Finding, Minimizing, and Counting Weighted Subgraphs[*†]

Virginia Vassilevska Williams[‡] Ryan Williams[§]

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 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 $\Omega(n^{3-\varepsilon})$ 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.

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 [16] and runs in $\Omega(n^{\omega k/3}) \geq \Omega(n^{0.791k})$ time and $n^{\Theta(k)}$ space, where $\omega < 2.376$ is the smallest exponent for which $n \times n$ matrix multiplication is in $n^{\omega+o(1)}$ time. 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 $O(s! \cdot n^{k-s} n^2)$ time, or in $O(2^s n^{k-s} n^3)$ time and space. Our polynomial space algorithms can also 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, 45, 42], computational biology, and network security [14, 25, 43]. In molecular biology, biomolecular

[*]The first author is currently supported by NSF Grant 0937060 to the Computing Research Association for the CIFellows Project. The second author is currently supported by the Josef Raviv Memorial Fellowship. This material is based in part on work done while the authors were members of the Institute for Advanced Study in Princeton. The authors were 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.

[†]A preliminary version of this paper appeared in the proceedings of STOC 2009.

[‡]University of Calfornia, Berkeley, Berkeley, CA 94720 USA Email: virgi@cs.berkeley.edu.

[§]IBM Almaden Research Center, San Jose, CA 95120 USA. Email: ryanwill@us.ibm.com.

networks are compared by identifying so-called *network motifs* [33] – 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 at least 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, 22, 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. 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 (APSP), the maximum weight version has received much attention (e.g. [46, 47, 19]). The authors have recently shown that APSP and the related maximum weight triangle problem in edge-weighted graphs are equivalent under subcubic reductions [48] so that maximum weight triangle has a truly subcubic algorithm if and only if all pairs shortest paths does.

The current best algorithm for *finding* a maximum weight H-subgraph in a *node-weighted* graph is by Czumaj and Lingas [19] 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 [46] 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 in $O(n^k)$ time 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. Just as the maximum weight triangle problem in edge-weighted graphs, this exact weight triangle problem is not known to have a truly subcubic algorithm, even though there is an extremely simple cubic time algorithm for it. In an attempt to explain this, we relate the exact-weight triangle problem to the 3SUM problem: given a list of n integers, determine whether there are three of them that sum up to 0. We first 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 [38] has recently proven a tight result relating 3SUM to a related problem, Convolution–3SUM. We show that using this relationship, our conditional lower bound for exact weighted triangles can be improved optimally, i.e. unless 3SUM has truly subquadratic algorithms, finding a triangle of weight 0 in an edge-weighted graph requires essentally 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 [27, 36, 15, 30, 44]. 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 seem to generalize for $k > 7$. Björklund et al. [10] have recently found an interesting algorithm for counting k-paths that runs in $\binom{n}{k/2}\text{poly}(n)$ 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 $\text{poly}(n, k)$ 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. A k-path in a graph is a sequence of k distinct vertices, every two consecutive ones of which are connected by an edge. A DAG is a directed graph with no cycles.

As usual, we define ω as the infimum of all real numbers such that $n \times n$ multiplication can be solved in $n^{\omega+o(1)}$ arithmetic operations. The current best upper bound on ω is 2.376 [16].

We say an algorithm on $n \times n$ matrices with entries in $[-M, M]$ (or n-node graphs with edge weights from $[-M, M]$) is *truly subcubic* if it runs in $O(n^{3-\delta} \cdot \text{poly}(\log M))$ time for some $\delta > 0$.

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*

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

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 in G. It discards T if the map $h_i \mapsto v_i$ for $i \in [t]$ is not a homomorphism. Note the number of choices for T is $t! \cdot \binom{n}{t}$.

Now suppose that $h_i \mapsto v_i$ is a homomorphism. 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.

[1]In the case where H is a k-path and G is unweighted, note that 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.

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 some $d' \leq (k-s)$). 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 $(k-s)!\binom{k}{k-s}T(k,s) \leq (k-s)!\binom{n}{k-s}T(n,s)$ time. $\square$

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

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. $\square$

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 $\mathrm{poly}(n,k)$ *space.*

For small k and s the above running time is $O(n^{3+k-s})$.

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 $k!\binom{n}{k/2}$*$\mathrm{poly}(n)$ *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* $\mathrm{poly}(n,k) + O(n^2 2^s)$ *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* $\mathrm{poly}(n,k)$ *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 that 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 [40]. He gives a formula based on inclusion-exclusion that computes the permanent of an $n \times n$ matrix over a ring in $O(2^n \text{poly}(n))$ time and $O(\text{poly}(n))$ 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 $O(\binom{n}{k}\text{poly}(n))$ time algorithm (this is well-known folklore, cf. [34], Section 7.2). Both of these prevent us from using Ryser's algorithm in the algorithms of the previous section. Kawabata and Tarui [28] 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 [34]. 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 [21]. 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 $2^{k+o(k)}\binom{n}{k/2}\text{poly}(n)$ 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

[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 [39], Yuval [50].

we have already computed the values W_i^u, for all nodes u 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 $perm^*(A_\pi) = \sum_v W_k^v$. $\square$

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

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

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

Let $\mathcal{L}$ be the set of sets L with $L \subseteq [k]$ and $|L| = \lfloor k/2 \rfloor$. The following identity is the key to the algorithm.

