The k-Clique hypothesis is dead, for good

Virginia Vassilevska Williams
MIT

Or Zamir
Tel Aviv University

Abstract

We present an algorithm that solves the k-Clique problem for any large enough k and any n-vertex graph in $n^{\left(\frac{\omega}{3}-c\right)k}$ time, where $\omega \geq 2$ is the matrix multiplication exponent and $c > 0$ is a positive constant *independent* of both ω and k. This refutes the long-standing k-Clique Hypothesis, regardless of the value of the matrix multiplication exponent, even if $\omega = 2$. In fact, our algorithm solves the more general problem of counting weighted k-cliques.

1 Introduction

Finding cliques in graphs is a classic problem in computer science with an abundance of applications. The problem is NP-hard when the desired clique size k is part of the input. When k is a fixed small constant, there is a well-known way to beat the brute-force k-tuple enumeration algorithm. More than 40 years ago, Nešetřil and Poljak [NP85] showed how to reduce k-Clique in n node graphs to triangle (i.e. 3-clique) detection in $O(n^{k/3})$ node graphs, when k is divisible by 3. This leads to an $n^{\omega k/3+o(1)}$-time algorithm for k divisible by three [NP85], and only adds at most 2/3 to the exponent[1] for other k. Here $\omega \leq 9/4$ denotes the exponent of square matrix multiplication (see e.g. [Ope26b, AvdBZ26, Shp26, Str69, CW90, ADV$^+$25]).

The exponent for k-clique detection has remained unchallenged until today, except for small improvements using rectangular matrix multiplication for the cases when k is not divisible by 3 (e.g. [EG04]). This motivated the somewhat dubious, yet perfectly consistent with the state of affairs, k-Clique Hypothesis [Woe04, ABV18]:

The *k-Clique Hypothesis*: in the word-RAM model with $O(\log n)$ bit words, for k divisible by 3, k-clique detection in n node graphs requires $n^{\omega k/3-o(1)}$ time.

Stronger versions [DVV21] even conjectured the optimality of the rectangular matrix multiplication solution for values of k not divisible by three.

Fine-grained complexity then developed many reductions from k-Clique to other algorithmic problems (e.g. [ABV18, Cha16, BW17, JS17, ABBK17, NNT26, LVW18, DVV21, DV22, Cha10, FKRS25, FKRS26, GK23]), showing that in order to make further progress on these problems, one must first refute the k-Clique Hypothesis.

The k-Clique Hypothesis is usually used for problems whose best running times rely on matrix multiplication (MM) in some way. To show that MM is likely necessary for the best algorithm for a problem π, one reduces to π, either MM itself, or another problem that is believed to require MM. Such problems include Boolean Matrix Multiplication (BMM, whose best algorithms are the same as for MM), triangle detection (which is subcubically equivalent to BMM [VW10, VW18]) or more

[1] If $k \equiv t \pmod 3$, we can reduce the problem to n^t instances of $(k-t)$-Clique, and the overall exponent for k-Clique becomes $\omega(k-t)/3 + t = \omega k/3 + (3-\omega)t/3 \leq \omega k/3 + t/3 \leq \omega k/3 + 2/3$.

generally k-Clique either for some other fixed k (such as $k = 4$) or for any constant $k \geq 3$, relying on the k-Clique Hypothesis. The latter route is particularly useful when the reduction has some polynomial overhead which the high conjectured running time of k-Clique lets you overcome.

For instance, in Abboud, Backurs and Vassilevska W.'s reduction from k-Clique to Context Free Grammar (CFG) Recognition [ABV18], an n-node graph is encoded as a string of length $O(n^{k/3+1})$ for a single fixed grammar, a factor of n more than the $n^{k/3}$ that a tight reduction would produce. An $O(n^{\omega-\varepsilon})$-time CFG recognizer would then solve k-Clique in $O(n^{(\omega-\varepsilon)(k/3+1)}) = O(n^{\omega k/3-\varepsilon k/3+\omega-\varepsilon})$ time. This is faster than $n^{\omega k/3}$ only when $k > 3(\omega-\varepsilon)/\varepsilon$ and, under the k-Clique Hypothesis that makes an assertion for all k, Valiant's $O(n^{\omega})$-time parser is optimal. If one were trying to get the reduction to work from small k, such as from triangle, it would not work because of the extra $+1$ overhead in the reduction exponent. There are many such examples, where a reduction from a fixed size clique such as triangle (or from BMM) would not quite work due to the overhead, so the k-Clique Hypothesis became useful.

In 2004, Woeginger [Woe04] challenged algorithms researchers to beat the Nešetřil-Poljak running time for some fixed k. Over two decades passed, the running time remained untouched, the hypothesis gained more traction and more fine-grained problems were found to be k-Clique-hard.

In this paper we strongly refute this hypothesis, and we bring forth hope that the k-Clique-hard problems can also have faster algorithms. We prove:

Theorem 1.1. *There exist constants K_0 and $\delta > 0$, independent of the matrix multiplication exponent $\omega \geq 2$, such that for every $k \geq K_0$, k-Clique can be solved in $O(n^{(\omega-\delta)k/3})$ time. In fact, the algorithm can return the number of k-cliques in the given graph.*

We did not try to optimize K_0 or δ, as we eventually discuss in Section 5.

Recent breakthroughs have refuted several central conjectures in fine-grained complexity, including the APSP and 3SUM hypotheses [AW26], the Exponential Time Hypothesis [Ope26a], and the Hyperclique Hypothesis [Fis26, ALP+26, CIMY26]. The k-Clique Hypothesis presents an additional difficulty: unlike these conjectures, its predicted running time depends on ω, whose exact value is unknown. Thus, even an improvement over the best known k-Clique algorithm could cease to contradict the hypothesis if a subsequent improvement in matrix multiplication lowered the known upper bound on ω. Our result avoids this moving target: the exponent saving is relative to the true value of ω, and therefore remains a refutation of the hypothesis regardless of any future progress on matrix multiplication, including in the case that it would be proven that $\omega = 2$.

In order to obtain improved k-Clique running times, we solve a potentially harder *weighted k-clique sum* problem that generalizes counting the number of k-cliques: Every edge e of the host graph G has a weight $w(e)$, the weight of a k-clique in G is the product of its edge weights, and we want to compute the sum of all weights of k-cliques in G. The Nešetřil-Poljak style reduction works just as well for weighted k-clique sum, and the problem in n node graphs can be reduced to weighted ℓ-clique sum in $O(n^{k/\ell})$ node graphs (when k is divisible by ℓ).[2] Thus, to get a faster weighted k-clique sum algorithm for all large enough k, it suffices to get a faster weighted ℓ-clique sum algorithm for some constant ℓ. We do this for $\ell = 9$.

In fact, we give an improved algorithm that computes a weighted sum over the copies of any given 9-node pattern H in an n-node graph G (see Section 3 for a definition for this more general

[2]The edges inside each set of k/ℓ vertices contribute a weight per node of the reduced graph; since that graph is without loss of generality ℓ-partite, this weight can be multiplied into the node's edges toward one fixed other part, so edge weights suffice. Vertex weights only become necessary for the general patterns H of Theorem 1.2.

problem).[3]

Theorem 1.2. *There is an absolute constant $\varepsilon > 0$ such that, over any field of characteristic zero, for any n-node host graph G and every pattern graph H on nine vertices the weighted sum over the copies of H in G can be computed using $n^{3\omega-\varepsilon+o(1)}$ arithmetic operations.*

As k-Clique in unweighted graphs can be converted to the corresponding weighted sum problem with small weights, the arithmetic complexity implies a similarly efficient algorithm in the word-RAM model, and combining the algorithm with the Nešetřil-Poljak reduction, we obtain Theorem 1.1 as a corollary.

Our proof makes central use of an algebraic decomposition lemma that we distilled, with a publicly available AI model's assistance, out of the recent ETH refutation of OpenAI [Ope26a]. We separate the proof of this decomposition (in Section 4) from our algorithmic framework that uses it in a black-box manner (in Section 3). We devote Section 6 to discuss and reflect in detail on our methodology of AI-use.

Consequences for exact exponential-time algorithms. Our result also improves the running time of exact algorithms for several NP-hard optimization problems. Williams [Wil05], and subsequently Koivisto [Koi06], showed that Max-2-CSP on n Boolean variables with polynomially bounded integer scores can be solved in $2^{\omega n/3+o(n)}$ time using fast matrix multiplication. Their framework includes Max-Cut and Max-2-SAT, as well as counting the number of optimal solutions. We improve this bound to

$$2^{(\omega/3-c)n+o(n)}$$

for some absolute constant $c > 0$, independent of ω. Indeed, we partition the variables into nine groups, regard each assignment to a group as a domain element, and encode the constraints by unary and pairwise weights of a weighted K_9 instance. Our algorithm then gives the stated improvement, including when $\omega = 2$. For integer scores, as in [Koi06], we first shift each constraint's scores by a constant so that all scores are nonnegative (which changes every assignment's total score by the same known amount), and then use powers of a formal variable to record the objective value, so that the resulting sum is a generating polynomial whose coefficients count the assignments attaining each score. When the number of constraints and their score magnitudes are polynomially bounded in n, this polynomial has polynomial degree, and the polynomial arithmetic contributes only a polynomial overhead. Thus, the improvement applies to exact optimization and counting optimal solutions for such Max-2-CSP instances, including unrestricted dense constraint graphs.

