The Complexity of Average-Case Dynamic Subgraph Counting

Monika Henzinger [*], Andrea Lincoln [†], Barna Saha [‡]

October 29, 2021

Abstract

Statistics of small subgraph counts such as triangles, four-cycles, and *s-t* paths of short lengths reveal important structural properties of the underlying graph. These problems have been widely studied in social network analysis. In most relevant applications, the graphs are not only massive but also change dynamically over time. Most of these problems become hard in the dynamic setting when considering the worst case. In this paper, we ask whether the question of small subgraph counting over dynamic graphs is hard also in the average case.

We consider the simplest possible average case model where the updates follow an Erdős-Rényi graph: each update selects a pair of vertices (u, v) uniformly at random and flips the existence of the edge (u, v). We develop new lower bounds and matching algorithms in this model for counting four-cycles, counting triangles through a specified point *s*, or a random queried point, and *st* paths of length 3, 4 and 5. Our results indicate while computing *st* paths of length 3, and 4 are easy in the average case with $O(1)$ update time (note that they are hard in the worst case), it becomes hard when considering *st* paths of length 5.

We introduce new techniques which allow us to get average-case hardness for these graph problems from the worst-case hardness of the Online Matrix vector problem (OMv). Our techniques rely on recent advances in fine-grained average-case complexity. Our techniques advance this literature, giving the ability to prove new lower bounds on average-case dynamic algorithms.

[*]This research was supported by the Austrian Science Fund (FWF) and netIDEE SCIENCE project P 33775-N and done in partwhile visiting Stanford University

[†]University of California Berkeley. This work is supported partly by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship. This work was also supported by the Simons NTT research fellowship.

[‡]University of California Berkeley. This work is supported partly by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship. This work was also supported by the Simons NTT research fellowship.

1 Introduction

It is a broad criticism of worst-case analysis that it is overly pessimistic and this also applies to the worst-case analysis of dynamic algorithms. For many dynamic graph problems this is not a shortcoming of algorithms designers, rather it is due to the existence of strong conditional lower bounds that preclude fast algorithms [1, 13]. For example, maintaining shortest paths in a dynamic graph of n vertices and m edges with subpolynomial query time requires $\Omega(m^{1-\varepsilon})$ time per update operation for any small $\varepsilon > 0$ and even counting s-t paths of length 3 requires an update time of $m^{1/2-o(1)}$ if the Online Matrix Vector (OMv) conjecture holds [13, 12]. Most real world networks are dynamic in nature. However, the worst case update sequences that are used to prove the conditional lower bounds are unlikely to arise in practice. These concerns motivate the study of dynamic algorithms beyond the worst case.

In the static world, the simplest possible average case model to study graph algorithms is Erdős-Rényi graphs (every possible edge (u,v) is included or excluded iid with probability $1/2$). Can we consider such a model in the dynamic setting? Suppose the adversary can choose when to update and when to query, but cannot control the precise nature of updates. Every update picks a uniformly random pair of vertices (u,v) and *flips* the existence of the edge (u,v). That is if we have a graph G with edge set E and we update (u,v) then we add (u,v) if $(u,v) \notin E$ and we delete (u,v) if $(u,v) \in E$. In this model we consider many possible types of queries, in this paper we restrict our attention to counting small subgraphs. This weakened adversary is similar to an Erdős-Rényi graph for offline problems. Notably, when the initial graph is Erdős-Rényi and any series of *average-case* updates are made in our model, the new updated graph remains drawn from an Erdős-Rényi distribution. This leads to a significantly weakened adversary which should lead to dynamic algorithms with better running time bounds! In the static setting, many problems become easy on Erdős-Rényi graphs such as APSP [7], whereas problems like k-clique remain hard [5]. Can such a dichotomy exist in the dynamic world?

Open question: Which problems become easy in the dynamic average-case model, and which remain hard?

Surprisingly we show that even in this most natural average case model where the adversary is significantly weaker than the worst-case adversary, some problems remain hard. In this paper, we focus on the problems of counting small subgraphs such as triangles and four cycles through specified nodes, and s-t paths of length $3,4$ and 5. These problems, especially counting triangles and four cycles are heavily studied in the data mining community, and reveal important structural properties of the underlying graphs [17, 16, 15]. Counting s-t paths of small length are intrinsically connected to the fundamental problem of computing shortest paths especially for small-diameter graphs, a property shared by many real-world networks [6]. We observe an interesting change in hardness when we move from s-t paths of length 4 to 5: while counting s-t paths of length 4 is easy on average, it becomes hard at length 5. Such a hardness shift is not known in the worst case.

In the traditional RAM model counting constant size subgraphs has the same complexity in both average-case Erdős-Rényi and worst case [9, 5]. Another interesting result of our work is that this is *not true* in the dynamic setting.

It should be noted here that in dynamic algorithm design, a worst case update time is often contrasted with amortized update time, whereas *we use worst case to refer to inputs that do not come from a fixed distribution and can be arbitrarily hard* and whether the analysis uses amortization or not does not matter.

Our work is motivated by the pioneering work of Alberts and Henzinger who define the first "average" case model for dynamic graph algorithms and analyzed the performance of dynamic algorithms for a variety of graph problems in that model [2]. In their model the adversary is somewhat stronger in that the adversary can choose in a worst-case fashion if they will make an edge deletion, edge insertion, or query. If an edge deletion is made, a uniformly random edge from the graph is selected and deleted. If an edge insertion is made, then a uniformly random non-edge is selected and an edge is inserted there. Note that this allows the adversary to keep the graph sparse or dense, and allows the adversary to change behavior based on what edges exist in the graph. In contrast, our model directly emerges from the Erdős-Rényi graphs, and is, in some sense, the easiest to design algorithms and hardest to design lower bounds. We chose this model as our main results in this paper are the lower bounds.

The average-case hardness of dynamic algorithms has not yet been well explored as it used to be difficult to prove lower-bounds due to the lack of average-case fine-grained lower-bound techniques. However, there has been recent progress in average-case fine-grained complexity for *static* graph problems (e.g. [5, 9]). We use and improve upon this recent progress. Section 1.2 describes the difficulties of applying the prior work to the dynamic setting. We give tight upper and lower bounds for a variety of dynamic graph problems. Interestingly, our lower bounds are derived