Claim 2.9

$$perm(A) = \sum_{L \in \mathcal{L}} \sum_{1 \leq j_2 < j_3 \leq n} B_L^{1,j_2} \cdot C_{|k|-L}^{j_3,n}.$$

Proof: Note that on the right hand side, we have $\sum_{L \in \mathcal{L}} \sum_{1 \leq j_2 < j_3 \leq n} B_L^{1,j_2} \cdot C_{|k|-L}^{j_3,n} =$

$$\sum_{\substack{L \in \mathcal{L} \\ 1 \leq j_2 < j_3 \leq n \\ \ell \in L, \ell' \in [k]-L}} A[\ell, j_2] \cdot A[\ell', j_3] \cdot \left(\sum_{\substack{f:(L \setminus \{\ell\}) \to \{1,\ldots,j_2-1\} \\ f \text{ is 1-1}}} \left(\prod_{i \in L \setminus \{\ell\}} A[i, f(i)] \right) \right) \cdot \left(\sum_{\substack{f:([k] \setminus L \cup \{\ell'\}) \\ \to \{j_3+1,\ldots,n\} \\ f \text{ is 1-1}}} \left(\prod_{i \in [k] \setminus L \cup \{\ell'\}} A[i, f(i)] \right) \right).$$

By distributivity, this sum is

$$= \sum_{L \in \mathcal{L}} \sum_{\substack{1 \leq j_2 < j_3 \leq n \\ \ell \in L, \ell' \in [k]-L}} \left(\sum_{\substack{f:L-\{\ell\} \to \{1,\ldots,j_2-1\} \\ f:([k]-L-\{\ell'\}) \to \{j_3+1,\ldots,n\} \\ f(\ell)=j_2, f(\ell')=j_3, f \text{ is 1-1}}} \prod_i A[i, f(i)] \right) = \sum_{L \in \mathcal{L}} \left(\sum_{\substack{f:[k] \to \{1,\ldots,n\} \text{ is 1-1} \\ \forall i \in L, j \notin L \; f(i) < f(j)}} \left(\prod_{i=1}^{k} A[i, f(i)] \right) \right).$$

But every one-to-one f fits the condition under the inner sum, for exactly one L of size $\lfloor k/2 \rfloor$. So the above is just

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

More generally, for all i, j, $perm(A^{i,j}_{[k]})$ can be expressed as a sum of products of permanents $perm(A^{i,j_2}_{L-\{\ell\}}) \cdot perm(A^{j_3,j}_{[k]-L-\{\ell'\}})$. This completes the proof. $\square$

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 $M[i, j] = perm(A^{i,j}_{[k]})$. Hence $M[1, n] = perm(A)$.

```
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 $|L| = \lfloor k/2 \rfloor$:
        $B_L, C_L :=$ the $n \times n$ matrix of all zeroes
        For all $\ell \in L$:
            $M_{L-\{\ell\}} :=$ PERMANENT($A^{1,n}_{L-\{\ell\}}$).
            For all $i, j_2 \in [n]$:
                $B_L[i, j_2] := B_L[i, j_2] + A[\ell, j_2] \cdot M_{L-\{\ell\}}[i, j_2 - 1]$.
        For all $\ell' \in [k] - L$:
            $M_{[k]-L-\{\ell'\}} :=$ PERMANENT($A^{1,n}_{[k]-L-\{\ell'\}}$).
            For all $j_3, j \in [n]$:
                $C_L[j_3, j] := C_L[j_3, j] + A[\ell', j_3] \cdot M_{[k]-L-\{\ell'\}}[j_3 + 1, j]$.
        $M' := n \times n$ matrix
        For all $i, j \in [n]$:
            $M'[i, j] := \sum_{j_2, j_3: i \le j_2 < j_3 \le j} B_L[i, j_2] \cdot C_L[j_3, j]$
        $M := M + M'$.
    Return $M$.
```

The correctness of PERMANENT follows from Claim 2.9. A naive way to construct the matrix 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 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

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

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. $\square$

We remark that Gurevich and Shelah [26] gave a $4^n \text{poly}(n)$ 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 [28]. We require a lemma which is a simple extension of the fast subset convolution of Bjorklund et al. [12].

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 $O(K2^N)$ time and space.

Proof: For any subset $X \subseteq [N]$ of size $\leq K$, define

$$X^0 = \{T \subseteq X \mid |T| = \lfloor K/2\rfloor\},\ \text{and}\ X^1 = \{T \subseteq X \mid |T| = \lceil K/2\rceil\},$$

$$\hat{f}(X) = \sum_{T\in X^0} f(T), \text{and}\ \hat{g}(X) = \sum_{T\in X^1} g(T).$$

Let $S \subseteq [N], |S| = K$. Consider

$$\begin{aligned}
\sum_{X\subseteq S}(-1)^{|S\setminus X|}\cdot\hat{f}(X)\cdot\hat{g}(X) &= \sum_{X\subseteq S}(-1)^{|S\setminus X|}\cdot\sum_{U\in X^0,T\in X^1} f(U)g(T) = \\
&= \left(\sum_{\substack{U\in S^0,T\in S^1\\ U\cap T=\emptyset}}(-1)^0 f(U)g(T)\right)+\sum_{r=1}^{K}\left(\sum_{\substack{U\in S^0,T\in S^1\\ |U\cap T|=r}} f(U)g(T)\sum_{i=0}^{r}\binom{r}{i}(-1)^{r-i}\right) = \\
&= \sum_{U\in S^0} f(U)g(S\setminus U)+\sum_{r=1}^{K}\sum_{\substack{U\in S^0,T\in S^1\\ |U\cap T|=r}} f(U)g(T)\cdot(-1+1)^r = \sum_{U\in S^0} f(U)g(S\setminus U).
\end{aligned}$$

Let $\hat{h}$ be the function with $\hat{h}(X) = \hat{f}(X)\cdot\hat{g}(X)$ for all $X \subseteq [n]$, $|X| \leq K$.