The improvement also extends to related counting problems. For every sufficiently large fixed k, it gives the same exponent saving for counting induced copies of any k-vertex graph, rather than only cliques [CDN25]. For a labeled pattern, each pairwise weight enforces adjacency or nonadjacency as prescribed by the pattern, and also excludes equal host vertices. Grouping the pattern vertices into nine blocks then gives a weighted K_9 instance, to which our algorithm applies. Moreover, Scott and Sorkin [SS09] extended the matrix-multiplication approach to polynomial constraint satisfaction problems, including computing Ising partition functions, counting cuts of every size, and solving graph bisection. Our algorithm similarly improves their general matrix-multiplication bounds to $2^{(\omega/3-c)n+o(n)}$ arithmetic operations. Under their polynomial boundedness assumptions on the polynomial representations and coefficient sizes, the same bound holds for the running time up to a $2^{o(n)}$ factor.

[3]The full proof in Section 3 computes the more general weighted homomorphism sum defined in (1); for simplicity, here we discuss a sum over copies of H. The sum can be restricted to copies by using the complete nine-vertex pattern and multiplying each pairwise weight by $\mathbf{1}[a_i \neq a_j]$, taking the original weight to be one on pairs absent from H.

2 Overview and organization

We give a brief overview of the algorithm, starting with the simpler case where $\omega > 2$. Our algorithm improves the standard $n^{3\omega}$ running time for computing the weighted sum for any *pattern* graph on nine vertices. Rather than designing the algorithm specifically for K_9, we prove the improvement simultaneously for all nine-vertex pattern graphs, by induction on the number of edges. For sufficiently sparse patterns, the weighted sum is easy to compute. For denser patterns, we will reduce the problem to computing the weighted sums for patterns with fewer edges, together with one additional term that can be computed faster using square matrix multiplication as a black-box.

The standard $n^{3\omega}$ algorithm à la [NP85] partitions the nine vertices of the pattern graph H into three sets of three. We denote these triples by V_1, V_2, V_3. Then to compute the weighted sum for H in G, we create three $n^3 \times n^3$ matrices M_{12}, M_{23}, M_{31} encoding the weights of the triples inside each V_i and where M_{ij} encodes the weights of the edges between every two triples in $V_i \times V_j$, and we finish by computing the trace of the matrix product $M_{12}M_{23}M_{31}$.

The original Nešetřil-Poljak style reduction leaves a lot of freedom how we pack the edge weights into the matrices M_{ij}. While it is clear that edges between two different parts V_i and V_j of H must be represented in M_{ij}, there is freedom how the edge weights of a triple inside a particular V_i are distributed between the two matrices $M_{i(i+1)}$ and $M_{i(i-1)}$ that see V_i.

This freedom is what our algorithm exploits. Suppose that H contains two edges xz and yz that share the vertex z; we call such a 2-path a *wedge*. We choose the partition so that x, z, y form one of the three triples, say the one corresponding to V_1, and we split the wedge between the two matrices that see V_1: the weight of xz is placed in M_{12} and the weight of yz in M_{31}. We call these the first and the second matrix, respectively.

In the trace of $M_{12}M_{23}M_{31}$, the assignment to the triple of V_1 is both the row index of M_{12} and the column index of M_{31}. The trace forces the two to coincide. We make this constraint explicit for x and y. Let s, u, v, t be vertices of the host graph G: s and u are the vertices to which the first matrix maps x and y, respectively, and v and t are the vertices to which the second matrix maps x and y, respectively. The middle vertex z of the wedge is not split: both matrices see the same assignment to z, which, with a slight abuse of notation, we also denote by z. The standard algorithm requires the two matrices to map x to the same vertex ($s = v$) and y to the same vertex ($u = t$). We express this by inserting, between M_{31} and M_{12}, an identity matrix I whose rows are indexed by (v, t) and whose columns are indexed by (s, u); this leaves the trace unchanged.

Instead, consider replacing this identity matrix with another permutation matrix, which we call the *swap operator* P, that crosses these constraints: it imposes $s = t$ and $u = v$, so that the vertex used for x by the first matrix is the one used for y by the second, and vice versa. Since the edge xz is in the first matrix and the edge yz is in the second, replacing I by P transforms their contribution from $w_{xz}(s, z)w_{yz}(t, z)$ into

$$w_{xz}(s, z)w_{yz}(s, z).$$

Thus the two edge weights can be replaced by a single weight on the edge xz. All other weights still depend on at most two variables, so the resulting sum is a weighted sum for a pattern with at least one fewer edge. We refer to this operation as *folding* the wedge. Figure 1 illustrates the swap and the fold on a small example.

Of course, replacing I by P changes the answer. However, noting that $I = P + (I - P)$ allows us to compute the original sum by adding the contribution of the swap and the contribution of $I - P$. The first can be computed recursively on a sparser pattern. For the second, we can take advantage of the fact that $I - P$ has rank only $n(n-1)/2$, rather than n^2, which translates to lowering one of

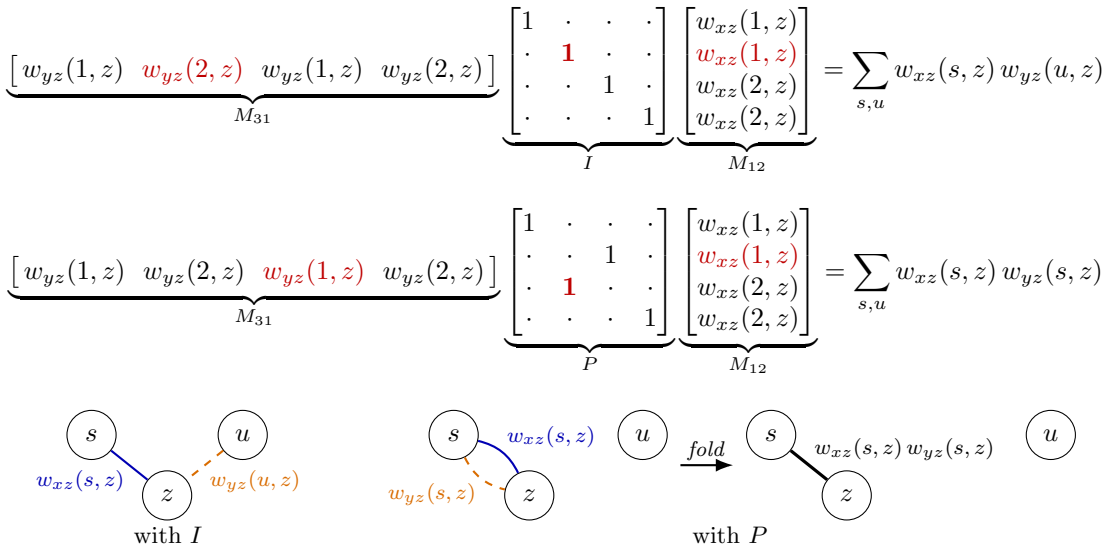


Figure 1: The swap and the fold on a small example with host vertices $\{1, 2\}$. We fix the assignments to V_2 and V_3 and the value of z, and keep only the factors coming from the two wedge edges. Since yz is stored in M_{31}, the relevant row of M_{31} is indexed by the pair (v, t) to which M_{31} maps (x, y), and its entry is $w_{yz}(t, z)$; since xz is stored in M_{12}, the relevant column of M_{12} is indexed by the pair (s, u) to which M_{12} maps (x, y), and its entry is $w_{xz}(s, z)$. Both are ordered $(1, 1), (1, 2), (2, 1), (2, 2)$, and dots are zeros. The identity I pairs each (s, u) with $(v, t) = (s, u)$, so the two edges depend on different host vertices and form the wedge s–z–u (bottom left; xz solid, yz dashed). The swap P pairs (s, u) with $(v, t) = (u, s)$, so both edges depend on (s, z) and fold into a single edge (bottom right). The highlighted entries follow the term $(s, u) = (1, 2)$, which contributes $w_{xz}(1, z)\, w_{yz}(2, z)$ under I and $w_{xz}(1, z)\, w_{yz}(1, z)$ under P.

the three dimensions in the matrix multiplication instance we need to solve. Unfortunately, this is still $\Omega(n^2)$ and does not give a polynomial improvement in the running time.

