Finding, Minimizing, and Counting Weighted Subgraphs*

[Extended Abstract]

Virginia Vassilevska
School of Mathematics
Institute for Advanced Study
Princeton, NJ 08540 USA
virgi@math.ias.edu

Ryan Williams
School of Mathematics
Institute for Advanced Study
Princeton, NJ 08540 USA
ryanw@math.ias.edu

ABSTRACT

For a pattern graph H on k nodes, we consider the problems of finding and counting the number of (not necessarily induced) copies of H in a given large graph G on n nodes, as well as finding minimum weight copies in both node-weighted and edge-weighted graphs. Our results include:

- The number of copies of an H with an independent set of size s can be computed exactly in $O^*(2^s n^{k-s+3})$ time. A minimum weight copy of such an H (with arbitrary real weights on nodes and edges) can be found in $O(4^{s+o(s)}n^{k-s+3})$ time. (The O^* notation omits $\mathsf{poly}(k)$ factors.) These algorithms rely on fast algorithms for computing the permanent of a $k \times n$ matrix, over rings and semirings.
- The number of copies of any H having minimum (or maximum) *node-weight* (with arbitrary real weights on nodes) can be found in $O(n^{\omega k/3} + n^{2k/3+o(1)})$ time, where $\omega < 2.4$ is the matrix multiplication exponent and k is divisible by 3. Similar results hold for other values of k. Also, the number of copies having exactly a prescribed weight can be found within this time. These algorithms extend the technique of Czumaj and Lingas (SODA 2007) and give a new (algorithmic) application of multiparty communication complexity.
- Finding an *edge-weighted* triangle of weight exactly 0 in general graphs requires $\Omega(n^{2.5-\varepsilon})$ time for all $\varepsilon > 0$, unless the 3SUM problem on N numbers can be solved in $O(N^{2-\varepsilon})$ time. This suggests that the edge-weighted problem is much harder than its node-weighted version.

*This material is based on work supported in part by NSF grant CCF-0832797 at Princeton University/IAS and Princeton University sub-contract to IAS No.00001583. Any opinions, findings and conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of the National Science Foundation.

STOC'09, May 31–June 2, 2009, Bethesda, Maryland, USA.

Categories and Subject Descriptors

F.2.2 [**ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY**]: Nonnumerical Algorithms and Problems; G.2.2 [**DISCRETE MATHEMATICS**]: Graph Theory—*Path and circuit problems, Graph algorithms*

General Terms

Algorithms, Theory

Keywords

subgraph finding, 3SUM, paths, cliques, weighted graph

1. INTRODUCTION

We consider the problems of finding and counting the copies of a fixed k node graph H in a given n node graph G (such copies are called H-subgraphs). We also study the case of finding and counting maximum weight copies when G has arbitrary real weights on its vertices or edges.

Subgraphs With Large Independent Sets.

In the unweighted case, the best known algorithm for counting H-subgraphs uses Coppersmith-Winograd matrix multiplication [14] and runs in $\Omega(n^{\omega k/3}) \geq \Omega(n^{0.791k})$ time and $n^{\Theta(k)}$ space. We present algorithms that do not rely on fast matrix multiplication yet still beat the above in both runtime and space usage, for H with a large independent set. In particular, if H has an independent set of size s, we can count the number of copies of H in an n-node graph in polynomial space and $O(4^{s+o(s)}n^{k-s}n^3)$ or $O(s! \cdot n^{k-s}sn^2)$ time, or in $O(2^s n^{k-s}n^3)$ (and exponential space).

Furthermore, our polynomial space algorithms can be used to find minimum weight H-subgraphs in a graph with arbitrary real edge weights. These improvements are obtained via new algorithms for computing the permanent of a rectangular matrix over a semiring. Our algorithms are simple and the runtime analysis does not hide huge constants.

Our results on counting and finding maximum subgraphs are interesting for both practical and theoretical reasons. On the practical side, pattern subgraph counting and detection are used in diverse areas, including the analysis of social networks

[8, 39, 36], computational biology, and network security [12, 22, 37]. In molecular biology, biomolecular networks are compared by identifying so-called *network motifs* [29] – connectivity patterns that occur much more frequently than expected in a random graph. Similar techniques are used to detect abnormal patterns in social networks (potential spammers, bots) and undesirable usage patterns in a computer network. Because of the extensive computational overhead of previous exact counting techniques, *approximate* counting based on the color coding technique [4] is typically used for pattern graphs on ≥ 4 nodes (e.g. [1]). Unfortunately, even for approximately counting trees, the current methods are not efficient for patterns with more than 9 nodes. Because some of the pattern graphs have large independent sets, we suspect our methods will be useful in the above settings: for instance, trees with many leaves will be counted fairly quickly.

On the theoretical side, our algorithms are interesting because the problem of counting k-subgraphs (even k-paths) is $\#W[1]$-complete (whereas approximately counting k-paths is not, cf. [2, 19, 6]). Hence if one could obtain a $O(n^{\alpha(k)})$ time algorithm for counting for a small enough function α, the Exponential Time Hypothesis would be false, and many NP problems would have subexponential algorithms. Alon and Gutner [2] have proven in a formal sense that the color-coding method cannot hope to do better than $O(n^{k/2})$ for counting paths exactly. Namely, for any family $\mathcal{F}$ of "balanced" functions from $[n]$ to $[k]$, we must have $|\mathcal{F}| \geq \Omega(n^{k/2})$. As we obtain $O(f(k)n^{k/2+c})$ algorithms, our results may be optimal in some sense (although they do not use color coding).

Node-Weighted Subgraphs Via Matrix Products.

In the second part of the paper, we give algorithms that apply fast matrix multiplication to find and count weighted H-subgraphs for *general* H. We consider three variants of the problem: finding and counting H-subgraphs of maximum weight, weight at least K, and weight exactly K (for any given weight K). Due to its relation to the all pairs shortest paths problem, the maximum weight version has received much recent attention.

The current best algorithm for *finding* a maximum weight H-subgraph in a *node-weighted* graph is by Czumaj and Lingas [16] and runs in $O(n^{\omega k/3+\varepsilon})$ time for all $\varepsilon > 0$ (when k is divisible by 3; other cases are similar). We show how to extend their approach to *counting* maximum weight H-subgraphs in the same time. Moreover, we show that the problem of counting the number of H-subgraphs of node weight at least K and even *exactly* K can also be done in the same time. The previous best algorithm for either of these problems is based on the dominance product method [40] and has a running time of $O(n^{\frac{(3+\omega)}{2}\frac{k}{3}})$ (for k divisible by 3). Our algorithms rely on a new $O(n^\omega + n^2 2^{O(\sqrt{\log n})})$ algorithm for counting the number of triangles of weight K in a node-weighted graph. In fact, we give two very different algorithms for exact node-weighted triangles: one based on the Czumaj-Lingas approach, and one based on a counterintuitive 3-party communication protocol for the Exactly-W problem.

Hardness Results for Edge-Weighted Subgraphs.

Finally, we provide theoretical evidence that finding *edge weighted* H-subgraphs faster than $O(n^k)$ will be difficult, for general H in arbitrary weighted graphs. We focus on the problem of finding triangles of weight exactly K in an edge-weighted graph. This triangle problem is not known to have a truly subcubic algorithm. In an attempt to explain this, we prove that unless 3SUM has a truly subquadratic algorithm, a triangle of weight sum K in an edge weighted graph cannot be found in $O(n^{2.5-\varepsilon})$ time for any $\varepsilon > 0$. 3SUM is widely believed to require essentially quadratic time (cf. [7] for a slight improvement), so our result suggests that the exact triangle problem for edge-weighted graphs is harder than that for node-weighted graphs. Patrascu [33] has recently observed that using more properties of the hash function in our reduction, the conditional lower bound for exact weighted triangles can be improved optimally to $\Omega(n^3)$, i.e. unless 3SUM has subquadratic algorithms, finding a triangle of weight 0 in an edge-weighted graph requires cubic time(!). We also show that subcubic algorithms for edge weighted triangle imply faster-than-2^n algorithms for multivariate quadratic equations, an important NP-complete problem in cryptography.