To compute $h(S) = \sum_{U\in S^0} f(U)g(S\setminus U)$ for all $S \subseteq [N]$ with $|S| = K$ in $O(K2^N)$ time, it suffices to be able to compute

1. $\hat{f}(X), \hat{g}(X)$ for all $X \subseteq [N]$ and $|X| \leq K$ in $O(K2^N)$ time, and
2. given all $\hat{h}(X)$, $\sum_{X\subseteq S}(-1)^{|S\setminus X|}\hat{h}(X)$ in $O(K2^N)$ time.

We will first show how to compute $\hat{f}(X)$ in $O(K2^N)$ time: Iterate over all s from 1 to N and over all sets X of size $\leq K$ containing s. For each fixed s and X, let s' be the largest element of X smaller than s, and set

$$\hat{f}_s(X) = \hat{f}_{s'}(X\setminus\{s\}) + \hat{f}_{s'}(X).$$

We initialize for each $X \subseteq [N]$

$$\hat{f}_0(X) = \begin{cases} f(X) & \text{if } |X| = \lfloor K/2\rfloor \\ 0 & \text{otherwise.}\end{cases}$$

The runtime of this procedure is $O(K\binom{N}{K}) = O(K2^N)$ as every set X is accessed at most $|X|+1$ times and $|X| \leq K$. The final result is $\hat{f}(X) = \hat{f}_N(X)$. Computing $\hat{g}(X)$ is analogous. To obtain $\hat{h}(S)$, simply go through all S in $O(2^N)$ time and set $\hat{h}(S) = \hat{f}(S)\cdot\hat{g}(S)$.

Computing $\sum_{X\subseteq S}(-1)^{|S\setminus X|}\hat{h}(X)$ given all $\hat{h}(X)$ proceeds similarly to above: Iterate over all s from 1 to N and over all sets X of size $\leq K$ containing s. For each fixed s and X, let s' be the largest element of X smaller than s, and set

$$\hat{h}_s(X) = -\hat{h}_{s'}(X\setminus\{s\}) + \hat{h}_{s'}(X).$$

We initialize for each $X \subseteq [N]$, $\hat{h}_0(X) = \hat{h}(X)$. The runtime of this procedure is also $O(K2^N)$ as every set X is accessed at most $|X|+1$ times and $|X| \leq K$. The final result is $\hat{h}_N(S)$. $\square$

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 $perm(A_T^{j_1,j_2})$ 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 $\lfloor k/2^i \rfloor, \lfloor k/2^i \rfloor - 1, \lfloor k/2^i \rfloor - 2$, $B_L^{j_1,j_2}$ and $C_L^{j_1,j_2}$ can be computed in $O(kn^2 2^k/2^i)$ time. Consider $A_S^{j_1,j_2}$ 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:

$$perm(A_S^{j_1,j_2}) = \sum_{\substack{L \subseteq S \\ |L| = \lfloor |S|/2 \rfloor}} \sum_{j_1 \le p_2 < p_3 \le j_2} B_L^{j_1,p_2} \cdot C_{S-L}^{p_3,j_2}.$$

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 $B_L^{j_1,p_2}$ and $C_L^{p_3,j_2}$ 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 $perm(A_S^{j_1,j_2})$ 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. $\square$

Subsequent Work Since the conference version of this paper appeared, there has been subsequent work on computing permanents of rectangular matrices. Recently, Bjorklund et al. [11] have obtained several algorithms for computing the permanent of a $k \times n$ matrix over semirings and rings. Let $\binom{n}{\downarrow s} = \sum_{t=0}^{s} \binom{n}{t}$. Bjorklund et al. show that the $k \times n$ permanent can be computed in

- $O(k \cdot \binom{n}{\downarrow k})$ time and $O(\binom{n}{\downarrow k})$ space over any semiring,
- in $O(kn2^k)$ time and space over any commutative semiring,
- in $O(k\binom{n}{\downarrow k/2})$ time, and $O(\binom{n}{\downarrow k/2})$ space over any ring, and
- in $O(kn2^k)$ time and $O(n)$ space over any commutative ring.

Koutis and Williams [31] also give an $O(2^k \text{poly}(n))$ time and space algorithm for the $k \times n$ permanent over commutative semirings, and an $O(2^k \text{poly}(n))$ time and polynomial space algorithm over commutative rings.

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 ([46, 47, 19]). 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 $O(k^2T(n^{k/3}))$ time and $O(S(n^{k/3}))$ space. Here we take a similar approach, and study the corresponding triangle problems.