We generalize the above attempt with a stronger version of this decomposition. We show that, for every constant $\varepsilon > 0$, there exists a constant $\delta > 0$ such that

$$I = R + \sum_{j=1}^{M} c_j S_j, \qquad M \le n^{\varepsilon}, \qquad \operatorname{rank}(R) \le n^{2-\delta},$$

where the matrices S_j are *partial swaps*: a generalized version of the above swap that still allows folding an edge (and thus reducing the number of edges in the pattern) in a similar manner.

Let us see why this decomposition is sufficient when $\omega > 2$. The contribution of R can be computed by replacing one of the three matrix dimensions, originally n^3, by $n \cdot \operatorname{rank}(R) \le n^{3-\delta}$. By partitioning the other two dimensions into blocks and using square matrix multiplication, the resulting contraction can be evaluated in

$$n^{3\omega-(\omega-2)\delta+o(1)}$$

time. When $\omega > 2$, this gives a polynomial improvement over $n^{3\omega}$. We can also afford to explicitly construct the original matrices and apply the low-rank decomposition, since their total size is only $n^6 < n^{3\omega}$.

The remaining contribution consists of at most n^{ε} recursive calls to patterns with fewer edges. Suppose inductively that we have already obtained a polynomial saving n^{α} for all these patterns.

We can choose $\varepsilon < \alpha$, so that even after making all the recursive calls we retain a positive saving. The algebraic decomposition guarantees that this choice still gives some $\delta > 0$, however small ε is. Since the number of edges in a nine-vertex pattern graph is at most 36, the induction has only constantly many steps, and we obtain a polynomial improvement for every pattern. Thus, as long as $\omega > 2$, the decomposition essentially gives the entire algorithm.

We state the decomposition formally in Section 3.2, and give a short self-contained proof in Section 4. The algebraic construction itself is essentially a binary specialization of a matrix decomposition from the recent OpenAI result on the Exponential Time Hypothesis [Ope26a], based on polynomial interpolation techniques previously used in matrix rigidity [AW17]. We isolate the particular properties needed for our algorithm and prove them directly, using little more than repeated applications of $I = P + (I - P)$, elementary polynomial interpolation, and a concentration bound for the binomial distribution. We essentially attribute this decomposition to ideas appearing in the OpenAI manuscript [Ope26a], and thus our main algorithmic contribution is in using this decomposition to cheaply reduce the weighted sum for a pattern to weighted sums for sparser patterns; this is our clique counting algorithm of Section 3.

There is one additional complication: we want an improvement that holds independently of the value of ω, even if $\omega = 2$. The "fold one wedge at a time to remove an edge" argument sketched above does not achieve this, since reducing only one of the three matrix dimensions gives a saving proportional to $\omega - 2$. Moreover, the explicit construction of the original matrices already costs n^6. To overcome both problems, we apply the decomposition simultaneously to two vertex-disjoint wedges, placed in two different sets of three vertices. The contribution in which both operators are replaced by their low-rank parts now has matrix dimensions $(n^{3-\delta}, n^{3-\delta}, n^3)$, and can be evaluated in $n^{3\omega-(\omega-1)\delta+o(1)}$ time, giving a saving even when $\omega = 2$. Crucially, the low-rank factors also have sparse, explicitly enumerable basis vectors: this lets us construct the projected matrices directly, using smaller matrix products and a fast basis transformation, within the same time bound and without first forming the original $n^3 \times n^3$ matrices. The other terms are reduced to sparser patterns by applying partial swaps, as before. If the pattern does not contain two vertex-disjoint wedges, we show that its weighted sum can already be computed in $O(n^5)$ time. We prove this stronger two-wedge reduction in Section 3.3, and complete the induction in Section 3.4.

3 Improved algorithm for counting weighted cliques

Let H be a *pattern* graph on a fixed set of nine labeled vertices. For each vertex i we are given a vertex-weight function $u_i : [n] \to \mathbb{R}$, and for each edge of the pattern graph $ij \in E(H)$ we are given an edge-weight function $w_{ij} : [n]^2 \to \mathbb{R}$. We wish to compute

$$Z_H = \sum_{a_1,\ldots,a_9 \in [n]} \prod_{i=1}^{9} u_i(a_i) \prod_{ij \in E(H)} w_{ij}(a_i, a_j). \tag{1}$$

All weights are arbitrary, and missing edges are interpreted as having weight one. We refer to Z_H as the *weighted sum* for the pattern H. Throughout, we present the algorithm for real-valued weights, but the proof works, with slight modifications, over any field of characteristic zero: the only constants it introduces are rational numbers (see Appendix A).

The above is a standard and natural generalization of the k-clique counting (and hence also detection) problem. If we set $H = K_9$, the adjacency matrix of a loopless graph as all the edge-weight functions w_{ij}, and the constant one as all the vertex-weight function, then $Z_H/9!$ is the number of nine-cliques. The generalization to all nine-vertex patterns H rather than only the clique K_9

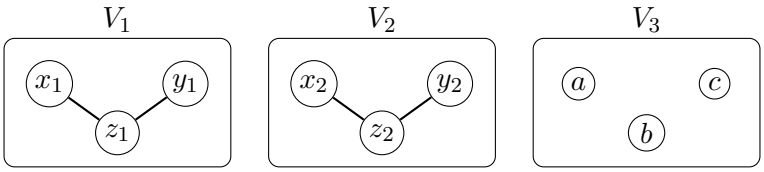


Figure 2: The standard reduction partitions the nine pattern vertices into three sets of three. When the pattern contains two vertex-disjoint wedges, we place one wedge in V_1 and one in V_2. The third set V_3 is arbitrary.

will be useful algorithmically, because we will iteratively construct improved algorithms for each of these patterns by making recursive calls to the same problem on sparser patterns.

The standard upper bound for detecting or counting 9-cliques works as-is also for computing (1): We partition the nine vertices of H into three sets V_1, V_2, V_3 of size three. Let $A, B, C \in [n]^3$ denote assignments to these sets. Every edge has both endpoints in at most two of the three sets, and its weight can therefore be assigned to one of three matrices $M_{12}(A, B)$, $M_{23}(B, C)$, $M_{31}(C, A)$. Unary weights are assigned to any one of the two matrices containing their corresponding vertex, with each weight appearing exactly once. This gives

$$Z_H = \sum_{A,B,C\in[n]^3} M_{12}(A, B)M_{23}(B, C)M_{31}(C, A) = \operatorname{tr}(M_{12}M_{23}M_{31}). \tag{2}$$

The matrices have order n^3, so using fast matrix multiplication to compute the right-hand trace gives the $n^{3\omega}$ upper bound. Note that the explicit construction of the matrices costs $O(n^6)$ time, which is not larger than the first term as $\omega \geq 2$.

3.1 Wedges and local swaps

Our algorithmic framework is as follows: we will prove, inductively, that the above baseline of $n^{3\omega}$ can be polynomially improved for every pattern graph H; starting from the sparsest patterns as our base case and reducing solving the problem for a pattern H to solving it on patterns H' with fewer edges as the inductive step.

A *wedge* is a triple of distinct vertices x, z, y for which $xz, yz \in E(H)$. We do not require xy to be absent (nor to exist). Two wedges are disjoint if they have no common vertex.

Our base case, that is, the set of patterns we can immediately give an improved algorithm for, is the family of all patterns H that do not contain two vertex disjoint wedges.

Lemma 3.1. *If H contains no two vertex-disjoint wedges, its weighted sum can be computed in $O(n^5) \leq O(n^{3\omega-1})$ time.*

Proof. If H contains no wedge, each connected component is an edge or an isolated vertex, so the sum factors into one- and two-variable sums. Otherwise fix any wedge x, z, y and enumerate its n^3 assignments. The subgraph of H induced by the remaining six vertices cannot contain another wedge, and is therefore a matching together with isolated vertices. After fixing the assignments to x, z, y, edges with exactly one endpoint among these vertices become unary weights on the remaining vertices, while their vertex weights and the edges entirely within $\{x, z, y\}$ contribute a scalar factor. The remaining sum can be evaluated in $O(n^2)$ operations for each fixed assignment, giving the stated bound. □

We next describe a local operation on one wedge, which we call a *swap*, and is meant to reduce the number of edges in the pattern by at least one. Place the wedge vertices x, z, y in the same three-vertex set among A, B, C in the reduction above. This set appears in two of the three matrices. In the first occurrence, use values (s, u) to denote the assignment for the pair (x, y); in the second, use values (v, t) to denote it. We can, without changing anything, place the identity matrix between these two matrices, as multiplying with it does not change the result. We can then interpret this identity matrix as if it "forces" an equality between the assignments (s, u) and (v, t) to (x, y) in both matrices. Indexing this matrix's rows and columns by these assignments, we write it as

$$I_{(s,u),(v,t)} = \mathbf{1}[s = v]\mathbf{1}[u = t]. \tag{3}$$

I is simply the identity matrix of order n^2. That is, I requires the first and the second matrix to agree on the vertex assigned to x ($s = v$) and on the vertex assigned to y ($u = t$), while the swap operator P defined in (4) below matches them crosswise ($s = t$ and $u = v$). Note that we are not planning to touch the third vertex (corresponding to z) so we do not bother writing it down, but every time we write a matrix of dimension n^2 we mean that the third variable z maintains equality (equivalently, the matrix should be tensorized with I_n).