	Average-Case Lower Bound			Worst-Case Lower Bound		
	$P(n)$	$U(n)$	$Q(n)$	$P(n)$	$U(n)$	$Q(n)$
#s-t 3 path (–)([12])	–	–	–	$O(poly(n))$	$\tilde{\Omega}(n)$	$O(n^{2-\varepsilon})$
#s-t 4 path (-)(S7)	–	–	–	$O(poly(n))$	$\tilde{\Omega}(n)$	$O(n^{2-\varepsilon})$
#s-t 5 path (S6)(“)	$O(n^{3-\varepsilon})$ $O(n^{3-\varepsilon})$	$\tilde{\Omega}(n)$ $O(n^{1-\varepsilon})$	$O(n^{2-\varepsilon})$ $\tilde{\Omega}(n^2)$	“ “	“ “	“ “
#4-Cycles (S4)(“)	$O(n^{3-\varepsilon})$ $O(n^{3-\varepsilon})$	$\tilde{\Omega}(n)$ $O(n^{1-\varepsilon})$	$O(n^{2-\varepsilon})$ $\tilde{\Omega}(n^2)$	“ “	“ “	“ “
#s4-Cycles (-)(S7)	–	–	–	$O(poly(n))$	$\tilde{\Omega}(n)$	$O(n^{2-\varepsilon})$
#△ Through a Queried Point (S5)(“)	$O(n^{3-\varepsilon})$ $O(n^{3-\varepsilon})$	$\tilde{\Omega}(n)$ $O(n^{1-\varepsilon})$	$O(n^{1-\varepsilon})$ $\tilde{\Omega}(n)$	“ “	“ “	“ “
#s-△ (-)([13])	–	–	–	$O(n^{3-\varepsilon})$	$\tilde{\Omega}(n)$	$O(n^{2-\varepsilon})$
#△ (S5)(“)	$O(n^{2-\varepsilon})$ $O(n^{2-\varepsilon})$	$O(n^{\omega-2-\varepsilon})$ $\tilde{\Omega}(n^{\omega-2})$	$\tilde{\Omega}(n^{\omega})$ $O(1)$	“ “	“ “	“ “

Table 1: Our lower bound results. We use $P(n)$ for preprocessing time, $U(n)$ for update time, and $Q(n)$ for the query time. Next to the problem name we either give a reference or mark the section in which the lower bound is proven ((S#) denotes section #). The symbol $-$ or $(-)$ indicates that there is no non-trival lower bound as there is an algorithm with $O(1)$ average time, “ indicates that the worst-case lower bound matches our average-case lower bound. Bounds in gray cells are from prior work. Our lower bounds in white cells come from the OMv hypothesis, in blue cells from the k-clique hypothesis when $k = 3$.

from worst-case hard problems. Specifically, we use the Online Matrix vector (OMv) hypothesis [13] to prove lower bounds, and show that average case OMv is hard assuming the worst case hypothesis.

DEFINITION 1.1. *In OMv, we are given a matrix $M \in \{0,1\}^{n\times n}$ then we are given n vectors v_i, each in $\{0,1\}^n$, in an online fashion. After each v_i is given one must return a vector $\vec{y}_i$ where $\vec{y}_i[\ell] = \min((Mv_i)[\ell], 1)$. The algorithm must return the correct vector with probability at least $2/3$.*

Conjecture 1. [Conjecture 1.1 from [13]] For any constant $\varepsilon > 0$, there is no $O(n^{3-\varepsilon})$-time algorithm that solves OMv with error probability at most $1/3$ in the word-RAM model with $O(\log n)$ bit words.

1.1 Our results We present tight upper and lower bounds for several subgraph counting problems which include counting triangles that contain a particular node (#s-triangles), or a uniformly random queried node, counting 4-cycles, and s-t paths of length up to 5. The problem of counting triangles that use a random queried point (#△ through a queried point) has the following queries: when making a query a uniformly random node u is selected and the query will return a tuple $(u, \text{number of triangles which contain } u)$. So the adversary has no control over which node is queried, but, they do know which node they receive a count from. We present all of our results in Tables 1 and 2.

There are four interesting result categories:

(1) *Average case dynamic upper bound matches the average case conditional lower bound up to a factor of n^{ε}.* All our problems except for triangle counting fall into this category. This means we settle the average-case dynamic complexity for these problems, up to a factor of n^{ε}.

(2) *Average case dynamic lower bound equals worst-case dynamic lower bound.* This means these problems are equally hard in the worst-case and the average case setting. #s-t 5 paths, #4-cycles, counting triangles through a random queried point fall in this category.

(3) *Average case dynamic upper bound beats worst-case conditional lower bound.* This is an interesting category as for these problems the worst-case conditional lower bound is “overly pessimistic”, i.e., the worst-case sequence is unlikely to arise in our average-case model. Problems #s-t 3 paths, #s-t 4 paths, #s 4-cycles and #s-triangles fall in this category. These are all problems for which we give constant time average-case algorithms.

(4) *Average case dynamic upper bound beats worst-case upper bound.* These are all problems for which we can give a better analysis knowing that the updated edge is chosen randomly. This is the case for all problems, except

	Average-Case Algorithm			Worst-Case Algorithm		
	$P(n)$	$U(n)$	$Q(n)$	$P(n)$	$U(n)$	$Q(n)$
#s-t 3 path (S7)([12])	$O(n^2)$	$O(1)$	$O(1)$	$O(m_0^{3/2}))$	$O(m^{1/2})$	$O(1)$
#s-t 4 path (S7)([10])	$O(n^\omega)$	$O(1)$	$O(1)$	$O(m_0 n)$	$O(n^2 \log^3 n)$	$O(p+\log n)$
#s-t 5 path (S8)([10])	$O(n^\omega)$	$O(n)$	$O(1)$	$O(m_0 n)$	$O(n^2 \log^3 n)$	$O(p+\log n)$
	$O(n^\omega)$	$O(1)$	$O(n^\omega)$	$O(m_0 n)$	$O(n^2 \log^3 n)$	$O(p+\log n)$
#4-Cycles (S9)([12])	$O(m^{5/3})$	$O(\min(n, m^{2/3}))$	$O(1)$	$O(m_0^{5/3})$	$O(m^{2/3})$	$O(1)$
	$O(n^\omega)$	$O(1)$	$O(n^\omega)$	$O(m_0^{5/3})$	$O(m^{2/3})$	$O(1)$
#s 4-Cycles (S7)([12])	$O(n^\omega)$	$O(1)$	$O(1)$	$O(m_0^{5/3})$	$O(m^{2/3})$	$O(1)$
#△ Through a Queried	$O(n^\omega)$	$O(n)$	$O(1)$	$O(n^\omega)$	$O(n)$	$O(1)$
Point (S11)([12])	$O(m_0^{3/2})$	$O(\sqrt{m})$	$O(\sqrt{m})$	$O(m_0^{3/2})$	$O(\sqrt{m})$	$O(\sqrt{m})$
#s-△ (S10)([12])	$O(n^2)$	$O(1)$	$O(1)$	$O(m_0^{3/2})$	$O(m^{1/2})$	$O(1)$
#△ ([14]) ([14])	$O(m_0+n)$	$O(m^{1/2})$	$O(1)$	$O(m_0+n)$	$O(m^{1/2})$	$O(1)$