In previous work [46] 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 [19] 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 [19]) *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 $|U| \le 3n^2$.*

Similarly to Czumaj and Lingas' approach in [19], 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 $n^{\omega+o(1)}$ time, improving on the $O(n^{(3+\omega)/2})$ time solution. The new algorithm immediately implies an $O(n^{\omega k} + n^{(2+\varepsilon)k})$ for every $\varepsilon > 0$ running time for counting weighted subgraph patterns on $3k$ nodes in an n-vertex 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 $n^2 \cdot exp(O(\sqrt{\log n})) + O(n^\omega)$ 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. Sort A, B and C (in nondecreasing order) and partition them into c 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, as follows. If $A = (a_1,\ldots,a_N)$, then for each $0 \le i < c-1$, let $A_{i+1} = (a_{i\lceil N/c\rceil+1},\ldots,a_{(i+1)\lceil N/c\rceil})$. B and C are split similarly. Each new sublist is now associated with the interval of weights between the smallest and largest weights of its nodes.

For each $1 \le i < c$ and each $1 \le k \le \lceil N/c \rceil$, let $A_i[k]$ refer to the k-th element of A_i in the sorted order. Define $B_i[k], C_i[k]$ similarly.

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: $a_i = a'_i$ or $b_j = b'_j$, or $c_k = c'_k$. This case represents the bottom of the recursion. 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 $(A_i[p], C_k[q]) \in E$, the entry $(XY)[p,q]$ 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. Without loss of generality, 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 $(A_i[p], C_k[q]) \in E$, the entry $(XY)[p,q]$ 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: $a_i < a'_i$ **and** $b_j < b'_j$**, 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

$$a_i + b_j + c_k \le K \le a'_i + b'_j + c'_k.$$

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 $a_i+b_j+c_k \le K < a'_i+b'_j+c'_k$ (type 1) and those with $a_i+b_j+c_k < K \le a'_i+b'_j+c'_k$ (type 2). Let T_1 and T_2 be the sets of type 1 and type 2 triples respectively. We show that $|T_i| \le 3c^2$ 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 $|T_1| \le 3c^2$, 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 $a'_i \le a_\ell$, $b'_j \le b_p$ and $c'_k \le c_q$. Hence

$$a_i + b_j + c_k \le K < a'_i + b'_j + c'_k \le a_\ell + b_p + c_q,$$

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:

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

for a constant d.

Czumaj and Lingas [19] showed that a similar recurrence solves to $O(n^\omega + n^{2+\varepsilon})$ for all $\varepsilon > 0$. We improve their analysis by proving that c can be chosen (depending on ω) so that the recurrence solves to $T(n) \le O(n^\omega) + n^2 2^{O(\sqrt{\log n})}$.

If $\omega > 2$, pick any ε with $0 < \varepsilon < \omega - 2$ and set $c = 6^{1/\varepsilon}$ (we can pick ε so that c is an integer). By the master theorem (see e.g. [17]), if $2 + \varepsilon = (\log 6 + 2\log c)/(\log c) < \omega$ and for some $c'' < 1$, $6c^2(n/c)^\omega \le c'' n^\omega$ then the runtime is $O(n^\omega)$. Now $6c^2(n/c)^\omega = 6n^\omega / c^{\omega-2} = n^\omega \cdot 6^{1-\frac{\omega-2}{\varepsilon}}$, and $6^{1-\frac{\omega-2}{\varepsilon}} < 6^{1-1} = 1$. Hence in this case the runtime is precisely $O(n^\omega)$.

Suppose $\omega = 2$. Set $c(n) = 2^{p\sqrt{\log n}}$ for some constant $p > 7$. Let $q = \frac{p}{2} - \varepsilon$ for some $0 < \varepsilon < 1/2$. Let $c'' = \frac{2^q c'}{2^q - 6}$ and notice $c'' > c' > 0$ and $q = \log\left(\frac{6c''}{c'' - c'}\right)$.

For large enough n,

$$q^2 - 2\varepsilon\sqrt{\log n - p\sqrt{\log n}} < 0, \text{ and}$$

$$(\sqrt{\log n - p\sqrt{\log n}} + q)^2 = (\log n - p\sqrt{\log n}) + (p - 2\varepsilon)\sqrt{\log n - p\sqrt{\log n}} + q^2 \le$$

$$\log n + q^2 - 2\varepsilon\sqrt{\log n - p\sqrt{\log n}} < \log n.$$

Hence, $\sqrt{\log n - p\sqrt{\log n}} + q < \sqrt{\log n}$. Substituting $q = \log \frac{6c''}{c''-c'}$, $p\sqrt{\log n} = \log c$ and multiplying by p,

$$\log(6c'') + p\sqrt{\log \frac{n}{c}} < p\sqrt{\log n} + \log(c'' - c').$$

Raising 2 to the power of each side, adding cc' to both sides and then multiplying both sides by n^2, we obtain

$$n^2(6c'' \cdot 2^{p\sqrt{\log(n/c)}} + cc') < c''n^2 2^{p\sqrt{\log n}}.$$

By the induction hypothesis, however, the left hand side is at least $6c^2T(n/c) + c'c^3(n^2/c^2) \geq T(n)$. We get that $T(n) \leq c''n^2 2^{p\sqrt{\log n}}$ and hence by induction our algorithm takes $n^2 \cdot 2^{O(\sqrt{\log n})}$ time. □

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 for node-weighted graphs[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. Note however, that there may be an algorithm for triangle finding which runs in $O(n^{3-\varepsilon})$ time and uses $n^{o(1)}$ space; the trivial $O(n^3)$ time algorithm uses only $O(\log n)$ space. With the reduction in Theorem 3.2 such an algorithm would not translate to a small space subcubic algorithm for node-weighted triangle finding. The linear space usage of the reduction also means that the corresponding 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 poly$(\log W)$ runtime overhead, where W is the maximum weight in the graph. Hence we can concentrate on the exact weight case.