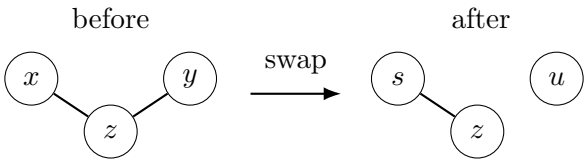
Consider what happens if we replace I with the following matrix P, which we call the *swap operator*,

$$P_{(s,u),(v,t)} = \mathbf{1}[s = t]\mathbf{1}[u = v]. \tag{4}$$

This exchanges the two values across the occurrences of the pair. It is a permutation matrix, and unlike I, it *does* change the computed value.

Before we discuss the relation between the new quantity we compute after this replacement and what we actually want to compute, we discuss the implications of applying this swap operator on the pattern graph H.

Intuitively, this swap *folds* a wedge by swapping the two pivot values x, y across their occurrences in adjacent matrices. The two edge-weights $w_{xz}(s, z)$ and $w_{yz}(t, z)$ become a single edge-weight on (s, z) after we impose that $s = t$.



To see it, place the edge xz in the first matrix and yz in the second. Their contribution is $w_{xz}(s, z)w_{yz}(t, z)$. Under (3) these are the original two edge weights. Under (4), the constraint $s = t$ turns their product into

$$w_{xz}(s, z)w_{yz}(s, z), \tag{5}$$

which can now be written as *one* edge weight on the pair (s, z). That is, we no longer need edge-weights on the second edge. Note that the other new variable $u = v$ remains a separate vertex. All remaining original edge weights still depend on at most two variables. Thus the exchange produces a weighted sum in which the two edges of the wedge have become one. In particular, we reduced the number of edges in the pattern by one. We will refer to this operation as *folding* the wedge.

This observation already gives us a promising approach to attempt: If we write the identity matrix as $I = P + (I - P)$, we can reduce solving our original problem to computing two sums (and then adding them): the first sum, uses exactly our swap operator P instead of I, and in turn

corresponds to solving the problem on a sparser pattern - which we assume inductively already has a faster algorithm. The other sum, uses the matrix $(I - P)$ rather than I which also gives a slight improvement: for the n out of the n^2 index pairs in which $x = y$, the swap operator does nothing; in particular, $I - P$ vanishes on these n index pairs, and on each of the remaining $n(n-1)/2$ pairs $\{(s,u),(u,s)\}$ it acts as the rank-one matrix $\left(\begin{smallmatrix} 1 & -1 \\ -1 & 1 \end{smallmatrix}\right)$, so its rank is $n(n-1)/2 < n^2$. Replacing I with a low-rank matrix is also very useful algorithmically when computing the final matrix multiplications: it reduces one of the three dimensions in the matrix multiplication from n^3 to $n \cdot \text{rank}$. Unfortunately, the rank of $(I - P)$ is still $\Omega(n^2)$, so this replacement doesn't result in any asymptotic saving.

On the other hand, this already motivates our entire algorithm: Instead of decomposing I to merely one swap P and the remainder $(I - P)$, we would decompose it to a combination of a larger number of swaps such that the overall remainder would be of sufficiently low rank. Intuitively, if we know we can get a polynomial saving of n^α on each of the sparser patterns, we can allow ourselves to solve reduced problems on polynomially many swap operators n^ε as long as $\varepsilon < \alpha$. We will show that a decomposition with any polynomial number n^ε of swap* operators always exists that reduces the rank of the overall remainder multiplicatively by some *polynomial* n^δ. The little asterisk is that the decomposition would not be to actual swap operators as we defined them above, but to a slightly more general version of them we will call partial swap, and is going to function in exactly the same manner.

Why do we ask for two disjoint wedges rather than one? It is worth explaining why we would simultaneously apply "two iterations" of our reduction to two disjoint wedges, rather than just doing so one by one. Suppose first that we apply it to a single wedge, placed in the first of the three blocks in (2). In the low-rank contribution, the first block has at most $n^{3-\delta+o(1)}$ possible assignments, while the other two blocks still have n^3 assignments each. Thus, the resulting triangle contraction has dimensions

$$(n^{3-\delta}, n^3, n^3).$$

In particular, one of the input matrices is already of size n^3-by-n^3 which has n^6 entries. While this is polynomially smaller than $n^{3\omega}$ when $\omega > 2$, it prevents any savings when $\omega = 2$.

Applying the decomposition to two disjoint wedges avoids this problem. In the contribution where both operators are replaced by their low-rank parts, two of the three blocks have at most $n^{3-\delta+o(1)}$ assignments. Hence, we will be able to avoid paying n^6 in the low-rank case, which provides an improvement even when $\omega = 2$.

3.2 Partial swaps and a decomposition using them

We first define the slightly more general swap operators that we will use. For simplicity, assume that $n = 2^\ell$, so we can view every element of $[n]$ as a binary string of length ℓ. This assumption is without loss of generality, since we can pad n to the next power of two.

A *partial swap* is defined by choosing a set $J \subseteq [\ell]$ of bit positions and fixing the values of the four bits s_i, u_i, v_i, t_i for every $i \in J$. On all other positions, we impose the same constraints as in the full swap (4), namely $s_i = t_i$ and $u_i = v_i$. Formally, for fixed values $\rho_i \in \{0,1\}^4$ for each $i \in J$, the corresponding matrix is

$$S_{(s,u),(v,t)} = \prod_{i \notin J} \mathbf{1}[s_i = t_i]\mathbf{1}[u_i = v_i] \prod_{i \in J} \mathbf{1}[(s_i, u_i, v_i, t_i) = \rho_i].$$

In particular, for $J = \emptyset$ no bits are fixed and the constraints $s = t$ and $u = v$ hold in full, so S is the original swap P of (4) rather than the identity.

As in the case of the full swap, we assign the two edges xz and yz to the two different matrices containing the wedge, and keep this assignment fixed throughout the decomposition. We next show that partial swaps retain the edge-folding property of the original swap.

Lemma 3.2. *Replacing the identity on the pivot pair of a wedge by any partial swap, with the edge weights assigned as above, gives a weighted sum for a pattern with at least one fewer edge. The new weight functions can be constructed in $O(n^2)$ time.*

Moreover, applying partial swaps simultaneously to two vertex-disjoint wedges reduces the number of edges by at least two.

Proof. Let J be the set of fixed bit positions of the partial swap. Outside J, the constraints are $s_i = t_i$ and $u_i = v_i$, while all four bits are fixed on J. Thus, there are two free strings $p, q \in \{0,1\}^{\ell-|J|}$, namely $p = (s_i)_{i\notin J} = (t_i)_{i\notin J}$ and $q = (u_i)_{i\notin J} = (v_i)_{i\notin J}$, such that both s, t are determined by p, and both u, v are determined by q: outside J they coincide with p (respectively q), and inside J their bits are the fixed values ρ_i. We write $s = s(p)$, $t = t(p)$, $u = u(q)$ and $v = v(q)$ accordingly.

In particular, the two edges of the wedge contribute

$$\widetilde{w}(p, z) = w_{xz}(s(p), z)w_{yz}(t(p), z),$$

which is a single edge-weight function. Every other weight still depends on at most two variables, so the result is a weighted sum for a pattern with at least one fewer edge. The free strings range over $\{0,1\}^{\ell-|J|}$, which may have fewer than n elements. We identify this set with a subset of $[n]$ via an arbitrary injection, and set the unary weights of p and q to zero outside its image, so that the additional values of the variables contribute nothing to the sum.

For two vertex-disjoint wedges, we can assign their edge weights and apply the partial swaps independently, reducing the number of edges by at least two. Since there are constantly many unary and pairwise weight functions, the new tables can be constructed in $O(n^2)$ time. □

We now state the algebraic decomposition that we need. It can be viewed as a special case of a construction from the recent OpenAI result on the Exponential Time Hypothesis [Ope26a]. We give a simple self-contained proof in Section 4.

Lemma 3.3 (Rank-or-swap decomposition)**.** *For every constant $\varepsilon > 0$, there exists a constant $\delta > 0$ such that the identity matrix of order n^2 can be written as*

$$I = R + \sum_{j=1}^{M} c_j S_j, \qquad M \le n^{\varepsilon}, \qquad \operatorname{rank}(R) \le n^{2-\delta},$$

where each S_j is a partial swap and $c_j \in \mathbb{R}$.

