{\frac{L_i}{U_i}} \right) U_i}{L_i} = \frac{\left(\frac{L_i}{U_i} \right)^{1/3} U_i}{L_i} = \left(\frac{U_i}{L_i} \right)^{2/3} , \tag{10}$$

and in the latter

$$\frac{U_{i+1}}{L_{i+1}} = \frac{U_i}{X} = \frac{U_i}{\sqrt{L_i U_i}} = \sqrt{\frac{U_i}{L_i}} \leq \left(\frac{U_i}{L_i} \right)^{2/3} . \tag{11}$$

In both cases we have that $U_{i+1}/L_{i+1} \leq (U_i/L_i)^{2/3}$. Therefore it converges to a constant after $O(\log\log M)$ iterations. Notice that an invocation of $\mathcal{B}(G', k, \frac{X}{\delta U/k'})$ costs $\tilde{O}(2^k \text{poly}(k) n^2)$ by Proposition 5, with the bound $B = \frac{X}{\delta U/k'}$ which is $O(k)$, as $\delta U = \Omega(X)$. We conclude that the overall cost of the main loop is $\tilde{O}(2^k \text{poly}(k) n^2 \log\log M)$.

As for the final call to $\mathcal{B}(G', k, \frac{U}{\varepsilon L/k'})$, we have that its running time is $\tilde{O}(2^k \text{poly}(k) n^2/\varepsilon)$ by Proposition 5, with the bound $B = \frac{U}{\varepsilon L/k'}$ which is $O(k/\varepsilon)$ since at this stage $U \leq 2L$. We conclude that the overall running time of the approximation algorithm is $\tilde{O}(2^k \text{poly}(k) n^2 (\log\log M + 1/\varepsilon))$.

Correctness. Throughout the execution, the algorithm maintains the invariant that $L < X < X + \delta U < U$. That can be easily seen by substituting X and δ for their values and observing that $L < \sqrt{LU} < L^{1/3}U^{2/3} < U$. Assume that there exists a k-path in G, and let OPT be the minimum-weight k-path. By the scaling arguments, and the fact that we have brought the loss of precision due to scaling into consideration when redefining U and L, we have that the invariant $L \leq w(OPT) \leq U$ always holds. Due to the running-time argument, when the main loop is done we have $U/L \leq 2$. Let P^* be the result of the call to $\mathcal{B}(G', k, \frac{U}{\varepsilon L/k'})$ at line 15 of the pseudo-code, and notice that the weights defined at line 13 incur an $\varepsilon L/k'$ loss of precision per edge, or equivalently εL per k-path. By the call to the exact algorithm, we have that $w'(P^*) \leq w'(OPT)$ and therefore also $w_{\text{eff}}(P^*) \leq w_{\text{eff}}(OPT)$. Accounting for the loss of precision, we have that $w(P^*) \leq w_{\text{eff}}(P^*) + \varepsilon L \leq w_{\text{eff}}(OPT) + \varepsilon L \leq (1+\varepsilon) w(OPT)$.

7 *k*-Tree

In [13], they provide a solution to the k-tree problem: given an n-vertex graph G and a k-node tree T, is there a (not necessarily induced) copy of T in G. Again their solution is based on a reduction to the question of is there a k-multilinear-monomial in the sum-product expansion of a given polynomial. We show how to handle the *minimum-weight k-tree* problem—in which we are given a weighted graph G, and wish to find a minimum-weight copy of T in it, across all copies of T in it—again, when the weights are integers in a given range $[-M, M]$.

Theorem 3. *Given a graph G, if the edge-weights are integers in $[-M, M]$, the minimum-weight k-tree can be found in $\tilde{O}(2^k \text{poly}(k) M n^3)$ time. If the edge-weights are reals in $[1, M]$, the problem can be approximated within $(1+\varepsilon)$ in $\tilde{O}(2^k \text{poly}(k) n^3 (\log\log M + 1/\varepsilon))$ time.*

Let $N_G(i)$ be the neighbor-set of vertex i in G, and let $X = \{x_1, \ldots, x_n\}$ be a variable-set corresponding to $V(G)$. We define the following polynomial on X, implemented as an arithmetic circuit:

Let $V(G) = [n]$ and $V(T) = [k]$. The polynomial $C_{T,i,j}(x_1, \ldots, x_n, z)$ is defined as follows. If $|V(T)| = 1$, then $C_{T,i,j} = x_j$. Otherwise, $C_{T,i,j}$ is defined recursively: let $\{T_{i,\ell} \mid \ell \in N_T(i)\}$ be the subtrees of T created by removing node i from T, where $T_{i,\ell}$ is the subtree containing ℓ. Then

$$C_{T,i,j} = x_j \prod_{\ell \in N_T(i)} \left(\sum_{j' \in N_G(j)} z^{w(j,j')} C_{T_{i,\ell},\ell,j'} \right) , \tag{12}$$

where as before, z is a symbolic variable. Finally, define the polynomial $P = \sum_{j \in V(G)} C_{T,1,j}$. We shall find the minimum degree of a coefficient containing a multilinear monomial in P, using the algorithm of Section 3, executed on P. This degree is the weight of the minimum-weight k-tree in G, as will be shown in the following:

Proposition 6. *The weight of the minimum-weight k-tree can be found with high probability in $\tilde{O}(2^k \mathrm{poly}(k) M n^3)$.*

Proof. P is a sum over all homomorphisms from T to subgraphs of G of size at most k: specifically $C_{T,i,j}$ aggregates over all homomorphisms that map $i \in V(T)$ to $j \in V(G)$ (proof can be found in [13][5]). Therefore, a monomial $x_{j_1} \cdots x_{j_k} \cdot z^d$ appears in the sum-product expansion of P if an only if there is a homomorphism mapping $V(T)$ to $\{j_1, \ldots, j_k\}$ such that if $(i, \ell) \in E(T)$, then $(j_i, j_\ell) \in E(G)$. If such a monomial is multilinear w.r.t. the x-variables, it corresponds to such a homomorphism in which $j_1, \ldots, j_k$ are distinct vertices, i.e., a vertex in G was not used more than once for the sake of a single mapping. Furthermore, as the monomial includes the term $z^{w(j,j')}$ in the product when the mapping uses the edge (j, j'), the degree of z in the monomial is the weight of the k-tree represented by it.

As for the running time, each $C_{T,1,j}$ can be implemented as a circuit containing $O(|E(T)| \cdot |E(G)|)$ addition and multiplication gates. To see this, notice that the expression $C_{T,1,j}$ implicitly assumes that T is a tree rooted by the node $1 \in V(T)$. To emphasize this, we define $T_1 = T$ and define T_i to be the subtree of T that has i as its root. For a rooted tree T_i, we also define $N'_T(i)$ to be i's children. When we compute the expression $C_{T',i,j}$, for some T' subtree of T, T' is actually the subtree T_i, and we can write $C_{T',i,j}$ as $C_{T_i,i,j}$. We can compute $C_{T_1,1,j}$ in a bottom-up fashion, that is, when we actually compute $C_{T',i,j} = C_{T_i,i,j}$, for some i, j, we have already computed $C_{T'_{i,\ell},\ell,j'} = C_{T_\ell,\ell,j'}$ for each $\ell \in N'_T(i), j' \in N_G(j)$. Therefore, once we have $C_{T_\ell,\ell,j'}$ for each $\ell \in N'_T(i), j' \in N_G(j)$, additional $O(|N'_T(i)| \cdot |N_G(j)|)$ operations are required to compute $C_{T_i,i,j}$. As we need to compute $C_{T_i,i,j}$ for every i, j, the overall number of gates is $\sum_{i=1}^{k} \sum_{j=1}^{n} O(|N'_T(i)| \cdot |N_G(j)|) = O(|E(T)| \cdot |E(G)|)$.