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. Let K be any integer. 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 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.*

Given a B-bit integer x we define $pre_i(x)$, the ith prefix of x, to be the integer obtained from x by removing the last $B-i$ bits of x. We will use $pre_i(x)$ to denote both the prefix integer and its representation as a bit sequence. The proof of Theorem 3.3 relies on Proposition 3.4 below. We use Proposition 3.4 to process the integer weights in a given weighted triangle instance by creating new instances of 0-weight triangle in which the weights are the prefices $pre_i(\cdot)$ for some i, or the prefices concatenated with a single bit, as in $pre_i(x)1$ or $pre_i(x)0$.

Proposition 3.4 *For three integers x, y, z we have that $x + y > z$, if and only if one of the following holds:*

- *there exists an i such that $pre_i(x) + pre_i(y) = pre_i(z) + 1$, or*
- *there exists an i such that $pre_{i+1}(x) = pre_i(x)1$, $pre_{i+1}(y) = pre_i(y)1$, $pre_{i+1}(z) = pre_i(z)0$ and $pre_i(x) + pre_i(y) = pre_i(z)$.*

Proof: If one of the two conditions holds, we clearly have $x + y > z$. We only need to prove the opposite direction. Suppose $x + y > z$. Then there exists some i with $pre_i(x) + pre_i(y) \geq pre_i(z) + 1$. Consider the smallest such i. If $pre_i(x) + pre_i(y) = pre_i(z) + 1$, then we are done, so assume $pre_i(x) + pre_i(y) \geq pre_i(z) + 2$. Then we have that

[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].

$pre_{i-1}(x) + pre_{i-1}(y) \leq pre_{i-1}(z)$. For some $b_1, b_2, b_3 \in \{0, 1\}$, $pre_i(x) = pre_{i-1}(x)b_1$, $pre_i(y) = pre_{i-1}(y)b_2$ and $pre_i(z) = pre_{i-1}(z)b_3$, and

$$2(pre_{i-1}(x) + pre_{i-1}(y)) + 2 \geq 2(pre_{i-1}(x) + pre_{i-1}(y)) + b_1 + b_2 \geq pre_i(z) + 2 =$$

$$2pre_{i-1}(z) + b_3 + 2 \geq 2(pre_{i-1}(x) + pre_{i-1}(y)) + 2 + b_3.$$

Hence we have that $b_3 = 0, b_1 = b_2 = 1$ and $pre_{i-1}(x) + pre_{i-1}(y) = pre_{i-1}(z)$. □

Now we can prove the Theorem:

Proof of Theorem 3.3. The maximum weight triangle problem can be reduced to the $> K$-weight triangle problem just by binary searching for the maximum weight. This incurs an overhead of $\log W$ in the running time and no space overhead. Hence we just reduce the $> K$-weight triangle problem to the 0-weight triangle problem. We assume that we have a $T(m, n, W)$ time, $S(m, n, W)$ space algorithm which can find a triangle of weight exactly 0 if one exists. Given this algorithm it is straightforward to obtain an $O(T(m, n, W))$ time, $O(S(m, n, W))$ algorithm which finds a triangle of weight K for any given K. We will give a procedure which given K, determines whether there is a triangle in the graph of weight sum $> K$. We give the reduction for edge weighted graphs. The node-weighted graph case is similar.

Implicitly build a tripartite graph $G' = (V', E')$ from $G = (V, E)$ by creating three copies V_1, V_2, V_3 of the vertex set V and set $V' = V_1 \cup V_2 \cup V_3$. For every $u \in V$, let u^i denote its copy in V_i. Then $E' = \{(u^i, v^j) \mid (u, v) \in E, i, j \in \{1, 2, 3\}, i \neq j\}$. Let $W = \max_{u,v} |w(u, v)|$. We set up the weights w' for G' as follows: we let $w'(u^1, v^j) = w(u, v) + W$ for $j \in \{2, 3\}$ and $w'(u^2, v^3) = 2W + K - w(u, v)$. Now, a triangle a^1, b^2, c^3 with the property that $w'(a^1, b^2) + w'(a^1, c^3) > w'(b^2, c^3)$ exists in G' iff there is a triangle a, b, c in G of weight sum $> K$. To create a procedure which determines whether there is such a triangle in G' we use Proposition 3.4.

The integers in G' have $O(\log W)$ bits. We create $O(\log W)$ instances of 0–weight triangle as follows. For every prefix $i = 0, \ldots, O(\log W)$ create two instances of 0-weight triangle G_{i1} and G_{i2}.

For every $(u^j, v^k) \in E'$, (u^j, v^k) is also an edge in G_{i1} with weight $pre_i(w'(u^j, v^k))$ if $j = 1$ and $k \in \{2, 3\}$ and with weight $-pre_i(w'(u^j, v^k)) - 1$ otherwise. This corresponds to the first bullet of Proposition 3.4.

For every $(u^j, v^k) \in E'$, (u^j, v^k) is an edge in G_{i2} if $j = 1$ and $pre_{i+1}(w'(u^j, v^k)) = pre_i(w'(u^j, v^k))1$, or if $j = 2, k = 3$ and $pre_{i+1}(w'(u^j, v^k)) = pre_i(w'(u^j, v^k))0$. The weight of (u^j, v^k) in G_{i2} is $pre_i(w'(u^j, v^k))$ if $j = 1$ and $-pre_i(w'(u^j, v^k))$ otherwise. This corresponds to the second bullet of Proposition 3.4.