Table 2: Our upper bound results, where m denotes the current number of edges, m_0 the initial number of edges, and p the number of counted paths. We use $P(n)$ for preprocessing time, $U(n)$ for update time, and $Q(n)$ for the query time. We mark where the algorithm are from by giving either a reference of the section number (S#) denotes section #). We have grayed out the cells of all results that come from other work and cite them in the table as ([#]). The papers [12] and [14] give dynamic algorithms with $O(\sqrt{m})$ time per update, which is $O(n)$ time in the dense Erdős-Rényi graphs we study.

for #△ (for which we do not give a new algorithm) and #△ Through a Queried Point, if one assumes that the graphs are dense. Note that the worst-case upper bounds for #s-t 4 paths and #s-t 5 paths follow from the fully dynamic all-pairs-shortest-path algorithm of Demetrescu and Italiano [10] if one observes that the algorithm actually maintains a set S of shortest paths (and up to $O(\log n)$ additional paths) between any pair of vertices. Testing whether a path of S still exists and has length 4 (resp. 5) takes time linear in its size, which is $O(1)$. Thus determining the number of s-t paths of length 4 (resp. 5) takes time linear in their number plus $O(\log n)$.

This leads to the following result for all problems we study: There is a clear dichotomy for them: the average-case dynamic complexity is either constant or the dynamic average-case hardness is equal to worst-case hardness.

There are two other interesting dichotomies we want to point out: (a) For #s-t paths up to length 4 there are algorithms with constant time per operation in our average-case model. However, the problem of counting paths of length 5, i.e. #s-t 5 paths, becomes hard, i.e. we give a conditional lower bound showing that there is no fast algorithm for s-t path in an average-case dynamic graph setting. The reduction is from OMv. We also show that this lower bound is tight as we present a dynamic algorithm for #s-t 5 path with matching running time. This shows that the switch from easy to hard happens at exactly paths of length 5. (b) In the traditional RAM model counting constant size subgraphs has the same complexity in both average case and worst case [7, 9, 5]. However, in the dynamic setting this is not the case for all problems that are in category (3) above.

1.2 New Techniques To show our results we develop a set of new techniques, namely *biased updates*, *average-case hardness of parity OMv*, *dynamic inclusion-edgesclusion*, and *fast average-case subgraph counting techniques*. We will describe each technique and its use briefly. First, a core challenge to overcome when getting average-case lower bounds from dynamic problems which are hard in the worst-case is the question of how to "hide" the dynamic updates. By that we mean that we have to make the update sequence "look" random, even though it originates from a worst-case sequence. The naive approach would cause us to insert and delete $\Theta(n^2)$ random edges for every update! This is far too expensive. We avoid this issue by using the algebraic structure of OMv to efficiently hide the updates as we describe below.

We also use the problem of OuMv. We give a formal definition in section 2. Informally it is a problem where you are given an n by n matrix M to pre-compute over. Then, n queries are given each is a single pair of vectors $\vec{u}_i$ and $\vec{v}_i$ one must return the value $\vec{u}_i^T \cdot M \cdot \vec{v}_i$ dynamically for query. They OuMv hypothesis states that the n queries must

take at least $n^{3-o(1)}$ time. This hypothesis is implied by the OMv hypothesis [13]. See Section 2 for a more formal discussion.

Average-case hardness for OMv. We present the theorem statements for the average-case hardness of OMv and OuMv in section 2 and the proofs in section A. To get the above lower bounds we use the average-case hardness of *parity OMv* and its variant *parity OuMv*, whose hardness we prove first. Using the hardness of the parity versions the core idea is that we can add random matrices and vectors to our worst-case matrices (M) and vectors ($\vec{v}_i$) and use the linearity of XOR to return the $M\vec{v}_i$ we originally wanted to find. So, for example we are given a worst-case $M \in \{0,1\}^{n\times n}$ and n vectors $\vec{v}_i \in \{0,1\}^n$ and we are asked to return $\min((M\vec{v}_i)[j],1)$ for all $j \in [0,n-1]$. We show that we can solve OMv with parity OMv, where we are instead asked to return $(M\vec{v}_i)[j] \mod 2$ for all $j \in [0,n-1]$. We can then solve parity OMv with four calls to instances that individually look indistinguishable from uniformly random $\{0,1\}$ matrices and vectors using the following observation: Select a uniformly random $R \in \{0,1\}^{n\times n}$ and uniformly random $\vec{r}_i \in \{0,1\}^n$. Then consider the following queries: $(R \oplus M)(\vec{v}_i \oplus \vec{r}_i)$, $(R \oplus M)(\vec{r}_i)$, $(R)(\vec{v}_i \oplus \vec{r}_i)$, and $(R)(\vec{r}_i)$. These queries are correlated, but each one looks uniformly random. If we have answers to all four queries and add them mod two, this gives $M\vec{v}_i \mod 2$. While this is the core idea of the reduction, it is hard to use it to get lower bounds for graph problems. That is why we developed the idea of a biased updates version, presented next.

Biased updates (Main New Technique) In section 2 we present the biased updates problem. We previously described average-case OMv with *fixed* uniformly random M and uniformly random $\vec{v}_i$ vectors. However, imagine trying to use this problem to generate hardness for a graph problem. In our graph problem when we make random updates we are taking a random pair of nodes (u,v) and flipping the edge between them (deleting it if it already exists and adding the edge if it doesn't exist). This introduces two problems.

• First, the standard way to represent a vector $\vec{v}_i$ in a reduction uses $\Theta(n)$ edges whereas M uses $\Theta(n^2)$ edges. If we want to do random updates until the edges representing $\vec{v}_i$ represent a new random vector it will require flipping $\Omega(n^2)$ edges in the graph. That is because the probability that any given update flips a vector-representing edge is $O(1/n)$. Since, we need $\Omega(n)$ flips of *vector-representing edges* to randomize $\vec{v}_i$, we would need at least $\Omega(n \cdot (1/n)^{-1})$ edge flips before we could hope to randomize $\vec{v}_i$. So we need some way of representing $\vec{v}_i$ so that an edge representing $\vec{v}_i$ and M are equally likely to be flipped. To deal with this problem, we represent our vectors in a new way in the graph, namely by $\Theta(n^2)$ edges: Each bit of the vector $\vec{v}_i[j]$ is represented with n edges. So, after x random updates to the graph we expect $\Theta(x)$ updates to the vector $\vec{v}_i$. Now when we make a series of $\Theta(n\lg^2(n))$ updates we effectively randomize the vector $\vec{v}_i$ to a new vector $\vec{v}_{i+1}$, and make a small number of changes to M. We use this technique in Sections 5, 4, and 6.