It follows that P contains $n \cdot |E(T)| \cdot |E(G)| = O(n^3 k)$ gates, and thus the overall running time is $\tilde{O}(2^k \mathrm{poly}(k) M n^3) = O^*(2^k M)$, by Theorem 1. □

From this point, notice that the method for finding the actual k-path (in Section 5), and the approximation algorithm (in Section 6) apply to the k-tree

[5] Their arithmetic circuit is defined as $\sum_{i \in V(T), j \in V(G)} C_{T,i,j}$, however, it seems to contain redundancy.

problem as well, as they are agnostic to the actual problem: the first one uses repeated calls to the above algorithm in order to throw away vertices that do not participate in the k-tree (k-path), and the second uses repeated calls to the above algorithm with different scales applied to the edge weights. We obtain that the minimum-weight k-tree problem with integer edge-weights in $[-M, M]$ can be solved in $\tilde{O}(2^k \text{poly}(k) M n^3)$ time and that if the edge-weights are reals in $[1, M]$, it can be approximated within $(1+\varepsilon)$ in $\tilde{O}(2^k \text{poly}(k) n^3 (\log\log M + 1/\varepsilon))$ time.

8 Acknowledgments

We would like to thank Ryan Williams and Danny Raz for helpful comments.

References

1. Abasi, H., Bshouty, N.H.: A simple algorithm for undirected hamiltonicity. Electronic Colloquium on Computational Complexity (ECCC) **20**, 12 (2013)
2. Alon, N., Yuster, R., Zwick, U.: Color-coding. J. ACM **42**(4), 844–856 (1995)
3. Björklund, A.: Determinant sums for undirected hamiltonicity. In: FOCS, pp. 173–182. IEEE Computer Society (2010)
4. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Narrow sieves for parameterized paths and packings. CoRR **abs/1007.1161** (2010)
5. Blum, M., Kannan, S.: Designing programs that check their work. J. ACM **42**(1), 269–291 (1995)
6. Chen, J., Lu, S., Sze, S.H., Zhang, F.: Improved algorithms for path, matching, and packing problems. In: N. Bansal, K. Pruhs, C. Stein (eds.) SODA, pp. 298–307. SIAM (2007)
7. Cygan, M., Gabow, H.N., Sankowski, P.: Algorithmic applications of baur-strassen's theorem: Shortest cycles, diameter and matchings. In: FOCS, pp. 531–540. IEEE Computer Society (2012)
8. DeMillo, R.A., Lipton, R.J.: A probabilistic remark on algebraic program testing. Inf. Process. Lett. **7**(4), 193–195 (1978)
9. Ergün, F., Sinha, R.K., Zhang, L.: An improved fptas for restricted shortest path. Inf. Process. Lett. **83**(5), 287–291 (2002)
10. Hassidim, A., Keller, O., Lewenstein, M., Roditty, L.: Finding the minimum-weight k-path. In: F. Dehne, R. Solis-Oba, J.R. Sack (eds.) WADS, *Lecture Notes in Computer Science*, vol. 8037, pp. 390–401. Springer (2013)
11. Kneis, J., Mölle, D., Richter, S., Rossmanith, P.: Divide-and-color. In: F.V. Fomin (ed.) WG, *Lecture Notes in Computer Science*, vol. 4271, pp. 58–67. Springer (2006)
12. Koutis, I.: Faster algebraic algorithms for path and packing problems. In: L. Aceto, I. Damgård, L.A. Goldberg, M.M. Halldórsson, A. Ingólfsdóttir, I. Walukiewicz (eds.) ICALP (1), *Lecture Notes in Computer Science*, vol. 5125, pp. 575–586. Springer (2008)
13. Koutis, I., Williams, R.: Limits and applications of group algebras for parameterized problems. In: S. Albers, A. Marchetti-Spaccamela, Y. Matias, S.E. Nikoletseas, W. Thomas (eds.) ICALP (1), *Lecture Notes in Computer Science*, vol. 5555, pp. 653–664. Springer (2009)

14. Monien, B.: How to find long paths efficiently. Annals of Discrete Mathematics **25**, 239–254 (1985)
15. Schwartz, J.T.: Fast probabilistic algorithms for verification of polynomial identities. J. ACM **27**(4), 701–717 (1980)
16. Williams, R.: Finding paths of length k in $o^*(2^k)$ time. Inf. Process. Lett. **109**(6), 315–318 (2009)
17. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: E.W. Ng (ed.) EUROSAM, *Lecture Notes in Computer Science*, vol. 72, pp. 216–226. Springer (1979)
18. Zwick, U.: All pairs shortest paths using bridging sets and rectangular matrix multiplication. J. ACM **49**(3), 289–317 (2002)