By Proposition 3.4, there is a triangle with weight $> K$ in G if and only if for some $i \in \{0, \ldots, O(\log W)\}$ and $b \in \{1, 2\}$, the graph G_{ib} contains a 0-weight triangle.

We note that we do not need to explicitly construct the graphs G_{ib}. Whenever the 0-weight triangle algorithm needs to know whether a particular edge exists and (if so) what its weight is, we compute this by checking the adjacency matrix of the original graph. Hence there is no space overhead. □

We now give a tight reduction from node-weighted triangle finding to triangle finding which is both time and space efficient.

Theorem 3.5 *Suppose there is a $T(m, n)$ time, $S(m, n)$ space algorithm which finds a triangle in an n-node, m-edge graph, or determines that the graph is triangle-free. Let K be any integer. Then there is a $T(m, n) \cdot 2^{O(\sqrt{\log W})}$ time, $O(S(m, n))$ space randomized algorithm which finds a K-weight triangle in an n node m edge graph with node weights in $[1, W]$, or determines that such a triangle does not exist, with high probability at least* $1 - 1/\text{poly}(W)$.

Our method relies on the existence of a good three-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 can thus see all n_j but n_i; all parties wish to determine if $\sum_i n_i = w$. This problem was defined by Chandra, Furst, and Lipton [13]. 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. We show how to modify the protocol so that it can be used 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 3.6 *Exactly-W has a simultaneous randomized three-party protocol with $O(\sqrt{\log W})$ communication complexity, where each party runs a $2^{O(\sqrt{\log W})}$ time algorithm and has access to $2^{O(\sqrt{\log W})}$ 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* $1 - 1/\text{poly}(W)$*, and if* $w_1 + w_2 + w_3 \neq w$ *then the protocol always* rejects.

We note that by making the acceptance probability $1/2^{\Omega(\sqrt{\log W})}$, one can easily modify the protocol of Theorem 3.6 so that it has $O(1)$ communication complexity, and each party runs a $\text{poly}(\log W)$ time algorithm and has access to $2\log W$ public random bits.

Proof: Let us survey the protocol in [13], then describe how to adapt it. Their protocol is deterministic, uses $O(\sqrt{\log W})$ bits of communication, and is non-constructive. At the heart of the protocol is a "hash" function $h(x)$ from $[W]$ to $[k]$, where $k << W$, which we now describe. Let $S \subseteq [W]$ be *3-AP-free*, meaning that it contains no arithmetic progression of length three. It is known that for all W, there is a constant $c > 0$ and 3-AP-free set $S \subseteq [W]$ of size $cW/2^{c\sqrt{\log W}}$. Consider a random translate of S, i.e. a set of the form $S_i = \{x + y_i \mid x \in S\}$ for a uniform random $y_i \in \{-W, \ldots, W\}$. The set $[W]$ is covered by $k \leq O(\frac{W}{|S|}\log W) \leq O(2^{c\sqrt{\log W}}\log W)$ random translates $\{S_1, \ldots, S_k\}$ with probability at least $1 - 1/\text{poly}(W)$, where each $|S_i| = |S|$, and each S_i is 3-AP-free. For $x \in [W]$, define $h(x)$ to be the smallest $i \in [k]$ such that $x \in S_i$.

Now we describe the protocol. Each party holds two of the $w_i \in [W]$ (along with knowledge of w) and wants to know if $w_1 + w_2 + w_3 = w$. Observe that each party "knows" what its missing number ought to be if the sum is w, for example the party holding w_1 and w_2 knows that w_3 should be $w - w_1 - w_2$. The ith party computes its own $x_i = w_1 + 2w_2 + 3w_3$, using the difference between w and its two known integers in place of its missing integer, then sends $h(x_i)$ using $O(\log(W/|S|))$ bits. We claim that $w_1 + w_2 + w_3 = w$ if and only if $h(x_1) = h(x_2) = h(x_3)$, so the protocol accepts if and only if all messages are identical. If $w_1 + w_2 + w_3 = w$, then obviously $x_1 = x_2 = x_3$. If $w_1 + w_2 + w_3 \neq w$, we claim that x_1, x_2, x_3 is a three-term arithmetic progression; by construction of S, it follows that not all $h(x_i)$ are equal. Let $\alpha = w - w_1 - w_2 - w_3 \neq 0$. Then $x_1 = (w - w_2 - w_3) + 2w_2 + 3w_3 = (\alpha + w_1) + 2w_2 + 3w_3$, $x_2 = w_1 + 2(w - w_1 - w_3) + 3w_3 = w_1 + 2(\alpha + w_2) + w_3$, $x_3 = w_1 + 2w_2 + 3(\alpha + w_3)$. Letting $a = w_1 + 2w_2 + 3w_3$, we have $x_1 = a + \alpha$, $x_2 = a + 2\alpha$, $x_3 = a + 3\alpha$, a progression.