• The second problem is that when we do random edge flips, as mentioned above, some of these will flip the edges representing M. In the original description of the worst-case OMv we have M to be fixed, but in our average-case model every edge including the edges representing M can be flipped. To deal with this, we use a technique similar to the parity idea sketched below:We create three correlated update sequences (that lead to three correlated dynamic graphs) that correspond to matrices $M_1 = M \oplus (\Delta_1 \oplus \Delta_2)$, $M_2 = M \oplus (\Delta_2 \oplus \Delta_3)$, and $M_3 = M \oplus (\Delta_3 \oplus \Delta_1)$ and ask a dynamic graph query that in effect returns $M_j\vec{v}_i$ with $j = 1,2,3$ in each of them. Note that combining these answers with XOR answers return the original OMv query $M\vec{v}_i \mod 2$. Note further that each of these update sequences "looks random" for its data structure, but due to the linearity of XOR, their XOR combination results in the original matrix M.

We describe next (in a simplified way) how to create Δ_1, Δ_2, and Δ_3 and the random update for $\vec{v}_i$: We want to generate a series of updates that *individually* look random, but across all three of our calls are correlated. To do this we make $\Theta(n(\log^2 n))$ coin flips where every head counts towards an update for $\vec{v}_i$ and every tail will become an update for M. Let β_v represent the edge updates corresponding to edges representing $\vec{v}_i$, and Δ_1, Δ_2, and Δ_3 each represents half of the random edge updates corresponding to edges representing M. Now, we generate three edge update sequences. One involves β_v, Δ_1, and Δ_2 corresponding to M_1, the second involves β_v, Δ_2, and Δ_3 corresponding to M_2, and the last involves β_v, Δ_1, and Δ_3 corresponding to M_3.

Note that M and β_v appear in all three instances. Further note that Δ_1, Δ_2, and Δ_3 all appear in exactly two instances. By XORing we will therefore get back the value $M(\vec{v}_i \oplus \beta_v)$ which we desired.

Dynamic Inclusion-Edgesclusion. As will be evident in most lower bound reductions, we need to create a k-partite graph. However, we are interested in general Erdős-Rényi graphs with no restrictions on edges. To deal with this issue we introduce in section 3 a generalization of the inclusion-edgesclusion technique of [9] which now works

for dynamic graphs. Most of our reductions happen in two steps. First, we prove the problem is hard on a k-partite graph (where we restrict some edges to not exist and not be randomly updated). Then, we use our Dynamic Inclusion-Edgesclusion theorem from section 3 to show hardness for the dynamic Erdős-Rényi setting. Note that the technique of [9] cannot be applied directly because it would need to be run on each update and it requires $\Theta(n^2)$ time, which is far too slow in the dynamic setting. Additionally, we generalize the allowed graphs to include those with fixed nodes (e.g. s-t paths and s triangles).

The core idea of dynamic inclusion-edgesclusion is that we can take k-partite Erdős-Rényi graphs and make them look fully Erdős-Rényi by adding random edges. We can solve counting problems on k-partite Erdős-Rényi graphs with algorithms that run on Erdős-Rényi graphs via multiple calls to correlated graphs by exploiting the linearity of XOR. We present a small illustrative example for s-t three paths in section 3 to help build intuition. This concept is used in all of our lower bounds in sections 5, 4, and 6.

Fast Average-Case Subgraph Counting. For the upper bound we use a simple idea from [2] to get faster algorithms: construct data structures that are fast for likely updates and can be slow for unlikely updates. Specifically, if we want to count the number of subgraphs containing a fixed vertex s then the key observation is that updates of edges incident to s will be unlikely, and can thus be slow, while updates of all other edges are likely and the data structure needs to be fast to process them. Assume, for example, we want to maintain the number of length-2 paths from a fixed node s to any other node v, where v is given as query parameter. This problem is conditionally hard in the worst-case (from OMv no less), however, as we sketch now, it is easy on average. Given $O(n^2)$ preprocessing time we can complete updates in expected $O(1)$ time and answer queries in expected amortized $O(1)$ time. Simply keep at each node $v \neq s$ the count $c(v)$ of length-2 paths to s and also a bit indicating whether v is a neighbor of s. Note that the update of an edge (u,v) only effects the counts of numbers of two paths from u and v to s, and *not* for any other nodes, and updating these counts takes $O(1)$ worst-case time. It takes $O(n)$ worst-case time to update counts when we add or remove an edge (s,u). However, this happens with probability $1/n$ resulting in expected amortized time $O(1)$. The data structures for the problems in Table 2 are more refined, but all are based on this idea. They are represented in Sections 10, 7, 8, and 9.

2 Average-Case OMv, OuMv, and Biased Updates

In this paper we show hardness for average-case problems based on the popular dynamic OMv conjecture, which is a conjecture for the worst-case complexity. To turn the worst-case complexity bound into an average-case complexity bound we first show the hardness of parity OMv and parity OuMv given the worst-case hardness of traditional OMv. In this section we present the theorems and lemmas but defer all proofs except for the proof of our biased updates Theorem to Appendix A. The core idea of all proofs in the section is that we can make several correlated calls and use the linearity of XOR to solve the worst-case problem.

OMv and OuMv Definitions First we will define the problems of OMv and OuMv.

DEFINITION 2.1. (OMV) *In OMv we are given a matrix $M \in \{0,1\}^{n\times n}$ then we are given n pairs of v_i each in $\{0,1\}^n$ in an online fashion. After each v_i is given one must return a vector $\vec{y}_i$ where $\vec{y}_i[\ell] = \min((Mv_i)[\ell], 1)$. The algorithm must return the correct vector with probability at least $2/3$. The OMv hypothesis is that OMv requires at least $n^{3-o(1)}$ time. (Conjecture 1.1 from [13])*

In parity OMv *one must instead return $Mv_i \mod 2$ with probability at least $2/3$. The parity OMv hypothesis is that parity OMv requires at least $n^{3-o(1)}$ time.*

We define input distribution to average-case OMv, D_{OMv}, as the distribution where M is drawn uniformly at random from $\{0,1\}^{n\times n}$ and each $\vec{v}_i$ is drawn uniformly at random and iid from all possible $\{0,1\}^n$ vectors. The problem of (uniform) average-case OMv *then asks for a solution of OMv. The problem of* (uniform) average-case parity OMv *requires solving parity OMv on inputs drawn from D_{OMv}.*

DEFINITION 2.2. (OUMV) *In OuMv we are given a matrix $M \in \{0,1\}^{n\times n}$ then we are given n pairs of (u_i, v_i) each in $\{0,1\}^n$ in an online fashion. After each (u_i, v_i) is given we must return the value $\min\left(u_i^T M v_i, 1\right)$ with probability at least $2/3$. The OuMv hypothesis is that OuMv requires at least $n^{3-o(1)}$ time. (Definition 2.6 from [13])*

In parity OuMv *one must instead return $u_i^T M v_i \mod 2$ with probability at least $2/3$. The parity OuMv hypothesis is that parity OuMv requires at least $n^{3-o(1)}$ time.*

We define input distribution to average-case OuMv, D_{OuMv}, as the distribution where M is drawn uniformly at random from $\{0,1\}^{n\times n}$ and each $\vec{v}_i$ and $\vec{u}_i$ is drawn uniformly at random and iid from all possible $\{0,1\}^n$ vectors. The problem of (uniform) average-case OuMv requires solving parity OMv on inputs drawn from D_{OuMv}. Note that M is randomly chosen at the beginning but remains unchanged in the average-case parity OMv and OuMv.