Prior Work.

Besides the references we have already mentioned, the theoretical problems of subgraph finding and counting are discussed in many works, for example [24, 31, 13, 27, 38]. Alon, Yuster and Zwick [5] showed that for all $k \leq 7$ the number of k-cycles in an unweighted graph can be computed in $O(n^\omega)$ time using fast matrix multiplication. Unfortunately their approach does not generalize for $k > 7$. Björklund et al. [10] have recently found an interesting algorithm for counting k-paths that runs in $\binom{n}{k/2}\,\mathsf{poly}(n)$ time. For sufficiently large k, their algorithm is faster than ours. However, their algorithm only works for k-paths and uses $\Omega(\binom{n}{k/2})$ space. For the special case where H is a bipartite graph, our algorithm uses $2^{k+o(k)}n^{k/2+3}$ time and $\mathsf{poly}(n,k)$ space.

Preliminaries.

For a node u in a graph (V,E), $N(u) = \{v \in V \mid (u,v) \in E\}$. For an integer n, let $[n] = \{1,2,\ldots,n\}$.

A graph homomorphism f from a graph $G = (V,E)$ to a graph $H = (V_H, E_H)$ is a mapping $f : V \to V_H$ so that if $(u,v) \in E$, then $(f(u), f(v)) \in E_H$. A graph isomorphism f from a graph $G = (V,E)$ to a graph $H = (V_H, E_H)$ is a bijective map from G to H such that both f and f^{-1} are homomorphisms. An automorphism is an isomorphism between a graph G and itself.

2. ALGORITHMS WITHOUT MATRIX MULTIPLICATION

We begin by reducing the problems of counting and minimizing subgraphs to computing permanents of rectangular matrices. We assume that all given graphs are undirected, but it is not hard to modify the proofs for directed graphs.

THEOREM 2.1. *Suppose the permanent of an $s \times n$ matrix can be computed in $T(n,s)$ time and $S(n,s)$ space. Let $H = \{h_1,\ldots,h_k\}$ be a graph on k nodes with an independent set of size s. Let $G = (V,E)$ be a graph on n nodes and $w : E \to \mathbb{R}$ be a weight function. Let C be the set of all (not necessarily induced) copies of H in G. Then the quantity*

$$\sum_{H' \in C} \prod_{e \in E(H')} w(e)$$

can be determined in $O((nks+T(n,s))\cdot(k-s)!\binom{n}{k-s})$ *time and* $O(ns+S(n,s))$ *space.*

Note that when $w(e)=1$ for all $e\in E$, the quantity in the theorem is just the number of (not necessarily induced) copies of H in G.

PROOF. Let I be an independent set of size s in H. Let $t=k-s$. Let $H'=H\setminus I$, with $H'=\{h_1,\ldots,h_t\}$ and $I=\{s_1,\ldots,s_s\}$. Our algorithm proceeds by iterating over all ordered t-tuples $T=(v_1,\ldots,v_t)$ of distinct nodes. It discards T if the map $h_i\to v_i$ for $i\in[t]$ is not a homomorphism. Note the number of choices for T is $t!\cdot\binom{n}{t}$.

Consider an ordered s-tuple $X=(x_1,\ldots,x_s)$ of distinct nodes. X is *good with respect to* T if, for every edge (h_i,s_j) between H' and I, the edge (v_i,x_j) is in G. Let

$$w(X,T)=\prod_{h_i\in H',s_j\in I,(h_i,s_j)\in E(H)} w(v_i,x_j),\text{ and}$$

$$w(T)=\prod_{v_i,v_j\in T,(h_i,h_j)\in E(H)} w(v_i,v_j).$$

Let $N_T=\sum_X w(X,T)$ where the sum ranges only over X that are good with respect to T. [1] Then the quantity of interest is

$$\frac{1}{|Aut(H)|}\sum_T w(T)N_T,$$

where $|Aut(H)|$ is the number of automorphisms of H. We want to compute each N_T in $O(T(n,s))$ time.

For a given $T=(v_1,\ldots,v_t)$ we make an $s\times n$ matrix A as follows. For a fixed $i\in[s]$ and $s_i\in S$, consider the neighbors of s_i in H, $N(s_i)=\{h_{i_1},\ldots,h_{i_{d'}}\}$ (for $d'\le(k-s)$). For every $j\notin T$, set

$$A[i,j]=\prod_{\ell\in[d'],(v_{i_\ell},j)\in E} w(v_{i_\ell},j),$$

else set $A[i,j]=0$. It takes $O(ns(k-s))$ time to create a matrix A. Over all T, it takes $O(nks(k-s)!\binom{n}{k-s})$ time to set up all A matrices.

The permanent of A is exactly N_T: it iterates over the ways to pick an ordered s-tuple $x_1,\ldots,x_s$ of distinct nodes from $V\setminus T$ so that if h_k is a neighbor of s_i in H, then x_i is a neighbor of v_k, summing over the edge weight products. The number of $s\times n$ permanent computations that we need to do is $(k-s)!\binom{n}{k-s}$. The space used is $O(ns+S(n,s))$ since we just need to store one matrix of size $s\times n$ at any point.

Finally, we observe that computing $|Aut(H)|$ takes negligible time, since we can apply the same approach. To compute $|Aut(H)|$, enumerate all $(k-s)!\binom{k}{k-s}$ ordered $(k-s)$-tuples T_H of distinct nodes of H which are isomorphic to H'. Then by using an $s\times k$ permanent computation we can determine the number N_{T_H} of good s-tuples X with respect to T_H, setting $|Aut(H)|=\sum_{T_H}N_{T_H}$. Hence $|Aut(H)|$ is computable in $(k-s)!\binom{k}{k-s}T(k,s)\le(k-s)!\binom{n}{k-s}T(n,s)$ time. □

A variant of the above also works for semirings where the addition operation is min or max.

[1] In the case where H is a k-path and G is unweighted, note N_T is the number of paths of the form $v_1\to w_1\to v_2\to w_2\to\cdots\to w_{t-1}\to v_t$, where the w_i are all distinct.

THEOREM 2.2. *Let* R *be a semiring with* min *(or* max*) as its addition operation, and* $\otimes$ *as its multiplication operation. Suppose the permanent of an* $s\times n$ *matrix over* R *can be computed in* $T(n,s)$ *time and* $S(n,s)$ *space. Let* $H=\{h_1,\ldots,h_k\}$ *be a graph on* k *nodes with an independent set of size* s*. Let* $G=(V,E)$ *be a graph on* n *nodes and* $w:E\to R$ *be a weight function. Let* C *be the set of all (not necessarily induced) copies of* H *in* G*. Then the quantity*

$$\min_{H'\in C}\bigotimes_{e\in E(H')} w(e)$$

(or the max*) can be determined in* $O((snk+T(n,s))\cdot(k-s)!\binom{n}{k-s})$ *time and* $O(sn+S(n,s))$ *space.*

PROOF. Analogous to the proof of Theorem 2.1, except we do not need to compute $Aut(H)$ in order to compute the minimum (or maximum). That is, the permanent of A over the semiring is just the minimum (maximum) value of $w(T)\otimes N_T$ over all t-tuples T. □

Let H be any graph on k nodes. Suppose H contains an independent set I of size s. Let G be an n node graph. Using the permanent algorithms of the next section, we obtain the below corollaries of Theorems 2.1 and 2.2.