Moreover, the decomposition is explicit, and R admits an explicit rank-one decomposition

$$R = \sum_{\lambda\in\Lambda} d_\lambda b_\lambda b_\lambda^\top, \qquad |\Lambda| \le n^{2-\delta},$$

where each vector b_λ has at most n nonzero entries, which can be efficiently enumerated. The vectors belong to an explicit basis whose corresponding basis transformation requires only $O(n^2 \log n)$ time.

Together with Lemma 3.2, Lemma 3.3 allows us to replace an identity matrix by either a low-rank matrix or a small number of recursive calls on sparser patterns. It remains to show that the contribution of the low-rank matrix can be evaluated sufficiently fast. For this, we will use not

only its rank bound, but also the additional structure of its rank-one decomposition: it will allow us to construct the required matrix inputs sufficiently fast to not lose the saving even when $\omega = 2$.

It is worth noting that the decomposition above already essentially gives an improved algorithm if $\omega > 2$, folding only a single wedge at a time. Indeed, replacing the identity on one pivot pair by R reduces one of the three matrix dimensions from n^3 to $n^{3-\delta}$, while the other two remain n^3. The resulting matrix multiplication can be performed in

$$n^{3\omega-(\omega-2)\delta+o(1)}$$

time, by partitioning the two larger dimensions into blocks of size $n^{3-\delta}$ and then using square matrix multiplication on each pair of blocks. We can also afford to construct the original matrices and apply the basis transformation in $n^{6+o(1)}$ time, since $\omega > 2$. The remaining terms give at most n^{ε} recursive calls on patterns with fewer edges. Choosing ε sufficiently small at each step of the induction therefore gives an exponent saving whenever $\omega > 2$. The two-wedge reduction is therefore needed to obtain a saving independent of ω, including the case $\omega = 2$. Since the formal analysis is essentially the same, we prove only this stronger version.

3.3 Applying the decomposition to two wedges

We now describe the algorithm for a pattern H containing two vertex-disjoint wedges. Place the wedges in V_1 and V_2, as in Figure 2, and assign the two edges of each wedge to different matrices as before.

Apply Lemma 3.3 to the pivot pair of each wedge, with $\varepsilon > 0$ to be chosen later (which will come from the saving proved by induction to the sparser patterns). For $i = 1, 2$, write

$$I_i = R_i + T_i, \qquad T_i = \sum_{j=1}^{M_i} c_{i,j} S_{i,j},$$

where $M_i \le n^{\varepsilon}$, and R_i has rank at most $n^{2-\delta}$. As before, these operators act only on the two pivot variables, while the third variable in each block is left unchanged.

We could expand the product $I_1 I_2$ into the four terms R_1R_2, R_1T_2, T_1R_2, and T_1T_2. However, the terms containing both a low-rank operator and a partial swap would not directly correspond to weighted sum instances of the form (1). Instead, we use the equivalent identity

$$I_1 I_2 = R_1R_2 + T_1I_2 + I_1T_2 - T_1T_2. \tag{6}$$

Consider first the last three terms. By expanding each T_i into its partial swaps, we obtain at most

$$M_1 + M_2 + M_1M_2 = O(n^{2\varepsilon})$$

weighted sum instances. Indeed, the term T_1I_2 applies a partial swap only to the first wedge, the term I_1T_2 applies one only to the second wedge, and the term T_1T_2 applies partial swaps to both. By Lemma 3.2, every resulting instance has strictly fewer edges than H. The new weight functions can be constructed in $O(n^2)$ time per instance.

Thus, we have reduced the computation of Z_H to $O(n^{2\varepsilon})$ recursive calls on sparser patterns, and one additional term corresponding to R_1R_2. We next show that this remaining term can be evaluated faster than the original matrix multiplication.

Lemma 3.4. *For any value of the matrix multiplication exponent $\omega \ge 2$, the contribution of R_1R_2 in* (6) *can be computed in*

$$n^{3\omega-(\omega-1)\delta+o(1)}$$

time, including the construction of all the required matrices.

Proof. Recall that each low-rank operator admits a decomposition

$$R_i = \sum_{\lambda \in \Lambda} d_\lambda b_\lambda b_\lambda^\top, \qquad |\Lambda| \le n^{2-\delta}.$$

By Lemma 3.3, this decomposition is explicit: the index set Λ, the nonzero entries of every b_λ and the scalars d_λ can be listed in $O(|\Lambda|\, n \log n) \le n^{3-\delta+o(1)}$ time, which is negligible compared to the bounds below, so we never need to compute a rank decomposition. Inserting these decompositions into (2) replaces each of the two pivot pairs by an index $\lambda \in \Lambda$. The third vertex of each wedge remains unchanged. Consequently, the first two blocks now have at most

$$D = n|\Lambda| \le n^{3-\delta}$$

possible assignments each, while the third block still has n^3 assignments. The coefficients d_λ can be absorbed into the entries of the resulting matrices.

If the matrices corresponding to this contraction were already given, we could evaluate it efficiently using square matrix multiplication. Indeed, the three dimensions are now (D, D, n^3). Padding the first two dimensions to $n^{3-\delta}$, partitioning the third dimension into $O(n^\delta)$ blocks of the same size, and using square matrix multiplication on each block gives a running time of

$$n^\delta \cdot n^{\omega(3-\delta)+o(1)} = n^{3\omega-(\omega-1)\delta+o(1)}. \tag{7}$$

In particular, this gives a universal polynomial saving of at least δ in the exponent, which is independent of ω.

However, we must also construct the matrices within this running time. As long as we want to be able to beat n^6, we cannot simply construct the original matrices of order n^3 and apply the low-rank projections afterward, since this would already require n^6 time.

Consider a matrix containing one of the two blocks to which we apply the low-rank decomposition. Denote the assignments to this block by (s, u, z), where (s, u) is the pivot pair, and let $B \in [n]^3$ be the assignment to the other block. For a fixed $\lambda \in \Lambda$, the corresponding projected matrix has entries

$$\widetilde{M}((\lambda, z), B) = \sum_{s,u \in [n]} b_\lambda(s, u) M((s, u, z), B). \tag{8}$$

Here we identify b_λ with a function on $[n]^2$.

By Lemma 3.3, b_λ has at most n nonzero entries. Enumerate these entries as

$$(s_j, u_j), \qquad j = 1, \dots, r_\lambda, \qquad r_\lambda \le n.$$

We can therefore rewrite (8) as

$$\widetilde{M}((\lambda, z), B) = \sum_{j=1}^{r_\lambda} b_\lambda(s_j, u_j) M((s_j, u_j, z), B).$$

Recall that every entry of M is a product of unary and pairwise weight functions. After substituting $s = s_j$ and $u = u_j$, each factor depends either on (z, j), on (j, B), or on (z, B). We may therefore group the factors and write

$$\widetilde{M}((\lambda, z), B) = K_\lambda(z, B) \sum_{j=1}^{r_\lambda} A_\lambda(z, j) B_\lambda(j, B), \tag{9}$$

where the coefficients $b_\lambda(s_j, u_j)$ are included in A_λ.

The sum on the right-hand side is a matrix multiplication between an $n \times r_\lambda$ matrix and an $r_\lambda \times n^3$ matrix. Since $r_\lambda \le n$, we can pad the first two dimensions to n, partition the second matrix into n^2 blocks of n columns each, and use square matrix multiplication. This takes $n^{\omega+2+o(1)}$ time for each λ. Constructing the matrices $A_\lambda, B_\lambda, K_\lambda$ and multiplying the result by K_λ takes only $O(n^4)$ additional operations, which is covered by the same bound.

As there are at most $n^{2-\delta}$ choices of λ, we can construct the entire projected matrix in

$$n^{\omega+4-\delta+o(1)} \tag{10}$$

time. The same argument applies when the projected block occurs in the columns rather than the rows.

This handles the two matrices connecting a projected block to the unprojected third block. The remaining matrix, M_{12}, has both endpoints projected. To construct it, we first apply the above procedure to the pivot pair in V_1, without yet projecting the pivot pair in V_2. This produces a matrix of size at most

$$n^{3-\delta} \times n^3,$$

containing at most $n^{6-\delta}$ entries.

We then apply the full basis transformation from Lemma 3.3 to the pivot pair in V_2, and retain only the indices in Λ. Since the transformation requires $O(n^2 \log n)$ operations on each array of n^2 entries, the total cost is

$$n^{6-\delta+o(1)}.$$

The two projections act on different variables, so this gives exactly the matrix obtained by applying both low-rank decompositions. In particular, we never need to construct the original n^3-by-n^3 matrix.

We have thus constructed all three matrices in $n^{\omega+4-\delta+o(1)}$ time, which is no larger than the multiplication time in (7). □

3.4 Completing the induction

We now combine the two parts to obtain the improved algorithm for every nine-vertex pattern.