Note that Theorem 2.7 from [13] shows that the OMv hypothesis implies the OuMv hypothesis. We will show in Lemma 2.1 that parity OuMv and parity OMv are also hard. The omited proofs of this section are all in Appendix A.

LEMMA 2.1. *With x calls to parity OMv we can answer an instance of OMv with probability at least $1-2^{-x}\cdot n$. With x calls to parity OuMv we can answer an instance of OuMv with probability at least $1-2^{-x}\cdot n$. Thus for $x \geq \log(3n)$ the parity OMv hypothesis and the parity OuMv hypothesis are implied by the OMv hypothesis.*

We now present a lemma that the hardness of worst case parity OuMv implies the hardness of uniform average-case parity OuMv. This proof is an "online version" of the Blum, Luby, and Rubinfeld proof that matrix multiplication is hard on average [4]. We can show that parity OMv and parity OuMv (see definitions 2.1 and 2.2) are hard on average with this inclusion/exclusion technique.

LEMMA 2.2. *An algorithm for (uniform) average-case parity OuMv (resp. OMv) that succeeds with probability $1-\varepsilon$ in time $T(n)$ implies a worst-case algorithm for parity OuMv (resp parity OMv) in time $O(T(n))$ that succeeds with probability $1-8\varepsilon$ (resp. $1-4\varepsilon$).*

Now, we can consider what a fast algorithm for the (uniform) average-case parity OuMv problem implies about worst-case OuMv. Note that OMv and OuMv are defined with n vectors or pairs of vectors, so we can encapsulate all of the updates and query times into a single running time for all operations.

THEOREM 2.1. *Given a dynamic algorithm $\mathscr{A}$, for (uniform) average-case parity OuMv (resp. parity OMv) which succeeds with probability at least $16/17$ and runs in time $T(n)$ we can solve worst-case OuMv (resp. OMv) with probability at least $1-2^{-\Omega(\lg^2(n))}$ in time $\tilde{O}(T(n))$.*

2.1 Biased Updates Now we are going to prove, as described in our new techniques section, our biased updates theorem. We will show that biased OuMv (defined below) is hard from the worst-case OMv hypothesis via (uniform) average-case parity OuMv. In Appendix A we present proofs for biased OMv.

DEFINITION 2.3. (BIASED AVERAGE-CASE OUMV) *We call the following problem* biased average-case OuMv. *Let n be a positive integer. The initial matrix M_0 and vectors $\vec{u}_0$ and $\vec{v}_0$ are chosen uniformly at random from all possible such n-dimensional Boolean vectors and $n\times n$-dimensional Boolean matrices. Now, $M_{i+1},\vec{u}_{i+1},\vec{v}_{i+1}$ are created from $M_i,\vec{u}_i,\vec{v}_i$ by flipping $n\lg^2(n)$ bits. Each bit flip that occurs has the following distribution: each bit in M_i is flipped with probability $\frac{1}{3n^2}$, each bit in $\vec{u}_i$ and $\vec{v}_i$ is flipped with probability $\frac{1}{3n}$. In the* biased average-case OuMv *problem for $1\leq i\leq n\lg^4(n)$ right after the construction of $M_i,\vec{u}_i,\vec{v}_i$ we must return a one if $(\vec{u}_i^T M_i\vec{v}_i)$ is non-zero and a zero otherwise. In the* biased average-case parity OuMv *problem for $1\leq i\leq nn\lg^4(n)$ right after the construction of $M_i,\vec{u}_i,\vec{v}_i$ we must return $(\vec{u}_i^T M_i\vec{v}_i) \mod 2$.*

Note that in biased average-case problems the matrix M used for producing the output can change. In a similar way we can define *biased average-case parity OMv*, see Appendix A Definition A.1. In the rest of the paper we only make use of biased average-case parity OuMv and biased average-case parity OMv.

We use the total variation distance (TVD) to describe how different some of our distributions are.

DEFINITION 2.4. (TOTAL VARIATION DISTANCE (TVD)) *Let D and D' be two probability distributions, and let X be the complete set of possible outcomes from both distributions, that is the union of the supports of both distributions. Then the total variation distance is*

$$\sum_{x\in X} |\,(Pr_{Y\sim D}[Y=x]) - (Pr_{Y\sim D'}[Y=x])\,|.$$

The TVD can be thought of as a bound on the probability with which one can distinguish the distributions. If TVD is δ, then intuitively $1-\delta$ of the time the sample is from the overlap of the distributions.

Both biased average-case parity OuMv [OMv] and average-case parity OuMv [OMv] start with random matrices and vectors. We have two differences to overcome. First, the matrix M of biased average-case OuMv [OMv] changes and second, all changes are made with bit flips (instead of freshly random vectors). For the first challenge we are going to create three instances of biased OuMv [OMv], use the linearity of XOR, and a clever set of updates. For the second challenge we use the fact that enough random bit flips of a vector creates a vector that has a very small TVD from a uniformly random vector.

THEOREM 2.2. *If a dynamic algorithm $\mathscr{A}$ solves biased average-case parity OuMv (resp. parity OMv) in time $U(n)$ per update and $Q(n)$ per query with probability at least $59/60$ then worst-case OuMv (resp. OMv) can be solved in time $U(n)n^{2+o(1)} + Q(n)n^{1+o(1)} + n^{2+o(1)}$.*

Proof. We use the reduction of Theorem 2.1 from worst-case OuMV to (uniform) average-case parity OuMv problem and, thus, assume that we are given an instance of (uniform) average-case parity OuMv with matrix M fixed and the sampled vectors $(\vec{u}_i, \vec{v}_i)$ from $i = 0, 1, \ldots, n$ given in an online manner. We will show how to turn this into three parity OuMv problems S_1, S_2, and S_3, each with the same set of vectors $(\vec{u'}_i, \vec{v'}_i)$ but with different but correlated i-th matrices such that the XOR of their individual answers gives the answer for the given instance of the (uniform) average-case parity OuMv problem. Each S_k with $k = 1, 2, 3$ will have a small TVD from a *biased* average-case parity OuMv problem. We will use Lemma C.1 to show that this implies a high probability of success on the *biased* average-case parity OuMv distribution.