A possible obstacle to obtaining a constructive version of the protocol from [13] described above is the 3-AP-free set S. Behrend [9] (building on Salem and Spencer [41]) gave the first 3-AP-free S of size $\Omega(W^{1-c/\sqrt{\log W}})$. His proof gives a procedure for generating elements of S, but in the above protocol we need to be able to recognize members of S; it is not clear how to do this quickly with Behrend's set. Fortunately, Moser [35] later gave a 3-AP-free S of comparable size for which membership of a $(\log W)$-bit number x can be determined in $O(\text{poly}(\log W))$ time.

Now suppose that each party i can test membership in S, then they could determine $h(x_i)$ in $2^{O(\sqrt{\log W})}$ time by enumerating through all $2^{O(\sqrt{\log W})}$ random translates S_j. Then they can easily send $\log(2^{O(\sqrt{\log W})}) = O(\sqrt{\log W})$ bits describing $h(x_i)$. Using this version of the protocol the three parties can determine whether $w_1 + w_2 + w_3 = w$, with high probability. The protocol requires $2^{O(\sqrt{\log W})}$ public random bits, $O(\sqrt{\log W})$ bits of communication, and each party runs a $2^{O(\sqrt{\log W})}$ time algorithm. $\square$

Proof of Theorem 3.5. 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. Let $B = O(\sqrt{\log W})$ be the communication complexity of the simultaneous protocol in Theorem 3.6. 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 $2^{O(\sqrt{\log W})}$.

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 3.6 for party i in $2^{O(\sqrt{\log W})}$ time, making sure it matches S. $\square$

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 weighted triangle in an *edge-weighted* graph, for any of the three versions of the problem we have considered. Finding a maximum weight triangle in truly subcubic time has received recent attention (e.g. [46, 47, 19]) due to its connection to all pairs shortest paths (APSP). This connection has recently been made formal by the authors [48]: a truly subcubic algorithm for maximum weight triangle in an edge-weighted graph exists if and only if there is a truly subcubic algorithm for APSP. Thus understanding edge-weighted triangle problems could explain why it seems so difficult to obtain a truly subcubic algorithm for APSP. In this section we relate the exact version of the edge-weighted triangle problem 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. [24]). Our goal is to show that if the K-edge-weight triangle problem can be solved in truly subcubic time then 3SUM is solvable in truly subquadratic time. We first prove by a direct reduction that 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. After this, we use combine our ideas with a recent result by Patrascu [38] to show a tight relationship between the two problems.

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 is 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 [20] 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 [4]. 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 $> 3m$ elements is $O(n/m)$ in expectation.

After the hashing we process all elements that get mapped to large buckets (size $> 3m$) as follows. Suppose that $a \in A$ is an element mapped to a large bucket. Then go through all elements b of B and check whether $a + b \in C$. Process elements of B and C mapped to large buckets similarly. 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 $i \in [n/m]$, pick an arbitrary ordering of the elements of A_i and then denote by $A_i[k]$ the kth element in A_i. Similarly, denote by $B_i[k]$ and $C_i[k]$ the kth elements of B_i and C_i.

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 $s \in [m]$ and $t \in [m]$ 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 $c \le m$ 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 $d = C_{k_{ijf}}[p]$, for $f \in \{0, 1\}$, $k, \ell, p \in \{1, \ldots, m\}$, and $i, j \in \{1, \ldots, n/m\}$. Then consider instance $c = fm + p$. The triangle $x_i, y_j, z_{k\ell}$ has weight $A_i[k] + B_j[\ell] + C_{k_{ijf}}[p] = a + b + d$.

Each graph has $O(n/m + m^2)$ 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,

[4]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.

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. □

Recently, Patrascu [38] gave a reduction from 3SUM to a problem called Convolution-3SUM problem. Combining his result with our ideas, we show that if one can find a 0–weight triangle in truly subcubic time, then 3SUM can be solved in truly subquadratic time, thus tightening the relationship between the two problems.

Convolution-3SUM problem is the following problem: given arrays $A[0\ldots N]$, $B[0\ldots N]$ and $C[0\ldots N]$, find some i,j with $A[i]+B[j]=C[i+j]$.

Theorem 4.2 (Patrascu [38]) *If Convolution-3SUM can be solved in $O(N^2/f(N))$ time for some function f, then 3SUM can be solved in $O(n^2/f^{1/3}(n))$ time.*

Here we give details on how to reduce Convolution-3SUM to 0–weight triangle.

Theorem 4.3 *If a 0-edge-weight triangle in a graph on c nodes can be found in $T(c)=O(c^3/f(c))$ time for some function f, then Convolution-3SUM on N length arrays can be solved in $O(N^2/f(\sqrt{N}))$ time time, and 3SUM on n integers can be solved in $O(n^2/f^{1/3}(\sqrt{n}))$ time.*

Proof: Consider an instance A,B,C of Convolution-3SUM of length N. We want to find i,j such that $A[i]+B[j]=C[i+j]$. For $i\in[\sqrt{N}]$, let A^i be the $\sqrt{N}$ length array with $A^i[j]=A[(i-1)\sqrt{N}+j]$, that is we split A up into consecutive chunks of length $\sqrt{N}$. We split up B similarly into B^i for $i\in[\sqrt{N}]$. For every $i\in[\sqrt{N}]$ we create a complete tripartite graph G_i on partitions L^i, R^i, S^i with $\sqrt{N}$ nodes each. Let $L^i[t]$, $R^i[t]$ and $S^i[t]$ denote the t-th node in partition L^i, R^i and S^i respectively. We add weights to the edges as follows:

- $w(L^i[s], R^i[t]) = A^s[t]$
- $w(R^i[t], S^i[q]) = B^i[q-t]$
- $w(L^i[s], S^i[q]) = -C[(s+i-2)\sqrt{N}+q]$

Now, if we find a 0-edge-weight triangle $L^i[s], R^i[t], S^i[q]$, then

$$0 = A^s[t] + B^i[q-t] - C[(s+i-2)\sqrt{N}+q],$$

$$A[(s-1)\sqrt{N}+t] + B[(i-1)\sqrt{N}+q-t] = C[(s+i-2)\sqrt{N}+q],$$

and if $m_1=(s-1)\sqrt{N}+t$, $m_2=(i-1)\sqrt{N}+q-t$, then

$$m_1+m_2=(s+i-2)\sqrt{N}+q,$$

and we have found $A[m_1]+B[m_2]=C[m_1+m_2]$.

If on the other hand there is some $m_1, m_2, m_3=m_1+m_2$ such that $A[m_1]+B[m_2]=C[m_1+m_2]$, then consider G_i for which $m_2=(i-1)\sqrt{N}+\bar{m}_2$ for $\bar{m}_2\in\{0,\ldots,\sqrt{N}-1\}$. Let $m_1=(s-1)\sqrt{N}+t$ for $0\le t<\sqrt{N}$. Consider the edge $(R^i[t], S^i[\bar{m}_2+t])$. It has weight $B^i[\bar{m}_2]=B[m_2]$. Edge $(L^i[s], R^i[t])$ has weight $A[m_1]$ and edge $(L^i[s], S^i[\bar{m}_2+t])$ has weight $-C[w]$ for $w=(s+i-2)\sqrt{N}+\bar{m}_2+t=m_1+m_2$. Hence there is a triangle with weight $A[m_1]+B[m_2]-C[m_1+m_2]=0$.

If a 0-edge-weight triangle in a graph on c nodes can be found in $T(c)=O(c^3/f(c^2))$ time, then Convolution-3SUM on N length arrays can be solved in

$$\sqrt{N}T(\sqrt{N}) = \sqrt{N}\cdot N^{3/2}/f(N) = N^2/f(N)$$

time, and hence 3SUM can be solved in $O(n^2/f^{1/3}(n))$ time. □

Corollary 4.4 *If* 0*-edge-weight triangle on* n *node graphs is in* $O(n^{3-\varepsilon})$ *time, then* 3*SUM on* N *integers is in* $O(N^{2-\varepsilon/6})$ *time.*

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. [32, 37]). 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 [29, 18]. We show that a 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 4.5 *There is a* $\delta \in (0,1)$ *and some* $k \geq 3$ *such that* F*-*WEIGHT k*-*CLIQUE *is in* $O(\text{poly}(b) \cdot n^{\delta k})$ *time over a field* F *of* $2^{\Theta(b)}$ *elements.*

Theorem 4.6 *Hypothesis 4.5 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 4.6. 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 [49], 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 $(a_1, \ldots, a_n) \in F^n$.

$$P(a_1, \ldots, a_n) = 0 \text{ (over } K\text{) if and only if for all } i = 1, \ldots, m,\ p_i(a_1, \ldots, a_n) = 0 \text{ (over } F\text{)}.$$

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 $O(\text{poly}(b) n^{\delta k})$ algorithm for F-WEIGHT k-CLIQUE translates to an $O(\text{poly}(m,n)|F|^{\delta n})$ 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 $|K| \leq |F|^m \text{poly}(n)$, so the hypothesis entails that this clique problem is in $O(\text{poly}(m,n)|F|^{\delta n})$ time.

5 Open Problems

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

- Is there a $f(k) \cdot n^{k(1/2-\varepsilon)}\text{poly}(n)$ 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 [28]. We are also grateful to Noga Alon for some helpful discussions.

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] F. A. Behrend. On the sets of integers which contain no three in arithmetic progression. In *Proc. Nat. Acad. Sci.* 23:331–332, 1946.

[10] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Counting Paths and Packings in Halves. In *Proc. ESA*, 578–586, 2009.

[11] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Evaluation of permanents in rings and semirings. *Inf. Process. Lett.*, 110(20):867–870, 2010.

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

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

[14] 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.

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

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

[17] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms (3rd ed.). 2009. MIT Press. ISBN 0-262-03384-4.

[18] 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.

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

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

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

[22] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

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

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

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

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

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

[28] 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.

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

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

[31] I. Koutis and R. Williams. Limits and Applications of Group Algebras for Parameterized Problems. In *Proc. ICALP*, volume 5555, pages 653–664, 2009.

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

[33] 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.

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

[35] L. Moser. On non-averaging sets of integers. *Canadian Journal of Mathematics* 5:245–252, 1953.

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

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

[38] M. Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proc. STOC*, pages 603–610, 2010.

[39] F. Romani. Shortest-path problem is not harder than matrix multiplication. *Inf. Proc. Lett.*, 11:134–136, 1980.

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

[41] R. Salem and D. C. Spencer. On sets of integers which contain no three terms in arithmetical progression. *Proc. Nat. Acad. Sci.* 28:561–563, 1942.

[42] 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.

[43] 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.

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

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

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

[47] 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.

[48] V. Vassilevska Williams and R. Williams. Subcubic Equivalences Between Path, Matrix, and Triangle Problems. In *Proc. FOCS*, pages 645–654 , 2010.

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

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