Let $\mathcal{T}_m(n)$ denote the maximum running time of our algorithm over all weighted patterns on nine vertices with at most m edges. We prove by induction on m that there exists a constant $\alpha_m > 0$, independent of ω, such that

$$\mathcal{T}_m(n) \le n^{3\omega-\alpha_m+o(1)}. \tag{11}$$

The base case $m = 0$ is immediate, and we may take $\alpha_0 = 1$. Suppose that the claim holds for $m-1$, with some $\alpha_{m-1} > 0$. Choose a sufficiently small constant $\varepsilon > 0$ such that

$$2\varepsilon < \frac{\alpha_{m-1}}{2},$$

and let $\delta > 0$ be the corresponding constant from Lemma 3.3.

Consider a pattern H with at most m edges. If it does not contain two vertex-disjoint wedges, Lemma 3.1 gives an $O(n^5)$ algorithm, which already satisfies (11).

Otherwise, we apply the decomposition to two vertex-disjoint wedges as described above. By Lemma 3.4, the contribution of R_1R_2 can be evaluated in

$$n^{3\omega-(\omega-1)\delta+o(1)}$$

time. All the remaining terms reduce to $O(n^{2\varepsilon})$ weighted sum instances for patterns with fewer edges, each of which can be constructed in $O(n^2)$ time. We therefore obtain

$$\mathcal{T}_m(n) \le n^{3\omega-(\omega-1)\delta+o(1)} + n^{2\varepsilon+o(1)}\big(\mathcal{T}_{m-1}(n) + O(n^2)\big). \tag{12}$$

By the induction hypothesis and our choice of ε, the second term is bounded by

$$n^{3\omega-\alpha_{m-1}+2\varepsilon+o(1)} \le n^{3\omega-\alpha_{m-1}/2+o(1)}.$$

For the first term, recall that $\omega \ge 2$, so $\omega - 1 \ge 1$. Its running time is therefore at most

$$n^{3\omega-\delta+o(1)}.$$

Consequently, both terms have a polynomial saving over $n^{3\omega}$. For example, we may take

$$\alpha_m = \frac{1}{4}\min\{\alpha_{m-1}, \delta\} > 0.$$

This completes the induction.

Since a nine-vertex pattern graph has at most $\binom{9}{2} = 36$ edges, we obtain some constant $\alpha_{36} > 0$ such that the weighted sum of every nine-vertex pattern can be computed in

$$n^{3\omega-\alpha_{36}+o(1)}$$

time. The constant α_{36} depends only on the decomposition lemma and the number of induction steps, and in particular is independent of ω.

The same result also gives an improvement for counting k-cliques whenever k is divisible by nine. Indeed, write $k = 9t$ and partition the k vertices into nine sets of t vertices each. We can regard an assignment to each set as a single value in a domain of size $N = n^t$, absorbing the edges within a set into unary weights and the edges between two sets into pairwise weights. Applying the algorithm above to this weighted nine-vertex instance gives

$$N^{3\omega-\alpha_{36}+o(1)} = n^{(\omega/3-\alpha_{36}/9)k+o(1)}.$$

Thus, we obtain a saving proportional to k over the standard $n^{\omega k/3}$ bound.

This already proves Theorem 1.1 for all k divisible by nine, with $\delta = \alpha_{36}/3$ (after absorbing the $n^{o(1)}$ factor into a slightly smaller δ). To obtain it for every sufficiently large k, write $k = 9t + r$ with $0 \le r \le 8$. We enumerate all $\binom{n}{r} \le n^r$ sets of r vertices, and for each one that forms an r-clique, count the $9t$-cliques in the common neighborhood of its vertices using the algorithm above. Every k-clique is counted exactly $\binom{k}{r}$ times, so dividing the total by $\binom{k}{r}$ gives the number of k-cliques. The running time is

$$n^{\,r+(\omega/3-\alpha_{36}/9)\cdot 9t+o(1)} = n^{\,\omega k/3-\alpha_{36}t+r(1-\omega/3)+o(1)}.$$

Since $\omega \ge 2$, the exponent is at most $\omega k/3 - \alpha_{36}t + r/3$, and for $t \ge 8/\alpha_{36}$ (that is, for $k \ge K_0 = 72/\alpha_{36} + 8$) it is at most $\omega k/3 - \alpha_{36}t/2 + o(1) \le (\omega - \delta)k/3$ for $\delta = \alpha_{36}/9$, say. Both K_0 and δ depend only on α_{36}, and are therefore independent of ω, as claimed.

We remark that the algorithm above is stated in the arithmetic model, as in Theorem 1.2, but it never creates large numbers: all its coefficients are rationals with numerators and denominators bounded by $n^{O(1)}$, and the recursion has constant depth, so for clique counting every intermediate value is, after clearing denominators, an integer of $O(\log n)$ bits. The algorithm can therefore be implemented on the word-RAM with $O(\log n)$-bit words within the same running time up to $n^{o(1)}$ factors, which is the model in which the k-Clique Hypothesis and Theorem 1.1 are stated; the details are given in Appendix A.

4 Proof of the rank-or-swap decomposition

We now prove Lemma 3.3. The construction is a binary specialization of the matrix decomposition used in the recent OpenAI result on the Exponential Time Hypothesis [Ope26a], which builds on earlier polynomial interpolation techniques from matrix rigidity [AW17]. A related decomposition using partial transposes, which is also a synthesis of the OpenAI paper, appears in the recent work of Alman et al. [ALP+26]. We give a direct proof starting from the elementary identity $I = P + (I - P)$.

Proof of Lemma 3.3. Let $n = 2^\ell$, and identify the pair space $[n]^2$ with ℓ copies of the one-bit pair space $\{00, 01, 10, 11\}$. On this four-dimensional space, the swap operator fixes $00, 11$ and exchanges $01, 10$. The four vectors

$$e_{00}, \qquad e_{11}, \qquad e_{01} + e_{10}, \qquad e_{01} - e_{10}$$

form an orthogonal eigenbasis. The first three have eigenvalue 1 under the swap, and the last has eigenvalue -1. We call the first three vectors *symmetric* and the last one *antisymmetric*. The product basis of $\mathbb{R}^{[n]^2} = (\mathbb{R}^4)^{\otimes \ell}$ consists of the $4^\ell = n^2$ tensor products of one such vector per bit position, and we say that a product-basis vector has *type* j if exactly j of its ℓ factors are antisymmetric.

For each position $i \in [\ell]$, let P_i swap the i-th pair of bits, leaving all other positions unchanged, and put $C_i = I - P_i$. For $0 \le m \le \ell$, let W_m be the sum of all products that choose P_i at exactly m positions and C_i at the others. Since $P_i + C_i = I$, we have

$$\sum_{m=0}^{\ell} W_m = \prod_{i=1}^{\ell} (P_i + C_i) = I. \tag{13}$$

The operators W_m are diagonal in the tensor product of the one-bit eigenbases described above. Consider the generating polynomial

$$\sum_{m=0}^{\ell} z^m W_m = \prod_{i=1}^{\ell} (zP_i + C_i).$$

On a basis vector with j antisymmetric factors, its eigenvalue is

$$z^{\ell - j}(2 - z)^j.$$

Indeed, a symmetric factor contributes z, while an antisymmetric factor contributes $2 - z$. In particular, the eigenvalue of $W_{\ell - h}$ is

$$(-1)^j (-2)^h \binom{j}{h}.$$

Fix a sufficiently small constant $\eta > 0$, to be chosen later, and let $d = \lfloor \eta \ell \rfloor$. We will write

$$I = R + T, \qquad T = \sum_{h=0}^{d} c_{\ell - h} W_{\ell - h}, \qquad R = I - T.$$

By the calculation above, the eigenvalue of T on a type-j basis vector is $(-1)^j Q(j)$, where

$$Q(j) = \sum_{h=0}^{d} c_{\ell - h} (-2)^h \binom{j}{h}.$$

Since the polynomials $\binom{j}{h}$, for $0 \le h \le d$, form a basis for polynomials of degree at most d, we can choose Q to be any such polynomial.

Choose an integer a such that the interval $[a, a+d]$ is centered around $\ell/4$, up to rounding, and set

$$Q(j) = (-1)^a \sum_{h=0}^{d} (-2)^h \binom{j-a}{h}, \tag{14}$$

where the binomial coefficients are interpreted as polynomials in j. For every integer $j \in [a, a+d]$, the sum ends at $h = j - a$, and the binomial theorem gives $Q(j) = (-1)^j$. Consequently, T has eigenvalue one on all these basis vectors, and R vanishes on them. The coefficients c_m are obtained by expressing (14) in the basis $\{\binom{j}{h}\}_{h=0}^{d}$, and can be computed explicitly in time polynomial in ℓ.

Rank of R. There are exactly $\binom{\ell}{j}3^{\ell-j}$ product-basis vectors with j antisymmetric positions. Their total number is $4^\ell = n^2$, and their normalized multiplicities are the probabilities of a random variable $J \sim \mathrm{Bin}(\ell, 1/4)$.