The initial matrix and vectors M_0, $\vec{v}_0$, and $\vec{u}_0$ are also the initial matrix and vectors for each S_k. Now for $0 \leq i \leq n-1$ given the vector $(\vec{u'}_j, \vec{v'}_j)$ and $(\vec{u}_i, \vec{v}_i)$ (where $\vec{u}_i, \vec{v}_i$) is the next vector pair we have a query on) from the uniform average-case parity OuMv we produce many vector pairs $(\vec{u'}_j, \vec{v'}_j)$ and the j-th matrices for each S_k as follows. We first need to determine the three numbers s_u, s_v, and s_M, where s_u, resp. s_v corresponds to the number of flipped bits that will be applied to $(\vec{u'}_{j-1}, \vec{v'}_{j-1})$ to generate $(\vec{u'}_j, \vec{v'}_j)$ and s_M will be used for constructing the three matrices. They are generated as follows: Initially set s_u, s_v, and s_M all to 0. Then $n\lg^2(n)$ times increment one of s_u, s_v, and s_M, each with equal probability. Then perform a check described below. We use a different random procedure to sample the vectors if it succeeds vs fails.

Recall that we are give a (uniform) random sampled $\vec{u}_i$. Additionally, we will maintain the invariant that $\vec{u'}_{j-1}$ is sampled from a distribution that has a TVD of at most $2^{-\Omega(n^2)}$ from uniform vectors (See Lemma C.2. Let b_u be the number of indices where $\vec{u'}_{j-1}$ and $\vec{u}_{i+1}$ differ. Let b_v be defined analogously. We call s_u (resp. s_v) *valid for b_u (resp. b_v)* if (a) $s_u \geq b_u$ and (b) the parity of s_u equals the parity of b_u. With probability $1/4$ s_u and s_v will match the parity constraints and $s_u < b_u$ or $s_v < b_v$ happens with probability at most $2^{-\omega(1)}$, i.e., with probability $1/4 - 2^{-\omega(1)}$ both are valid. We will sample an S and describe slightly different procedures bellow depending on if S is valid or not. Our goal is that individually, each of S_1, S_2, S_3 are series of updates sampled from a distribution with a TVD of zero from the biased updates distribution. Additionally, whenever we sample a valid S we will answer the query of $(\vec{u}_i, \vec{v}_i)$ and we can increment i to solve the next vector pair. When S is invalid we will effectively be solving a random problem unrelated to our query. Note that after sampling s_u and s_v at most $\lg^4(n)$ times they are valid for both b_u and b_v with probability at least $1 - 2^{-\lg^3(n)}$.

In the case where we have selected valid s_u and s_v, we create the actual updates for $\vec{u'}_{j-1}$ and $\vec{v'}_{j-1}$ to create $\vec{u'}_j = \vec{u}_i$ and $\vec{v'}_j = \vec{v}_i$. Consider $\vec{u'}_{j-1}$ and the set of indices that change when going to $\vec{u'}_i$. On each of those indices of $\vec{u}_i$ we will generate an odd number of flips and on all other indices we will generate an even number of flips. We sample a set of updates to the indices of $\vec{u}_i$ conditioned on having s_u and respecting the transformation from $\vec{u}_i$ to $\vec{u}_{i+1}$ as follows: we add one flip for each index that changes between $\vec{u}_i$ and $\vec{u}_{i+1}$, and then we sample $(s_u - b_u)/2$ indices in $[1, n]$ uniformly at random and add two bit flips at the sampled indices. Finally, we randomize the order of the flips in β_u. Applying these bit flips to $\vec{u}_i$ gives the vector $\vec{u'}_{i+1}$. We denote the set of updates for $\vec{u}_i$ by β_u. Then, we perform the same procedure for v to generate β_v creating vector $\vec{v'}_{i+1}$. Finally to create the list of updates for both, we merge the lists β_u and β_v and randomize the order of updates between them. Call this list of updates $\beta_{u,v}$.

In the case where S was invalid (either s_u or s_v was invalid, instead sample s_u uniformly random updates iid and them to β_u. Sample s_v uniformly random updates iid and them to β_v. Note that regardless of what S we sample we use it and fill S with flips.

Now we will analyze the TVD of the flips we perform in β_u to the uniformly sampled flips. We will use $Bin_\mu[x]$

to refer to the Binomial distribution where a success happens with probability μ and there are x samples. First, by Lemma C.2 we have that a vector that results from the sequence of $Bin_{1/3}[n\lg^2(n)]$ bit flips on $\vec{u}_i$ and a uniformly random vector have TVD of at most $2^{-\Omega(\lg^2(n))}$. Note that (1) we can use the linearity of XOR here to start from $\vec{u}_i$ instead of the empty vector and (2) the length of a sequence of $Bin_{1/3}[n\lg^2(n)]$ bit flips on $\vec{u}_i$ has the same distribution as the number s_u picked by one iteration of the above procedure. This is relevant for the case where S is valid. In the case where S is invalid we are sampling flips in a distribution that is indistinguishable from the biased updates distribution we describe. So, the distribution of the vectors $\vec{u'}_{i+1}$ and $\vec{v'}_{i+1}$) that result from the above procedure and the distribution over two uniform random vectors have TVD of at most $3 \cdot 2^{-\Omega(\lg^2(n))} = 2^{-\Omega(\lg^2(n))}$. Additionally, the distribution over updates to achieve these outcomes is has a TVD of at most $2^{-\Omega(\lg^2(n))}$.

Now we need to sample the bit flips in M. We have two cases based on whether s_M is even or odd. If the number of samples is even then sample $s_M/2$ iid uniformly random updates to M three times, call these possible updates Δ_a^i, Δ_b^i, and Δ_c^i. We will additionally define the set F_i to be the empty set of updates for the even case. If s_M is odd then sample $\lfloor s_M/2 \rfloor$ entries for Δ_a^i, Δ_b^i, and Δ_c^i. We additionally sample as single bit flip in the matrix M and let F_i be the set of just this flip in the odd case. We create three sampled flips $f_1 = \Delta_a^i \oplus \Delta_b^i \oplus F_i$, $f_2 = \Delta_a^i \oplus \Delta_c^i \oplus F_i$, and $f_3 = \Delta_b^i \oplus \Delta_c^i \oplus F_i$, to be used by S_1, resp. S_2, resp. S_3, one per instance. Note that no matter whether s_M is even or odd, we produced three instances with $n\lg^2(n)$ uniformly random samples.