COROLLARY 2.3. *There is an algorithm which counts the number of copies of* H *in* G*, in*

$$O\left(n^2(k-s)!\binom{n}{k-s}\min\left\{s!,n4^{s+o(s)}\right\}\right)$$

time. The algorithm uses $\mathsf{poly}(n,k)$ *space.*

COROLLARY 2.4. *Let* H *be a bipartite graph on* k *nodes. The number of copies of* H *in an* n *node graph* G *can be counted in* $k!\binom{n}{k/2}\,\mathsf{poly}(n)$ *time.*

COROLLARY 2.5. *There exists an* $O(n^3(k-s)!\binom{n}{k-s}s2^s)$ *time algorithm which counts the number of copies of* H *in* G*. The algorithm uses* $\mathsf{poly}(n,k)+O(n^2 2^s)$ *space.*

COROLLARY 2.6. *Let* G *be a graph with real weights on its edges. There is an* $O(n^2(k-s)!\binom{n}{k-s}\min\{s!,\ n4^{s+o(s)}\})$ *time algorithm which can find a* minimum weight *copy of* H *in* G*. The algorithm uses* $\mathsf{poly}(n,k)$ *space.*

The last corollary is obtained by applying Theorem 2.2 with a permanent computation over the $(\min,+)$-semiring (where addition is min, and multiplication is $+$, over $\mathbb{R}\cup\{\infty,-\infty\}$). By negating all weights we can compute the *maximum weight* copy as well. Note if the weights on edges are treated as probabilities, and we wish to find a copy of H with *maximum probability*, this can be found by working over the $(\max,\times)$-semiring.

2.1 Computing Permanents of Rectangular Matrices

We now investigate the problem of computing the permanent on matrices with a small number of rows. The best known algorithm for computing the permanent is very old, due to Ryser [35]. He gives a formula based on inclusion-exclusion that computes the permanent of an $n\times n$ matrix over a ring in $O(2^n\,\mathsf{poly}(n))$ time and $O(\mathsf{poly}(n))$ space. There are two

downsides to his algorithm (other than its high running time). First, it cannot be feasibly applied to algebraic structures without subtraction, due to its use of the inclusion-exclusion principle.[2] Secondly, when one tries to generalize the formula to $k \times n$ matrices, one only obtains an $O(\binom{n}{k}\,\mathsf{poly}(n))$ time algorithm (this is well-known folklore [30]). Both of these prevent us from using Ryser's algorithm in the algorithms of the previous section. Kawabata and Tarui [25] have given a $k \times n$ permanent algorithm over rings that runs in $O(2^k n + 3^k)$ time and $O(2^k)$ space, by exploiting the Binet-Minc formula for the permanent [30]. In this section, we present new algorithms that work over commutative semirings and run in FPT time with respect to k.

Over the integers, the permanent of a $k \times n$ Boolean matrix counts the number of matchings in a bipartite graph with one partition of size k and the other of size n. The more general $\#k$-MATCHING problem is to count the number of matchings on k nodes in an n node graph. It is a major open problem in parameterized complexity to determine if $\#k$-MATCHING is FPT or if it is $W[1]$-hard [18]. We do not resolve the complete problem here, but our results do show that for some bipartite graphs (with $f(k)$ vertices in one partition, for some function f) the problem is fixed-parameter tractable. Our results also imply a $2^{k+o(k)}\binom{n}{k/2}\,\mathsf{poly}(n)$ time, polynomial space algorithm for $\#k$-MATCHING.

THEOREM 2.7. *The permanent of a $k \times n$ matrix A can be computed in $O(k! \cdot kn^2)$ operations over any finite commutative semiring.*

Note that we count time in terms of the number of plus and times operations over the semiring along with other basic machine instructions, and we count space in terms of the total number of elements of the semiring that need to be stored at any given point in the computation.

PROOF. For a $k \times n$ matrix A where $k \leq n$, we have

$$perm(A) = \sum_{\substack{f:[k]\to[n] \\ f \text{ is 1-1}}} \left(\prod_{i=1}^{k} A[i, f(i)] \right).$$

Our permanent algorithm tries all possible permutations $\pi : [k] \to [k]$ of the rows in A. Let A_π be the resulting matrix. A function f on $[k]$ is *increasing* if $f(i+1) > f(i)$ for all $i = 1, \ldots, k-1$. Given π, define

$$perm^*(A) = \sum_{f \text{ is increasing}} \left(\prod_{i=1}^{k} A[i, f(i)] \right).$$

Observe that

$$perm(A) = \sum_{\pi} perm^*(A_\pi),$$

since for any one-to-one f there is a unique permutation π on $[k]$ such that f' with $f'(i) = f(\pi(i))$ is increasing.

We now show how to compute each $perm^*(A_\pi)$ efficiently. Make a layered DAG having k layers and at most n nodes per layer. We include a node labelled j in layer i if and only if $A_\pi[i,j] \neq 0$. Give the node labelled j in layer i a weight of $A_\pi[i,j]$. Now from layer i to layer $i+1$, put arcs from all nodes labelled j to all nodes labelled j', for all $j < j'$.

Finally, we need to sum the weights of all k-paths in this DAG, where a path with node weights $w_1, \ldots, w_k$ is said to have weight $\prod_{i=1}^{k} w_i$. Note this sum is precisely $perm^*(A_\pi)$. The idea is to process the nodes in topological order and do dynamic programming. At each node v, we maintain the weight W_i^v of all i-paths that end with v, for all $i = 1, \ldots, k$. Observe when v has indegree 0, computing W_i^v is trivial. For an arbitrary node v, we may assume that we have already computed the $W_i^{v'}$'s, for all nodes v' with arcs to v. Let the nodes with arcs to v be $v'_1, \ldots, v'_d$ and let $w(v)$ be the weight of node v. Clearly, $W_1^v = w(v)$. For every $i = 1, ..., k-1$, compute

$$W_{i+1}^v = \left(\sum_{j=1}^{d} W_i^{v'_j} \right) \cdot w(v).$$

When this process completes, we have the weights of all k-paths that end in each node v. It follows that $perm^*(A_\pi) = \sum_v W_k^v$. □

We can improve the dependence on k by using recursion.

THEOREM 2.8. *The permanent of a $k \times n$ matrix can be computed in $O(4^{k+o(k)}n^3)$ time and $O(kn^2)$ space over any commutative semiring.*

PROOF. Let A be the matrix. The idea is to try all possible partitions of $[k]$ into sets L and R of cardinality $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$ respectively, performing a recursive call on an $|L| \times n$ and an $|R| \times n$ submatrix (one indexed by L, one indexed by R) which returns all the information we need to reconstruct the permanent. More precisely, let $j_1 \leq j_2$ and define $A_L^{j_1,j_2}$ to be the $|L| \times |j_2 - j_1 + 1|$ submatrix of A with rows indexed by L and columns ranging from the j_1th column of A to the j_2th column of A. Note $A = A_{[k]}^{1,n}$. Let

$$B_L^{j_1,j_2} = \sum_{\ell \in L} A[\ell, j_2] \cdot perm(A_{L\setminus\{\ell\}}^{j_1,j_2-1}), \text{ and}$$

$$C_L^{j_1,j_2} = \sum_{\ell \in L} A[\ell, j_1] \cdot perm(A_{L\setminus\{\ell\}}^{j_1+1,j_2}).$$

The following identity is the key to the algorithm (the proof appears in the full version):

CLAIM 2.9.

$$perm(A) = \sum_{\substack{L \subseteq [k] \\ |L| = \lfloor k/2 \rfloor}} \sum_{1 \leq j_2 < j_3 \leq n} B_L^{1,j_2} \cdot C_{|k|-L}^{j_3,n}.$$

We give a simple algorithm PERMANENT to recursively compute $perm(A)$ using the claim. In particular, given a $k \times n$ matrix A, the algorithm returns an $n \times n$ matrix M where $M[i,j] = perm(A_{[k]}^{i,j})$. Hence $M[1,n] = perm(A)$.