Since R vanishes on all basis vectors with $j \in [a, a+d]$, its rank is bounded by the number of vectors outside this interval. The interval is centered at $\ell/4$ and has width $\eta\ell - O(1)$, so every integer outside it is at distance at least $\eta\ell/2 - O(1)$ from the mean. By the standard Hoeffding bound,

$$\Pr[|J - \ell/4| \ge t] \le 2e^{-2t^2/\ell}.$$

Therefore,

$$\mathrm{rank}(R) \le n^2 \exp(-\eta^2\ell/2 + O(1)) \le n^{2-\eta^2/4+o(1)}.$$

Expansion into partial swaps. Every word in T contains at most d factors $C_i = I - P_i$. In the standard basis on a single bit pair, C_i has exactly four nonzero entries. Expanding a word by choosing one of these entries at every position where C_i appears gives a signed combination of operators that swap the two bit values at all other positions and fix all four bits at these positions. These are precisely the partial swaps defined in Section 3.2.

The total number of partial swaps, counting multiplicity, is at most

$$\sum_{h=0}^{d} \binom{\ell}{h} 4^h \le n^{H_2(\eta)+2\eta+o(1)},$$

where H_2 denotes binary entropy. Since $H_2(\eta) + 2\eta$ tends to zero as $\eta \to 0$, for every constant $\varepsilon > 0$ we can choose $\eta > 0$ sufficiently small that the number of partial swaps is at most n^ε. The rank bound above then gives $\mathrm{rank}(R) \le n^{2-\delta}$ for some constant $\delta > 0$.

Structure of the basis. Each one-bit eigenvector has at most two nonzero entries, so each product-basis vector has at most $2^\ell = n$ nonzero entries, which can be enumerated directly from its one-bit factors.

Since R is diagonal in this orthogonal basis, we obtain the rank-one decomposition

$$R = \sum_{\lambda \in \Lambda} d_\lambda b_\lambda b_\lambda^\top, \qquad |\Lambda| = \mathrm{rank}(R) \le n^{2-\delta},$$

where the norms of the basis vectors are absorbed into the coefficients d_λ.

Finally, the change of basis can be performed by applying the same four-dimensional transformation separately at each of the ℓ bit positions. Each position requires $O(n^2)$ arithmetic operations, so the entire transformation takes $O(n^2 \log n)$ time, as required. □

5 Discussion and possible improvements

We disproved the k-Clique Hypothesis by obtaining a universal constant improvement to the conjectured $\frac{\omega}{3}$ coefficient in the running time exponent, which is independent of either k or the matrix multiplication exponent $\omega \geq 2$. Our guiding principle throughout this work and its writing was to obtain and present the preceding statement in the simplest and easiest to explain manner. In particular, we did not make any attempt at optimizing the resulting constant—quite the contrary, we made it significantly worse for the sake of making the arguments cleaner. For example, to be able to get a fixed improvement irrespective of the eventual ω, we never use rectangular matrix multiplication (but instead reduce it to square products); or, in the inductive argument of Section 3, we analyze a universal bound on all patterns with the same number of edges rather than analyzing each pattern separately to optimize bounds (there is only a constant number of patterns, but an absurdly large one). Therefore, we do not state or claim any specific numerical bounds (which, with the presented framework, will seemingly range between minuscule without optimization to tiny with significant optimizations).

Similarly, for simplicity we focus on the case of 9-clique which was the smallest clique size for which we could prove an improvement independent of ω and is also divisible by three and thus immediately gives an improvement for all large enough k, but in fact variants of our arguments can yield an improvement over the previously best k-clique algorithms also for some smaller clique sizes.

We leave as interesting open problems obtaining a *significantly* smaller exponent than $\frac{\omega}{3}k$, and in particular the question of whether linear dependence on k in the exponent is even necessary. As the recent refutation of the Exponential Time Hypothesis [Ope26a, Cla26] suggests, large improvements in the exponent are possible.

6 Reflection on Human-AI interaction

This work was achieved by a combination of a traditional research process with extensive repeated use of a publicly available AI tool (OpenAI's ChatGPT-6, on the web interface without special harnesses, tools, token use, etc.). We note that both of us have our memory feature turned on, so any result we attribute to the AI could have also been affected by previous projects, but nonetheless we will attempt to use this section to describe our AI-use methodology to the best of our understanding.

Our starting point was providing the AI model with previous ideas of ours for an approach to improve the running time for finding 5-cliques specifically. We also gave it the context of recent breakthroughs in fine-grained complexity and in particular the ETH refutation of OpenAI [Ope26a]. After testing some ideas in a back-and-forth conversation, it was eventually able to come up with the first improvement for 5-cliques: it gave an improvement over the $\omega(2,1,2)$ exponent. This improvement initially appeared rather complicated. While the model gave a generalization for some larger clique sizes, the improvements were minuscule and it initially did not report an improvement for clique sizes divisible by three, which is needed for the generalization to infinitely many values of k (since we want an improvement in terms of ω and not rectangular exponents).

What followed was a process of AI-assisted distillation and generalization of that K_5 idea, which eventually resulted in our understanding of the algebraic decomposition of Section 4, which we could separate and cleanly define, as well as trace it to ideas from the OpenAI paper [Ope26a]. Similar ideas (plausibly originating from the same place) also appeared in the three recent AI-assisted refutations of the k-Hyperclique Hypothesis [ALP$^+$26, Fis26, CIMY26].

Much of the eventual algorithmic framework that uses this algebraic decomposition came from us, and understanding the algebraic decomposition and how to use it allowed us to obtain an algorithm

that generalizes to all large enough k, gives an improvement despite only using square matrix multiplication as a black-box (rather than using rectangular matrix multiplication or attempting to directly 'invent' new matrix-multiplication-like tensors; which would make a universal bound independent of ω very unlikely), and is rather natural and simple when taking the algebraic decomposition as a given black-box. We then also used iterative AI use to simplify the algebraic decomposition and write it in a purely combinatorial manner.

One of our (subjective) observations from this process, which we state as it might be useful for other researchers developing an AI-use methodology, is the following: While we all know by now that the obfuscated and cumbersome proofs generated by AI are very difficult for us humans to read and comprehend, we were impressed that they are also difficult for the AI itself to understand. The models were consistently unable to develop further ideas generated by them, but when presented with a clean simplification of the ideas, they (as well as we) were suddenly much more able to take further steps.

Roughly, then, we view the algebraic decomposition of Section 4 as almost-purely the AI's contribution (originating from [Ope26a]), and the algorithmic framework that uses it as a black-box in Section 3 as mainly ours.

References

[ABBK17] Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve. In *FOCS*, pages 192–203. IEEE Computer Society, 2017.

[ABV18] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is Valiant's parser. *SIAM J. Comput.*, 47(6):2527–2555, 2018. Preliminary version in FOCS 2015.

[ADV+25] Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication. In *SODA*, pages 2005–2039. SIAM, 2025.

[ALP+26] Josh Alman, Baitian Li, Neha Pant, Kevin Pratt, and Ryan Williams. Finding hypercliques faster than brute force, 2026. URL: `https://arxiv.org/abs/2611.05082`, `arXiv:2611.05082`.

[AvdBZ26] Alimzhan Amanov, Maxim van den Berg, and Jeroen Zuiddam. Multiplying matrices via polynomials: exposition and barriers. arXiv:2610.30054, 2026. URL: `https://arxiv.org/abs/2610.30054`.

[AW17] Josh Alman and R. Ryan Williams. Probabilistic rank and matrix rigidity. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 641–652, 2017. `arXiv:1611.05558`, `doi:10.1145/3055399.3055484`.

[AW26] Josh Alman and Virginia Vassilevska Williams. Truly subquadratic 3sum and truly subcubic apsp via triangles in sparse lopsided graphs, 2026. URL: `https://arxiv.org/abs/2610.06783`, `arXiv:2610.06783`.

[BW17] Karl Bringmann and Philip Wellnitz. Clique-based lower bounds for parsing tree-adjoining grammars. In *CPM*, volume 78 of *LIPIcs*, pages 12:1–12:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[CDN25] Radu Curticapean, Simon Döring, and Daniel Neuen. Counting small induced subgraphs: Scorpions are easy but not trivial. In *Proceedings of the 33rd Annual European Symposium on Algorithms (ESA)*, volume 351 of *LIPIcs*, 2025. doi:10.4230/LIPIcs.ESA.2025.96.

[Cha10] Timothy M. Chan. A (slightly) faster algorithm for Klee's measure problem. *Comput. Geom.*, 43(3):243–250, 2010.

[Cha16] Yi-Jun Chang. Hardness of RNA folding problem with four symbols. In *CPM*, volume 54 of *LIPIcs*, pages 13:1–13:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[CIMY26] Timothy M. Chan, Reina Itakura, Zhiyuan Ma, and Yuancheng Yu. The 3-uniform hyperclique hypothesis is false, 2026. URL: https://arxiv.org/abs/2611.06813, arXiv:2611.06813.