Now we show that if we make calls to the three biased updates instances and sum their outputs we will get $\vec{u}_{i+1}M\vec{v}_{i+1}$. Note that the matrix used by the first instance for operation $i+1$ is $M \oplus \bigoplus_{j=1}^{i+1}(\Delta_a^j \oplus \Delta_b^j \oplus F_i)$, for the second it is $M \oplus \bigoplus_{j=1}^{i+1}(\Delta_a^j \oplus \Delta_c^j \oplus F_i)$, and for the third it is $M \oplus \bigoplus_{j=1}^{i+1}(\Delta_b^j \oplus \Delta_c^j \oplus F_i)$. Thus,

$$\vec{u}_{i+1}(M \oplus \bigoplus_{j=1}^{i+1}(\Delta_a^j \oplus \Delta_b^j \oplus F_i))\vec{v}_{i+1} \oplus \vec{u}_{i+1}(M \oplus \bigoplus_{j=1}^{i+1}(\Delta_b^j \oplus \Delta_c^j \oplus F_i))\vec{v}_{i+1} \oplus \vec{u}_{i+1}(M \oplus \bigoplus_{j=1}^{i+1}(\Delta_c^j \oplus \Delta_a^j \oplus F_i))\vec{v}_{i+1} =$$

$$3\vec{u}_{i+1}M\vec{v}_{i+1} \oplus 2(\vec{u}_{i+1}\bigoplus_{j=1}^{i+1}(\Delta_a^j \oplus \Delta_b^j \oplus \Delta_c^j)\vec{v}_{i+1}) \oplus 3\vec{u}_{i+1}(\bigoplus_{j=1}^{i+1}F_i)\vec{v}_{i+1} =$$

$$(\vec{u}_{i+1}M\vec{v}_{i+1} \oplus \vec{u}_{i+1}(\bigoplus_{j=1}^{i+1}F_i)\vec{v}_{i+1}) \mod 2.$$

We will save the values of all sampled F_i, these have between zero and one entries each and each such entry consist of a 0 or a 1 and a location in M. Computing $c = \vec{u}_{i+1}(\bigoplus_{j=1}^{i+1}F_i)\vec{v}_{i+1}$ takes $O(i)$ time in the following way. We initialize c to be zero at the beginning of the reduction and whenever the j-th vector pair of the uniform average-case parity OuMv instance arrives and we have determined the new F_j then we perform the following operation: If F_j is non-empty and has location $M[a][b]$ then we XOR our current value of c with $\vec{u}_{i+1}[a]\vec{v}_{i+1}[b]$. We can then return the value $\vec{u}_{i+1}M\vec{v}_{i+1} \mod 2$ by XORing c to the value $(\vec{u}_{i+1}M\vec{v}_{i+1} \oplus \vec{u}_{i+1}(\bigoplus_{j=1}^{i+1}F_i)\vec{v}_{i+1})$ received by XORing the query answers of S_1, S_2, and S_3. Computing c for all n OuMv queries requires $O(n^2)$ time across all n pairs $(\vec{u}_{i+1}, \vec{v}_{i+1})$. This shows the correctness of our reduction.

Lemma C.1 shows that if two distributions have TVD ε then an algorithm that succeeds on one distribution with probability p must succeed on the other with probability at least $p - \varepsilon$. We will use this to overcome the distance of our problem from the TVD of at most $2^{-\Omega(\lg^2(n))}$ from the uniform distribution. Assume we have an algorithm for biased average-case parity OuMv with failure probability at most $1/60$. If we apply it to one of the instances S_k, it has failure probability at most $1/60 + 2^{-\Omega(\lg^2(n))}$. Thus, the total failure probability of the algorithm when applied to S_1, S_2, and S_3 is at most $3/60 + 3 \cdot 2^{-\Omega(\lg^2(n))}$, which is a smaller probability of failure than $1/3$. Thus, the algorithm could be used to answer the given (uniform) average-case parity OuMv problem with probability at least 2/3. Our reduction uses $O(n^2 \log^n) = O(n^{2+o(1)})$ updates and n queries. Combined with Theorem 2.1 the result follows.

We restate the above proof and give the proof for biased average-case parity OMv in Appendix A. □

It follows that any dynamic algorithm for biased average-case parity OuMv or OMv with update time $n^{1-\varepsilon}$ and query time $n^{2-\varepsilon}$ for any small $\varepsilon > 0$ would contradict the worst-case OMv conjecture.

3 Dynamic Inclusion-Edgesclusion

We extend the idea of inclusion-edgesclusion from [9], which uses inclusion-exclusion based techniques on the edge sets of graphs. This allows us to use an algorithm for counting unlabeled subgraphs to count labeled subgraphs. In [9] they used the technique to move from a k-partite Erdős-Rényi graph into a fully connected Erdős-Rényi graph. We show here how to achieve this in the dynamic setting. This is not straightforward because the version presented in [9] has a step which takes $O(m)$ time, which is not acceptable in the dynamic setting. The original reduction uses recursion over subgraphs of H down to trees and uses a $O(m)$ step on these trees. We will avoid this issue by making our base case a two node graph with a single edge between them. We can track the number of edges between two partitions efficiently with a dynamic algorithm. All of the recursive calls in our reduction can be made by having several copies of dynamic algorithms with different (still random looking) input graphs. The final change we make is that we allow the subgraph H to contain fixed points, which enables us to count *st* paths, *s* triangles, etc (see definition 3.1). Let us start first with some definitions.

DEFINITION 3.1. *Given a graph $G=(V,E)$, let H be a subgraph of G with k nodes and e edges. We call H* labeled *if we label the nodes as $v_1,\ldots,v_k$. We say H has* fixed nodes *if we take a given labeled vertex v_i and identify it with a particular node $s\in V$ in G. (For example if H is an st three path with nodes $v_1-v_2-v_3-v_4$ we are identifying $v_1\in H$ with $s\in G$ and $v_4\in H$ with $t\in G$. To notate an st three path as H_S then set S is $\{(v_1,s),(v_4,t)\}$. Note that the unfixed nodes v_2 and v_3 don't appear in S.) We denote a subgraph H with fixed nodes $\{v_{i_1},\ldots,v_{i_f}\}\subset\{v_1,\ldots,v_k\}$ where we fix $v_{i_j}=s_j$ by $H_{\{(v_{i_1},s_1),\ldots,(v_{i_f},s_f)\}}$ or as H_S where $S=\{(v_{i_1},s_1),\ldots,(v_{i_f},s_f)\}$, is called the* set of identified nodes *of H.*

DEFINITION 3.2. *We say a graph G is H_S-*partite *if we can partition the vertices of G into k sets $V_1,\ldots V_k$ such that: (1) For all $(v_{i_j},s_j)\in S$ the set $V_{i_j}=\{s_j\}$, i.e., the partitions that correspond to fixed vertices consist of just that vertex alone. (2) If there is no edge between v_i and v_j in H_S then there are no edges between nodes a and b whenever $a\in V_i$ and $b\in V_j$. (3) There are no edges between a and b whenever $a\in V_i$ and $b\in V_i$.*

We call a graph G a labeled Erdős-Rényi H_S-partite graph *if given the labeling of G and H_S all permitted edges have a $1/2$ chance of existing.*

We call a set of updates to a graph G dynamic H_S-partite Erdős-Rényi updates *given the labeling of G and H_S and the set of updates are sampled from permitted edges. We call updates* dynamic Erdős-Rényi updates *if they sample their flips from all available edges (no restrictions).*

For our lower bounds, the graph is initialized as a H_S-partite Erdős-Rényi graph. For our upper bounds the initial graph can either be empty or an Erdős-Rényi graph. Finally, we will define the counting problems for H_S.