The correctness of PERMANENT follows from Claim 2.9. A naive way to construct the M' of the algorithm requires $\Theta(n^4)$ time. To implement it in $O(n^3)$, first compute for all i, j, ℓ, $N_L[i,\ell] = \sum_{x=i}^{\ell} B_L[i,x]$ and $N_R[\ell,j] = C_R[\ell+1,j]$ whenever $\ell < j$ and $N_R[\ell,j] = 0$ otherwise. Via dynamic

[2] It is possible to apply the algorithm to structures like the $(\min, +)$-semiring by embedding that structure in the ring, but such embeddings require an exponential blowup in the representations of elements in the semiring, cf. Romani [34], Yuval [43].

PERMANENT(A):
  If $k = 1$ then
    Return $n \times n$ M with $M[i,j] = \sum_{\ell: i \le \ell \le j} A[1,\ell]$
  $M :=$ the $n \times n$ matrix of all zeroes
  For all $L \subseteq [k]$ with $|L| = \lfloor k/2 \rfloor$,
    Let B_L and C_L be initially all zero
    For all $\ell \in L$:
      Let $M_{L-\{\ell\}} :=$ PERMANENT($A^{1,n}_{L-\{\ell\}}$).
      For all i, j_2:
        add $A[\ell, j_2] \cdot M_{L-\{\ell\}}[i, j_2 - 1]$ to $B_L[i, j_2]$.
    For all $\ell' \in [k] - L$:
      Let $M_{[k]-L-\{\ell'\}} :=$ PERMANENT($A^{1,n}_{[k]-L-\{\ell'\}}$).
      For all j_3, j:
        add $A[\ell', j_3] \cdot M_{[k]-L-\{\ell'\}}[j_3 + 1, j]$ to $C_L[j_3, j]$.
    Define M' by
      $M'[i,j] = \sum_{j_2, j_3 : i \le j_2 < j_3 \le j} B_L[i, j_2] \cdot C_L[j_3, j]$
    $M := M + M'$.
  Return M.

programming, building up N_R and N_L takes only $O(n^2)$ operations. We claim that $M = N_L \cdot N_R$ where the matrix product is over the semiring. Indeed, for all i, j we have

$$\sum_{\ell} N_L[i,\ell] \cdot N_R[\ell, j]$$
$$= \sum_{\ell:\ i \le \ell \le j} (B_L[i,i] + \cdots + B_L[i,\ell]) \cdot C_R[\ell+1, j]$$
$$= \sum_{\ell_1, \ell_2 : i \le \ell_1 < \ell_2 \le j} B_L[i, \ell_1] \cdot C_R[\ell_2, j] = M'[i,j].$$

The runtime recurrence is

$$T(k) \le k \binom{k}{\lceil k/2 \rceil} \left(T(k/2) + O(n^3) \right),$$

yielding $T(k) \le O(k^{\log k} 4^k n^3)$. The space bound holds, since only $O(n^2)$ semiring elements are stored in each recursive call. □

We remark that Gurevich and Shelah [23] gave a 4^n poly(n) algorithm for solving TSP, by trying all partitions of the vertices into two halves and recursing. In retrospect, the above approach is similar in spirit.

Finally, we can obtain a faster permanent algorithm over rings. While it also uses exponential space, it still exponentially improves on Kawabata and Tarui's algorithm [25]. We require a lemma which is a simple extension of the fast subset convolution of Bjorklund et al. [9].

LEMMA 2.10. *Let N be a positive integer and R be a ring. Let f be a function from the subsets of $[N]$ of size $\lfloor K/2 \rfloor$ to R and let g be a function from the subsets of $[N]$ of size $\lceil K/2 \rceil$ to R. Suppose we are given oracle access to f and g. Consider h which is a function on the subsets of $[N]$ of size K to R and is defined as follows:*

$$h(S) = \sum_{L:\ |L| = \lfloor K/2 \rfloor} f(L) \cdot g(S - L).$$

Then one can compute $h(S)$ for all $S \subset [N], |S| = K$ in overall $O(K 2^N)$ time.

THEOREM 2.11. *The permanent of a $k \times n$ matrix over any ring can be computed in $O(kn^3 2^k)$ time and $O(n^2 2^k)$ space.*

PROOF. We use the formula from Claim 2.9 from the proof of Theorem 2.8.

Suppose that $perm(A^{j_1,j_2}_T)$ is known for all $j_1, j_2 \in [n]$ and all sets T of size $\lfloor k/2^i \rfloor - 1$, $\lfloor k/2^i \rfloor - 2$, $\lfloor k/2^i \rfloor - 3$. Then for all sets L of size $\lfloor k/2^i \rfloor$, $\lfloor k/2^i \rfloor - 1$, $\lfloor k/2^i \rfloor - 2$, B^{j_1,j_2}_L and C^{j_1,j_2}_L can be computed in $O(kn^2 2^k / 2^i)$ time. Consider A^{j_1,j_2}_S for S of size $\lfloor k/2^{i-1} \rfloor - 1$, $\lfloor k/2^{i-1} \rfloor - 2$, $\lfloor k/2^{i-1} \rfloor - 3$. From Claim 2.9 we have:

$$perm(A^{j_1,j_2}_S) = \sum_{\substack{L \subseteq S \\ |L| = \lfloor |S|/2 \rfloor}} \ \sum_{j_1 \le p_2 < p_3 \le j_2} B^{j_1,p_2}_L \cdot C^{p_3,j_2}_{S-L}.$$

The size $\lfloor |S|/2 \rfloor$ is $\lfloor k/2^i \rfloor$, $\lfloor k/2^i \rfloor - 1$, or $\lfloor k/2^i \rfloor - 2$, and $\lceil |S|/2 \rceil$ is $\lfloor k/2^i \rfloor$ or $\lfloor k/2^i \rfloor - 1$. The values for B^{j_1,p_2}_L and C^{p_3,j_2}_L for L of such sizes have been computed and stored. By computing N_L and N_R as in the previous theorem, and swapping the order of the sums in the resulting expression, we can use the fast subset convolution of Lemma 2.10 to compute $perm(A^{j_1,j_2}_S)$ in $O(n^3 k 2^k / 2^i)$ time, for all S of size $\lfloor k/2^{i-1} \rfloor - 1$, $\lfloor k/2^{i-1} \rfloor - 2$, or $\lfloor k/2^{i-1} \rfloor - 3$, and $j_1, j_2 \in [n]$.

Therefore computing $perm(A)$ takes $O(\sum_{i=0}^{\log k} kn^3 2^{k-i})$ $= O(kn^3 2^k)$ time. The space usage is $O(n^2 2^k)$ since at each stage we need to store $O(n^2 2^k)$ values. □

3. COUNTING WEIGHTED PATTERNS

In the following, we assume $k = |H|$ is divisible by 3. However our results trivially extend to all k, with possibly an extra factor of n or n^2 in the running time. The weight of a subgraph is defined to be the sum of its node (or edge) weights. A graph has *K-weight* if its weight is K.

The algorithms in the previous section can find a maximum (or minimum) weight H-subgraph in a given G. They can be extended to *count* maximum weight subgraphs if the weights in G are bounded. However, it is unclear how to extend the results of the previous section for counting general weighted subgraphs H.

There has been a lot of recent work in finding weighted H-subgraphs in node-weighted graphs ([40, 41, 16]). There are several versions of the problem: (1) find a maximum (or minimum) weight H-subgraph, (2) find an H-subgraph of weight at least K for a given K, and (3) find a K-weight H-subgraph for a given weight K. The idea which has been used in attacking all three versions of the problem is that each version can be reduced to finding a weighted (maximum, at least K, or K-weight) *triangle* in a larger node-weighted graph. If such a triangle can be found in $T(n)$ time and $S(n)$ space in an n node graph, then the corresponding weighted H-subgraph problem can be solved in $O(k^2 T(n^{k/3}))$ time and $O(S(n^{k/3}))$ space. The same reduction works for counting H-subgraphs: if the weighted triangles in an n node graph can be counted in $T(n)$ time and $S(n)$ space, then the weighted H-subgraphs can be counted in $O(k^2 T(n^{k/3}))$ time and $O(S(n^{k/3}))$ space. Here we take a similar approach, and study the corresponding triangle problems.