[Cla26] Claude (Anthropic). A $2^{o(n)}$ time algorithm for 3-SAT, October 2026. Anthropic preprint. URL: https://github.com/anthropics/formal-math/blob/16a5ce41304fcdaea4e0be7c132e1c361dc26da5/eth/paper.pdf.

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[DV22] Mina Dalirrooyfard and Virginia Vassilevska Williams. Induced cycles and paths are harder than you think. In *FOCS*, pages 531–542. IEEE, 2022.

[DVV21] Mina Dalirrooyfard, Thuy-Duong Vuong, and Virginia Vassilevska Williams. Graph pattern detection: Hardness for all induced patterns and faster noninduced cycles. *SIAM J. Comput.*, 50(5):1627–1662, 2021. Preliminary version in STOC 2019.

[EG04] Friedrich Eisenbrand and Fabrizio Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.*, 326(1-3):57–67, 2004.

[Fis26] Nick Fischer. The hyperclique hypothesis is false, 2026. URL: https://arxiv.org/abs/2611.03880, arXiv:2611.03880.

[FKRS25] Nick Fischer, Marvin Künnemann, Mirza Redzic, and Julian Stieß. The role of regularity in (hyper-)clique detection and implications for optimizing Boolean CSPs. In *ICALP*, volume 334 of *LIPIcs*, pages 78:1–78:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[FKRS26] Timo Fritsch, Marvin Künnemann, Mirza Redzic, and Julian Stieß. When does sparsity help for k-independent set in hypergraphs and other Boolean CSPs? In *ICALP*, volume 374 of *LIPIcs*, pages 94:1–94:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2026.

[GK23] Egor Gorbachev and Marvin Künnemann. Combinatorial designs meet hypercliques: Higher lower bounds for Klee's measure problem and related problems in dimensions $d \geq 4$. In *SoCG*, volume 258 of *LIPIcs*, pages 36:1–36:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[JS17] Rajesh Jayaram and Barna Saha. Approximating language edit distance beyond fast matrix multiplication: Ultralinear grammars are where parsing becomes hard! In *ICALP*, volume 80 of *LIPIcs*, pages 19:1–19:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[Koi06] Mikko Koivisto. Optimal 2-constraint satisfaction via sum-product algorithms. *Information Processing Letters*, 98(1):24–28, 2006. doi:10.1016/j.ipl.2005.11.013.

[LVW18] Andrea Lincoln, Virginia Vassilevska Williams, and R. Ryan Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *SODA*, pages 1236–1252. SIAM, 2018.

[NNT26] Taisei Nogami, Yoshiki Nakamura, and Tachio Terauchi. Hardness of regular expression matching with extensions. *CoRR*, abs/2601.03020, 2026. URL: https://arxiv.org/abs/2601.03020.

[NP85] Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.

[Ope26a] OpenAI. The exponential time hypothesis is false, October 2026. OpenAI Math Release preprint. URL: https://github.com/openai/math/tree/main/preprints/The-Exponential-Time-Hypothesis-is-False-October-25-2026.

[Ope26b] OpenAI. An upper bound of 9/4 for the matrix multiplication exponent. Preprint, October 2026, 2026. URL: https://github.com/openai/math/blob/adc7f1241b42e322a6451854ab7e4b4c146bf78a/preprints/Matrix-Multiplication-Nine-Fourths-October-2-2026/paper.pdf.

[Shp26] Amir Shpilka. Exposition of OpenAI's matrix multiplication in exponent 9/4 (and an explicit algorithm). Electronic Colloquium on Computational Complexity (ECCC), TR26-281, 2026. URL: https://eccc.weizmann.ac.il/report/2026/281/.

[SS09] Alexander D. Scott and Gregory B. Sorkin. Polynomial constraint satisfaction problems, graph bisection, and the ising partition function. *ACM Transactions on Algorithms*, 5(4):45:1–45:27, 2009. doi:10.1145/1597036.1597049.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969. doi:10.1007/BF02165411.

[VW10] Virginia Vassilevska Williams and R. Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *FOCS*, pages 645–654. IEEE Computer Society, 2010.

[VW18] Virginia Vassilevska Williams and R. Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *J. ACM*, 65(5):27:1–27:38, 2018.

[Wil05] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2–3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.

[Woe04] Gerhard J. Woeginger. Space and time complexity of exact algorithms: Some open problems (invited talk). In *IWPEC*, volume 3162 of *Lecture Notes in Computer Science*, pages 281–290. Springer, 2004.

A From arithmetic operations to the word-RAM

We explain why the arithmetic-operation bound of Theorem 1.2 yields the claimed word-RAM running time for clique counting. Throughout this appendix, the clique size is fixed, and the word size is $O(\log n)$.

We first bound the coefficients introduced by Lemma 3.3. Write $n = 2^\ell$, padding the domain to the next power of two if necessary, and retain the notation a, d, Q from Section 4. The coefficients $c_{\ell-r}$ are defined by

$$Q(j) = \sum_{r=0}^{d} c_{\ell-r}(-2)^r \binom{j}{r}.$$

In fact, all these coefficients are integers. Indeed, the polynomial identity

$$\binom{j-a}{h} = \sum_{r=0}^{h} \binom{j}{r}\binom{-a}{h-r}$$

and the definition

$$Q(j) = (-1)^a \sum_{h=0}^{d} (-2)^h \binom{j-a}{h}$$

give

$$c_{\ell-r} = (-1)^a \sum_{q=0}^{d-r} (-2)^q \binom{-a}{q}, \qquad 0 \le r \le d.$$

Since generalized binomial coefficients at integer arguments are integers, each $c_{\ell-r}$ is integral. Expanding the operators $W_{\ell-r}$ into partial swaps only introduces signs, so the coefficients of the individual partial swaps are integers as well.

These integers have polynomial magnitude in n. For sufficiently large ℓ, we have $0 \le a \le \ell$ and $d \le \ell$. Consequently,

$$|c_{\ell-r}| \le \sum_{q=0}^{d} 2^q \binom{\ell+d}{q} \le 2^{\ell+2d} = n^{O(1)}.$$

The same estimate, applied to the defining expression for $Q(j)$, shows that $|Q(j)| = n^{O(1)}$ for $0 \le j \le \ell$.

Next consider the rank-one decomposition of R. Every product-basis vector b_λ has entries in $\{0, 1, -1\}$. If exactly m of its one-bit factors are of the form $e_{01} + e_{10}$ or $e_{01} - e_{10}$, then

$$b_\lambda^\top b_\lambda = 2^m, \qquad 0 \le m \le \ell.$$

If j of its factors are antisymmetric, its eigenvalue under R is $1 - (-1)^j Q(j)$. Thus the coefficient in the rank-one decomposition is

$$d_\lambda = \frac{1 - (-1)^j Q(j)}{2^m}.$$

Its numerator has magnitude $n^{O(1)}$, and its denominator is a power of two dividing $2^\ell = n$. In particular, no factorial denominators are needed.

Now consider the weighted nine-vertex instance obtained from the clique-counting reduction in Section 3.4. Its initial unary and pairwise weights belong to $\{0, 1\}$. The recursion has depth at most 36. At each level, new weights are products of constantly many previous weights and decomposition

coefficients, and there are only polynomially many subinstances. The matrix constructions likewise use products of constantly many weights and sums of polynomially many such products.

It follows inductively that every weight, every entry of a matrix supplied to fast matrix multiplication, and every returned value is a rational number of the form

$$\frac{z}{2^q}, \qquad |z| \le n^C, \qquad q \le C \log n,$$

for a constant C independent of n. For the initial nine-vertex problem, C is absolute; after grouping vertices to count k-cliques, it may depend on the fixed value of k. Polynomially many additions preserve these bounds after increasing C.

To implement a matrix product, clear the power-of-two denominators in each input matrix. The resulting integer matrices have entries of $O(\log n)$ bits. Exact multiplication of $N \times N$ integer matrices with such entries can be performed in $N^{\omega+o(1)} n^{o(1)}$ word-RAM time; the polynomially bounded matrix dimensions used here allow the $n^{o(1)}$ overhead to be absorbed into the existing $n^{o(1)}$ factor. Rescaling the output restores the desired rational product. All other arithmetic on the bounded rational values above takes $n^{o(1)}$ time per operation.

Therefore, the arithmetic-operation bound of Theorem 1.2 translates to the same word-RAM running time up to an $n^{o(1)}$ factor. The constant-factor losses from padding the domain and the final exact divisions used to recover the number of unlabeled cliques do not affect the exponent. Finally, as in Section 3.4, the $n^{o(1)}$ factor can be absorbed into a slightly smaller constant exponent saving, giving Theorem 1.1.