In previous work [40] we showed that the triangles of weight at least K in a node-weighted graph on n nodes can be counted in $O(n^{\frac{3+\omega}{2}})$ time. The same approach yielded an $O(n^{\frac{3+\omega}{2}})$ runtime for counting K-weight triangles. By binary searching

on K, this gave a way to count the maximum weight triangles in a node-weighted graph in $\tilde{O}(n^{\frac{3+\omega}{2}})$ time. This in turn implied an $O(n^{0.896k})$ running time for counting weighted H-subgraphs (for any of the three versions of the problem), and constituted the first nontrivial improvement over the brute force $O(n^k)$ runtime.

Czumaj and Lingas [16] used an interesting technique to show that a maximum weight triangle can be found in $O(n^\omega + n^{2+\varepsilon})$ time for all $\varepsilon > 0$. Their method is based on a combinatorial lemma which bounds the number of triples in a set where no triple strictly dominates another.

LEMMA 3.1 (CZUMAJ AND LINGAS [16]). *Let U be a subset of $\{1,\ldots,n\}^3$. If there is no pair of points (u_1,u_2,u_3), $(v_1,v_2,v_3) \in U$ such that $u_j > v_j$ for all $j = 1,2,3$, then $|U| \leq 3n^2$.*

We show that Lemma 3.1 can be used to solve all three versions of the weighted triangle problem in node-weighted graphs. Furthermore, it can be used to *count* node-weighted triangles in $O(n^\omega)$ time, improving on the $O(n^{\frac{3+\omega}{2}})$ time solution. The new algorithm immediately implies an $O(n^{\omega k/3})$ running time for counting weighted subgraph patterns in a node-weighted graph. We prove the result for counting exact node-weighted triangles. Counting maximum weight triangles or triangles of weight at least K can be done similarly, hence we omit those algorithms.

THEOREM 3.2. *Let $G = (V,E)$ be a given n node graph with weight function $w : V \to \mathbb{R}$. Let $K \in \mathbb{R}$ be given. Then in $n^2 \cdot 2^{O(\sqrt{\log n})} + O(n^\omega)$ time one can compute for every pair of vertices i,j the number of K-weight triangles that include the edge (i,j). Furthermore, for every (i,j) in a K-weight triangle, the algorithm can find a witness node k such that i,j,k form a K-weight triangle. The witness computation incurs only a polylogarithmic factor in the runtime.*

PROOF. Create a global $n \times n$ output matrix D that is initially zero. After the completion of the algorithm, $D[i,j]$ will contain the number of K-weight triangles that include i and j.

Sort the vertices in nondecreasing order of their weights in $O(n \log n)$ time. We build three identical sorted lists A, B, C of the n nodes. Our algorithm counts all triangles with a single node in each of A, B, and C.

The algorithm is recursive, and its input is three sorted lists of nodes A, B, C each having at most N nodes. The algorithm does not return information, but rather adds numbers to D when the recursion bottoms out.

Let c be a parameter. Partition A, B and C into c sorted sublists $\{A_1,\ldots,A_c\}$, $\{B_1,\ldots,B_c\}$, $\{C_1,\ldots,C_c\}$ of at most $\lceil N/c \rceil$ nodes each. In particular, the partition splits the sorted lists into c sorted sublists; for example, if $A = (a_1,\ldots,a_N)$ then $A_{i+1} = (a_{i\lceil N/c\rceil+1},\ldots,a_{(i+1)\lceil N/c\rceil})$ for $i+1 < c$. Each new sublist is now associated with the interval of weights between the smallest and largest weights of its nodes.

Consider all c^3 triples (A_i,B_j,C_k). Let $[a_i,a_i']$, $[b_j,b_j']$ and $[c_k,c_k']$ be the weight intervals for A_i, B_j, and C_k, respectively.

Case 1: $a_i = a_i'$, $b_j = b_j'$, or $c_k = c_k'$. If $b_j = b_j'$, then create two matrices X and Y, defined as:

$$X[p,q] = \begin{cases} 1 & \text{if } (A_i[p],B_j[q]) \in E, \\ 0 & \text{otherwise,} \end{cases} \quad \text{and}$$

$$Y[q,r] = \begin{cases} 1 & \text{if } (B_j[q],C_k[r]) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Multiply X and Y. For all p,q with $w(A_i[p])+w(C_k[q]) = K - b_j$ and $(A_i[p],C_k[q]) \in E$, $(XY)[p,q]$ gives the number of nodes in B_j which form a K-weight triangle with $A_i[p]$ and $C_k[q]$. Add $(XY)[p,q]$ to $D[A_i[p],C_k[q]]$.

The cases $a_i = a_i'$ and $c_k = c_k'$ are symmetric. WLOG assume $a_i = a_i'$. Then create two matrices X and Y as follows:

$$X[p,q] = \begin{cases} 1 & \text{if } (A_i[p],B_j[q]) \in E \\ 0 & \text{otherwise.} \end{cases}$$

$$Y[q,r] = \begin{cases} 1 & \text{if } (B_j[q],C_k[r]) \in E \text{ and} \\ & w(B_j[q]) + w(C_k[r]) = K - a_i, \\ 0 & \text{otherwise.} \end{cases}$$

Multiply X and Y. For every p,q with $(A_i[p],C_k[q]) \in E$, $(XY)[p,q]$ gives the number of nodes in B_j which form a K-weight triangle with $A_i[p]$ and $C_k[q]$. Add $(XY)[p,q]$ to $D[A_i[p],C_k[q]]$.

In both cases above, one can find witnesses and incur only a polylogarithmic factor by using the Boolean matrix product witness algorithm of Alon et al. [3].

Case 2: $a_i < a_i'$, $b_j < b_j'$, and $c_k < c_k'$. Recurse on all triples (A_i,B_j,C_k) with intervals $[a_i,a_i']$, $[b_j,b_j']$, $[c_k,c_k']$ satisfying

$$a_i + b_j + c_k \leq K \leq a_i' + b_j' + c_k'.$$

Note we can disregard all other triples of nodes, as they could not contain a K-weight triangle with a node in each of A_i, B_j, and C_k. This concludes the algorithm.

Observe that we only add to D when the recursion bottoms out, and at least one sublist has the same weight on all of its nodes. Because of the partitioning, we never overcount, and every triangle of weight K is counted exactly once.

We claim that the number of recursive calls in the algorithm is at most $6c^2$. There are two types of triples that the algorithm recurses on: those with $a_i + b_j + c_k \leq K < a_i' + b_j' + c_k'$ (type 1) and those with $a_i + b_j + c_k < K \leq a_i' + b_j' + c_k'$ (type 2). Let T_1 and T_2 be the sets of type 1 and type 2 triples respectively. We show that $|T_i| \leq 3c^2$ for $i = 1,2$.

Each triple in T_1 is uniquely determined by the three *left* endpoints of its weight intervals, (a_i,b_j,c_k). This follows since $a_i < a_i'$, $b_j < b_j'$ and $c_j < c_j'$. Similarly, each triple in T_2 is uniquely determined by the three *right* endpoints of its weight intervals, (a_i',b_j',c_k').

To prove that $|T_1| \leq 3c^2$, let $(A_i,B_j,C_k) \in T_1$ and consider any (A_ℓ,B_p,C_q) with $a_i < a_\ell$, $b_j < b_p$ and $c_k < c_q$. Because of the way we partitioned A,B,C, we must have $a_i' \leq a_\ell$, $b_j' \leq b_p$ and $c_k' \leq c_q$. Hence

$$a_i + b_j + c_k \leq K < a_i' + b_j' + c_k' \leq a_\ell + b_p + c_q,$$

and therefore $(A_\ell,B_p,C_q) \notin T_1$. That is, no triple in T_1 is strictly dominated by another one. By Lemma 3.1 there are at most $3c^2$ triples in T_1. The argument for T_2 is symmetric.

The time recurrence has the form:

$$T(n) \leq 6c^2 T(n/c) + dc^3 (n/c)^\omega,$$

for a constant d. By a technical analysis (the proof is in the full version), c can be chosen (depending on ω) so that the recurrence solves to $T(n) \leq O(n^\omega) + n^2 2^{O(\sqrt{\log n})}$. □

Theorem 3.2 can be viewed as an efficient reduction from counting node-weighted triangles to counting unweighted triangles. However the reduction does not preserve the sparsity of the original graph, and hence a very good algorithm for counting/finding triangles in a sparse unweighted graph does not necessarily imply an algorithm with a comparable running time[3]. Furthermore, because of the sorting, the method used in Theorem 3.2 would require *linear* space to solve weighted triangle problems in truly subcubic time, even if triangle finding can be done in $n^{o(1)}$ space and $O(n^{3-\varepsilon})$ time. This means that the reduction from counting or finding weighted H-subgraphs to counting or finding triangles would require $n^{\Omega(k)}$ space. To resolve these issues, we give a completely different construction which reduces the problem of finding a weighted triangle to a small number of instances of finding unweighted triangles in graphs with the same number of nodes and edges.

We first observe that all versions of the weighted triangle existence problems can be reduced to the K-weight (exact weight) case with a $\mathsf{poly}(\log W)$ runtime overhead, where W is the maximum weight in the graph. Hence we can concentrate on the exact weight case. The proof of Theorem 3.3 is in the full version of the paper.

THEOREM 3.3. *Suppose there is a $T(m,n,W)$ time and $S(m,n,W)$ space algorithm which determines if there is a triangle of weight 0, in an node or edge weighted graph on n nodes and m edges with maximum weight W. Then given an n node m edge graph with node or edge weights at most W, there are $O(S(m,n,W))$ space algorithms for finding a triangle of weight at least K (for any K) and for finding a maximum weight triangle, with time $O(T(m,n,W)\log W)$ and $O(T(m,n,W)\log^2 W)$, respectively.*

Our goal is to prove the theorem:

THEOREM 3.4. *Suppose there is a $T(m,n)$ time, $S(m,n)$ space algorithm which finds a triangle in an n-node, m-edge graph. Then there is a $T(m,n)\cdot 2^{O(\sqrt{\log W})}$ time, $O(S(m,n))$ space algorithm which finds a K-weight triangle (for any K) in an n node m edge graph with node weights in $[1,W]$.*

Our method relies on the existence of a good 3 party communication protocol for Exactly-W. Exactly-W is the multiparty communication problem where $w \in [W]$ is known to all parties, the ith party has an integer $n_i \in [1,W]$ on its forehead, and all wish to determine if $\sum_i n_i = w$. This problem was defined by Chandra, Furst, and Lipton [11]. They showed that Exactly-W has three-party communication complexity $O(\sqrt{\log W})$, but they did not give an effectively computable version of their protocol. In the full version of the paper, we show how to modify the protocol to run in polynomial time, with $O(\log W)$ public bits of randomness and $O(1)$ communication. Although the number of random bits is large, fortunately the error probability is good enough that we can apply this protocol to obtain a fast exact triangle algorithm. A crucial aspect of the protocol is that it does not have false positives: if the sum is not w then it always rejects.

THEOREM 3.5. *Exactly-W has a simultaneous randomized three-party protocol with $O(1)$ communication complexity, where each party runs a $\mathsf{poly}(\log W)$ time algorithm and has access to $2\log W$ public random bits. In particular, consider an instance $(w_1,w_2,w_3) \in [W]$ of Exactly-W for three parties. If $w_1+w_2+w_3 = w$ then the protocol* accepts *with probability at least $1/2^{\Omega(\sqrt{\log W})}$, and if $w_1+w_2+w_3 \neq w$ then the protocol always* rejects.

[3]We note that an $O(m^{1.41})$ runtime for counting weighted triangles is possible using the high degree/low degree method of Alon, Yuster, and Zwick [5].

Proof of Theorem 3.4. Let $G=(V,E)$ be a given graph. Let $V=\{v_1,\ldots,v_n\}$ and let G have weights $w: V \to \{1,\ldots,W\}$. We run the following algorithm an expected $2^{O(\sqrt{\log W})}$ number of times. Pick $2\log W$ bits at random. Let $B=O(1)$ be the communication complexity of the simultaneous protocol in Theorem 3.5. The algorithm cycles over every possible sequence b_1,b_2,b_3, where $b_j \in \{0,1\}^*$ and $|b_j| \leq B$ for $j \in \{1,2,3\}$. These sequences represent all possible simultaneous communications that could take place. Note the number of sequences is $O(1)$.

Given a possible communication sequence S, we *implicitly* construct a graph G'_S on $3n$ nodes. G'_S is tripartite with vertex partitions $V^1=\{v_1^1,\ldots,v_n^1\}, V^2=\{v_1^2,\ldots,v_n^2\}, V^3=\{v_1^3,\ldots,v_n^3\}$. For $i \in \{1,2,3\}$ and $k,\ell \in [n]$, there is an edge between v_k^i and v_ℓ^{i+1} (the indexing is done $\mod 3$) iff (1) $(v_k,v_\ell) \in E$ and (2) the ith party accepts while holding $w(v_k)$ and $w(v_\ell)$ and viewing communication sequence S.

If one finds a triangle in G'_S then the corresponding triangle in G has weight K, by the correctness of the protocol. If there is a K-weight triangle in G, then with constant nonzero probability there is a triangle in G'_S for some S, after $2^{O(\sqrt{\log W})}$ runs of the algorithm. We do not need to construct any of the graphs G'_S, rather every time we need to check whether an edge (v_j^i, v_k^{i+1}) is in G'_S, we run the protocol of Theorem 3.5 for party i in $\mathsf{poly}(\log W)$ time, making sure it matches S. □

4. HARDNESS FOR EDGE WEIGHTED SUBGRAPHS

The methods for finding weighted triangles described in the previous section still fail in the edge weighted case. No truly subcubic algorithms are known for finding a (maximum / at least K / K-weight) triangle in an *edge-weighted* graph. Finding a maximum weight triangle in truly subcubic time has received recent attention (e.g. [41]) due to its connection to all pairs shortest paths (APSP): the distance product (a.k.a. $(\min,+)$-product) of two matrices can be used to find for every pair of nodes the minimum weight of a triangle going through them. Understanding the hardness of finding edge-weighted triangles could explain why it seems so difficult to obtain an $O(n^{3-\varepsilon})$ algorithm for APSP in n node graphs. In this section we relate the edge-weighted triangles to 3SUM and the multivariate quadratic equations problem. We say that a triangle in an edge-weighted graph has K-edge-weight if the sum of its edge weights is K.

4.1 3SUM

First we show a connection between finding K-edge-weight triangles and the 3SUM problem, which is widely believed to have no truly subquadratic algorithm (cf. [21]). In particular, if the K-edge-weight triangle problem can be solved in $O(n^{2.5-\varepsilon})$ time then 3SUM is solvable in $O(n^{2-\varepsilon'})$ time. (Recall that in the *node-weighted* case of the previous section, we obtained an $O(n^\omega)$ solution.) Therefore if one can use an algorithm for the distance product to find an exact edge

weighted triangle in the same time, then APSP requires essentially $\Omega(n^{2.5})$, unless 3SUM can be solved in subquadratic time.[4] Such a conclusion would be intriguing, especially since the decision tree complexity of APSP is $O(n^{2.5})$ ([20]).

THEOREM 4.1. *If for some $\varepsilon > 0$ there is an $O(n^{2.5-\varepsilon})$ algorithm for finding a 0-edge-weight triangle in an n node graph, then there exists a randomized algorithm which solves 3SUM on n numbers in expected $O(n^{\frac{8}{5}} + n^{2-\frac{4}{5}\varepsilon})$ time.*

PROOF. Suppose we are given an instance (A, B, C) of 3SUM so that A, B and C are sets of n integers each. We first use a hashing scheme given by Dietzfelbinger [17] and used by Baran, Demaine and Patrascu [7] which maps each distinct integer independently to one of n/m buckets where m is a parameter we will choose later [5]. For each $i \in [n/m]$, let A_i, B_i, and C_i be the sets containing the elements hashed to bucket i. The hashing scheme has two nice properties:

1. for every pair of buckets A_i and B_j there are two buckets $C_{k_{ij0}}$ and $C_{k_{ij1}}$ (which can be located in $O(1)$ time given i, j) such that if $a \in A_i$ and $b \in B_j$, then if $a + b \in C$ then $a + b$ is in either $C_{k_{ij0}}$ or $C_{k_{ij1}}$,
2. the number of elements which are mapped to buckets with at least $3m$ elements is $O(n/m)$ in expectation.

After the hashing we process all elements that get mapped to large buckets (size $> 3m$). Suppose $a \in A$ is such an element (WLOG it is in A). Then go through all elements b of B and check whether $a + b \in C$. This takes $O(n^2/m)$ time overall in expectation.

Now the buckets A_i, B_i, C_i for all $i \in [n/m]$ contain $O(m)$ elements each. For each set $S \in \{A_i, B_i, C_i\}_{i\in[n/m]}$ we pick an arbitrary ordering of the elements and then denote by $S[k]$ the kth element in S.

For every $c \in [2m]$ we will create an instance of 0-edge-weight triangle as follows. For every $i \in [n/m]$, create nodes x_i and y_j. For every $r \in [m]$ and $s \in [m]$ create a node z_{st}. Add an edge (x_i, z_{st}) for every i, s, t with weight $A_i[s]$. Add an edge (z_{st}, y_j) for every j, s, t with weight $B_j[t]$. For every i, j add an edge (x_i, y_j) with weight $C_{k_{ij0}}[c]$ if $c \leq m$ and $C_{k_{ij1}}[c - m]$ if $c > m$.

If in any one of the $2m$ instances there is a triangle of edge weight 0, then this triangle has the form x_i, z_{st}, y_j and hence $A_i[s], B_j[t], C_{k_{ij}}[c]$ for some c is a solution to the 3SUM instance. Suppose on the other hand that there is a solution a, b, d of the 3SUM instance. Either we found this solution during the reduction to 0-edge-weight triangle, or $a = A_i[k]$, $b = B_j[\ell]$ and $d = C_{k_{ijb}}[p]$, for $b \in \{0, 1\}$, $k, \ell, p \in \{1, \ldots, m\}$, and $i, j \in \{1, \ldots, n/m\}$. Then consider instance $c = bm+p$. The triangle $x_i, y_j, z_{k\ell}$ has weight $A_i[k]+B_j[\ell]+C_{k_{ijb}}[p] = a + b + d$.

Each graph has $n/m + m^2$ nodes and can be constructed in $O(n^2/m^2 + m^4)$ time. The entire reduction takes $O(n^2/m + m^4)$ expected time. By setting $m = \Theta(n^{1/3})$ we obtain $O(n^{1/3})$ instances of 0-edge-weight triangle, each on $O(n^{2/3})$ nodes. Hence if a 0-edge-weight triangle in an N node graph can be found in $O(N^{2.5-\varepsilon})$ time, then 3SUM is in $O(n^{5/3} + n^{\frac{1}{3}+\frac{(2.5-\varepsilon)2}{3}}) = O(n^{5/3} + n^{2-\varepsilon\frac{2}{3}})$ time.

We note that the reduction can be improved slightly by instead of creating $n^{1/3}$ instances of size $n^{2/3}$ we create one instance of size $n^{4/5}$. Then an $O(n^{2.5-\varepsilon})$ algorithm for 0-edge-weight triangle would imply an $O(n^{8/5} + n^{2-\frac{4}{5}\varepsilon})$ algorithm for 3SUM. To do this, for each i and j create $2\sqrt{m}$ copies x_{ic} and y_{jc} (for $c = 1, \ldots, 2\sqrt{m}$) of the original nodes x_i and y_j. For each c and each node z_{st} the weight of edge (x_{ic}, z_{st}) is $A_i[s]$ and that of edge (y_{jc}, z_{st}) is $B_j[t]$. Now there are $4m$ instances $(x_{ic}, y_{jc'})$ of the original edge (x_i, y_j) and we can place the $2m$ numbers of $C_{k_{ij0}} \cup C_{k_{ij1}}$ on these edges so that each number appears at least once. Now we have one instance on $O(n/\sqrt{m} + m^2)$ nodes created in $O(n^2/m + m^4)$ time. By setting $m = n^{2/5}$ we obtain the result. □

[4] We note that Patrascu's recent modification[33] of our reduction implies an $\Omega(n^3)$ hardness for APSP.

[5] The scheme performs multiplications with a random number and some bit shifts hence we require that these operations are not too costly. We can ensure this by first mapping the numbers down to $O(\log n)$ bits, e.g. by computing modulo some sufficiently large $\Theta(\log n)$ bit prime.

4.2 Multivariate Quadratic Equations

Finally, we show that faster algorithms for finding edge-weighted triangles would also imply faster algorithms for NP-hard problems. In particular, a better algorithm for exact edge-weighted triangle over finite fields could be used to solve MULTIVARIATE QUADRATIC EQUATIONS (abbreviated as MQS) faster than exhaustive search. An instance of MQS consists of a set of m equations over n variables that take values from a finite field F, where each equation is of the form

$$p(x_1, \ldots, x_n) = 0$$

for a degree-two polynomial p. The task is to find an assignment $(x_1, \ldots, x_n) \in F^n$ that satisfies all equations.

Several very important cryptosystems have been designed under the assumption that MQS is intractable even in the average case (e.g. [28, 32]). A faster algorithm for MQS would help attack these.

To our knowledge, there are no known algorithms for MQS that improve significantly on exhaustive search in the worst case, though some practical algorithms suggest that MQS may have such an algorithm [26, 15]. We show that an better worst-case algorithm for MQS *does* exist, if edge-weighted triangle (or even k-clique) can be solved faster. More precisely, in the F-WEIGHT k-CLIQUE problem, we are given an edge-weighted undirected graph with weights drawn from a finite field F of $2^{\Theta(b)}$ elements, and are asked if there is a k-clique whose total sum of edge weights is zero over F. We consider the hypothesis that this problem can be solved faster than brute-force search. Observe the trivial algorithm can be implemented to run in $O(b \cdot n^k)$ time.

HYPOTHESIS 4.2. *There is a $\delta \in (0, 1)$ and some $k \geq 3$ such that F-WEIGHT k-CLIQUE is in $O(\mathsf{poly}(b) \cdot n^{\delta k})$ time over a field F of $2^{\Theta(b)}$ elements.*

THEOREM 4.3. *Hypothesis 4.2 implies that MQS over a field F on n variables has an algorithm running in $O(|F|^{\delta n})$ time, for some $\delta < 1$.*

In the following paragraphs we establish Theorem 4.3. The idea is to reduce MQS to the problem of determining whether a sum of degree-two polynomials has a zero solution, then reduce that problem to edge-weighted k-clique. Our reduction

is very similar to known algorithms for MAX CUT and MAX 2-SAT [42], so we only describe it briefly here.

Let $p_1 = 0, \ldots, p_m = 0$ be an instance of MQS. Let $F = GF(p^\ell)$ for some prime p and positive integer ℓ. Let $K = GF(p^{\ell m})$. Treat K as an m-dimensional vector space over F, and let $\mathbf{e_1}, \ldots, \mathbf{e_m}$ be a basis for this space. Define a polynomial $P : F^n \to K$ as

$$P(x_1, \ldots, x_n) := \sum_{i=1}^{m} \mathbf{e_i} p_i(x_1, \ldots, x_n).$$

The following is immediate from the representation of K as a vector space over F. Let $(a_1, \ldots, a_n) \in F^n$.

CLAIM 4.4. *$P(a_1, \ldots, a_n) = 0$ (over K) $\iff$ for all $i = 1, \ldots, m$, $p_i(a_1, \ldots, a_n) = 0$ (over F).*

Hence we have reduced the original problem to that of finding an assignment $a \in F^n$ satisfying $P(a) = 0$ over K. It remains to show that this problem can be reduced to F-WEIGHT k-CLIQUE so that an $O(\mathsf{poly}(b)n^{\delta k})$ algorithm for F-WEIGHT k-CLIQUE translates to an $O(\mathsf{poly}(m,n)|F|^{\delta n})$ algorithm for MQS. Briefly, the reduction works by

- splitting the set of variables into k parts and listing the $|F|^{n/k}$ partial assignments for each part,
- building a complete k-partite graph on $k|F|^{n/k}$ nodes, where the nodes correspond to partial assignments, and
- putting weights (from the field K) on edges $\{u, v\}$ corresponding to the sum of those monomials in P whose variable are assigned by the partial assignments u and v. Here we need to assign degree-one terms via some convention so that we do not overcount the degree-one terms of P.

Find a k-clique with 0 edge weight, when evaluated over K. Note $|K| \leq |F|^m \,\mathsf{poly}(n)$, so the hypothesis entails that this clique problem is in $O(\mathsf{poly}(m,n)|F|^{\delta n})$ time.

5. OPEN PROBLEMS

We conclude with three interesting open problems related to this work.

- Is there a $f(k) \cdot n^{k(1/2-\varepsilon)}\,\mathsf{poly}(n)$ time algorithm for $\#k$-MATCHING for some constant $\varepsilon > 0$ and some function f only depending on k?
- Can one use a fast distance product algorithm to obtain a fast algorithm for finding a 0-edge-weight triangle?
- Is there any way to find triangles fast without recourse to matrix multiplication?

6. ACKNOWLEDGEMENTS

We thank Yiannis Koutis for calling our attention to the reference [25]. We are also grateful to Noga Alon for some helpful discussions.

7. REFERENCES

[1] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):241–249, 2008.

[2] N. Alon and S. Gutner. Balanced families of perfect hash functions and their applications. In *Proc. ICALP*, pages 435–446, 2007.

[3] N. Alon and M. Naor. Derandomization, witnesses for boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16:434–449, 1996.

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

[5] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.

[6] V. Arvind and V. Raman. Approximation algorithms for some parameterized counting problems. In *Proc. ISAAC*, pages 453–464, 2002.

[7] I. Baran, E. Demaine, and M. Patrascu. Subquadratic algorithms for 3SUM. *Algorithmica*, 50(4):584–596, 2008.

[8] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proc. ACM SIGKDD*, pages 16–24, 2008.

[9] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets Möbius: fast subset convolution. In *Proc. STOC*, pages 67–74, 2007.

[10] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. The fast intersection transform with applications to counting paths. *CoRR*, abs/0809.2489, 2008.

[11] A. K. Chandra, M. L. Furst, and R. J. Lipton. Multi-party protocols. In *Proc. STOC*, pages 94–99, 1983.

[12] S. S. Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagl, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS – a graph based intrusion detection system for large networks. In *Proc. 19th National Information Systems Security Conference*, pages 361–371, 1996.

[13] N. Chiba and L. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14:210–223, 1985.

[14] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9(3):251–280, 1990.

[15] N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Proc. EUROCRYPT*, pages 392–407, 2000.

[16] A. Czumaj and A. Lingas. Finding a heaviest triangle is not harder than matrix multiplication. In *Proc. SODA*, pages 986–994, 2007.

[17] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proc. STACS*, pages 569–580, 1996.

[18] J. Flum and M. Grohe. The parameterized complexity of counting problems. *SIAM J. Comput.*, 33(4):892–922, 2004.

[19] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer-Verlag New York, Inc., Secaucus, NJ,

USA, 2006.

[20] M. L. Fredman. On the decision tree complexity of the shortest path problems. In *Proc. FOCS*, pages 98–99, 1975.

[21] A. Gajentaan and M. Overmars. On a class of $o(n^2)$ problems in computational geometry. *Computational Geometry*, 5(3):165–185, 1995.

[22] B. Gelbord. Graphical techniques in intrusion detection systems. In *Proc. 15th International Conference on Information Networks*, pages 253–238, 2001.

[23] Y. Gurevich and S. Shelah. Expected computation time for hamiltonian path problem. *SIAM J. Comput.*, 16(3):486–502, 1987.

[24] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Computing*, 7(4):413–423, 1978.

[25] T. Kawabata and J. Tarui. On complexity of computing the permanent of a rectangular matrix. *IECIE Trans. on Fundamentals of Electronics*, 82(5):741–744, 1999.

[26] A. Kipnis and A. Shamir. Cryptanalysis of the HFE public key cryptosystem by relinearization. In *Proc. CRYPTO*, volume 1666, pages 19–30, 1999.

[27] T. Kloks, D. Kratsch, and H. Müller. Finding and counting small induced subgraphs efficiently. *Inf. Proc. Letters*, 74(3-4):115–121, 2000.

[28] S. Landau. Polynomials in the nation's service: using algebra to design the advanced encryption standard. *American Mathematical Monthly*, 111:89–117, 2004.

[29] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[30] H. Minc. *Permanents*. Cambridge University Press, New York, NY, USA, 1984.

[31] J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Commentationes Math. Universitatis Carolinae*, 26(2):415–419, 1985.

[32] J. Patarin. Cryptoanalysis of the matsumoto and imai public key scheme of eurocrypt'88. In *Proc. Annual Cryptology Conference (CRYPTO)*, pages 248–261, 1995.

[33] M. Patrascu. Personal communication.

[34] F. Romani. Shortest-path problem is not harder than matrix multiplication. *Information Processing Letters*, 11:134–136, 1980.

[35] H. Ryser. *Combinatorial mathematics*. Wiley & Math. Assoc. Amer., 1963.

[36] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*, pages 606–609, 2008.

[37] V. Sekar, Y. Xie, D. A. Maltz, M. K. Reiter, and H. Zhang. Toward a framework for internet forensic analysis. In *Third Workshop on Hot Topics in Networking (HotNets-III)*, 2004.

[38] G. Sundaram and S. S. Skiena. Recognizing small subgraphs. *Networks*, 25:183–191, 1995.

[39] C. E. Tsourakakis. Fast counting of triangles in large real networks, without counting: Algorithms and laws. In *Proc. IEEE ICDM*, volume 14, 2008.

[40] V. Vassilevska and R. Williams. Finding a maximum weight triangle in $n^{3-\delta}$ time, with applications. In *Proc. STOC*, pages 225–231, 2006.

[41] V. Vassilevska, R. Williams, and R. Yuster. Finding the smallest H-subgraph in real weighted graphs and related problems. In *Proc. ICALP*, volume 4051, pages 262–273, 2006.

[42] R. Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2–3):357–365, 2005.

[43] G. Yuval. An algorithm for finding all shortest paths using $N^{2.81}$ infinite-precision multiplications. *Inf. Proc. Letters*, 4:155–156, 1976.