DEFINITION 3.3. *The H_S* counting problem *in a graph G returns the numbers of tuples of vertices $(u_1,\ldots,u_k)$ such that $u_{i_j}=s_j$ for all $(v_{i_j},s_j)\in S$ and such that if $(v_a,v_b)\in H$ then (u_a,u_b) is in the edge set for G. The* average-case H_S counting problem *is the same problem but where G is drawn from the distribution of Erdős-Rényi graphs. The* dynamic average-case H_S counting problem *is a H_S counting problem where the updates for G are dynamic Erdős-Rényi updates. The graph is initialized by drawing a random Erdős-Rényi graph.*

The unlabeled counting H_S problem counts the number of copies of H_S that exist in G where all identified nodes $(v_{i_j},s_j)\in S$ are in fact matched. However, all nodes that are not identified need not have their label in H_S match the partition they are a part of.

We define H_S and H_S counting the way we do to encapsulate, for example, *st*-5-path counting. We want to capture the notion of counting *st*-5-paths when they move through the partitions in our 'desired' order with labeled H_S counting. We want to capture the notion of counting *st*-5-paths with only the restriction of starting at s and ending at t with unlabeled H_S counting.

We will show the following: Assuming that counting labeled copies of H_S in a dynamic H_S-partite Erdős-Rényi graph is hard, then counting the same subgraph H_S in dynamic Erdős-Rényi graphs is also hard. So, we allow ourselves to add 'fake' edges to turn a dynamic H_S-partite Erdős-Rényi graph into a dynamic Erdős-Rényi graphs and use the linearity of XOR to return the value we actually care about. Note that it can be difficult to prove that counting labeled subgraphs H_S is hard, even in dynamic H_S-partite Erdős-Rényi graphs. We prove this result for various subgraphs in sections 4, 5, and 6 using the biased updates theorem for OMv and OuMv.

Our result generalizes the previous work in two ways. First, we make the technique work dynamically. This primarily involves a change to the base case and some changes to the recursive argument. Secondarily, we allow fixed

labeled points. Notably, this allows us to use this inclusion-edgesclusion argument for things like counting *st* paths of length 5.

Small Illustrative Example Assume we want to count *st* paths of length 3 in a graph $G = (V,E)$ with $V = A \cup B \cup \{s,t\}$ with $A \cap B = \emptyset$ and $|A| > 0$ and $|B| > 0$ and E consisting of randomly sampled edges between E_{sA}, E_{AB}, and E_{Bt} (see Figure 1). Note that G doesn't look like an Erdős-Rényi graph. It is missing edges in E_{sB} and E_{tA} and also within the partition A and within the partition B. Imagine adding to G random 'bad' edges of this sort. We will now use the notation of *xyzw* to indicate the four vertices in order that form a fourpath. For example *saat* would refer to a path from s to some node in A to some node in A to the node t. We still want to count all paths of the form *sabt*, however, a naive count will also include: *saat*, *sbbt*, and *sbat*. The inclusion-exclusion-based techniques that exist in the literature allow us to count just the paths that use exactly one vertex from each partition, so we can count all *sabt* and *sbat* paths. However, the question remains how to count c, the number of *sabt* paths.

Our idea is as follows: Create a random edge sets E_{sB} and E_{At} with edges between E_{At} and E_{sB}, as well as $\bar{E}_{sB} = \{(s,b) | (s,b) \notin E_{sB}\}$ and $\bar{E}_{tA} = \{(t,a) | (t,a) \notin E_{tA}\}$. Now consider graphs $G_1 = E \cup E_{sB} \cup E_{At}$, $G_2 = E \cup \bar{E}_{sB} \cup E_{At}$, $G_3 = E \cup E_{sB} \cup \bar{E}_{At}$, and $G_4 = E \cup \bar{E}_{sB} \cup \bar{E}_{At}$ (see Figure 1). We use $\#G_i$ to denote the number of length-3 *st* paths that exist in graph G_i. The crucial observation is that, by construction, any path *sbat* for which $(b,a) \in E$ exists in *exactly one* of the graphs G_i, while every path *sabt* (that we want to count) appears in each graph G_i. It follows that the sum $\#G_1 + \#G_2 + \#G_3 + \#G_4 = 4c + |E_{AB}|$. It is easy (statically or dynamically) to count the number of edges that exist between A and B ($|AB|$). So, with 4 calls to counting all *st* paths of length 3 in the (correlated) Erdős-Rényi graphs $G_1, \ldots G_4$ we can compute c, the number of *sabt* paths in the graph G.

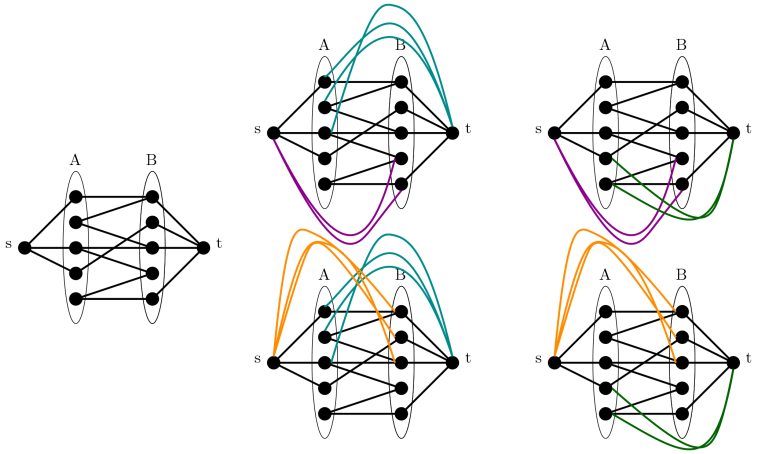

Figure 1: We have a graph with edges in E_{sA}, E_{AB}, and E_{Bt} drawn in black. Then we have four graphs G_1, G_2, G_3, and G_4 with edges between E_{sB} and E_{tA} added. We show in orange and purple two flipped edge sets for E_{sB} and in cyan and green two edge sets flipped between $E_{At}S$.

Dynamic Inclusion-Edgesclusion We will show how to generalize the idea presented in the above example in Appendix B. The core concept is flipping each 'bad' edge set. With larger subgraphs H there may be larger subraphs that remain, however, we can count those as well using a recursive algorithm. We will now state the dynamic inclusion-edgesclusion theorem (proof in Appendix B).

THEOREM 3.1. *Imagine we are given a dynamically updated graph with random updates, where some edges are marked as not-allowed. The graph will be split into k partitions and edges with 'disallowed' updates must be the complete edge sets between two partitions. These edges will not be randomly updated by the random updates. We use this structure to make our proofs easier in the body of the paper.*

Let H_S be a graph with k nodes. Assume that the problem of counting H_S in a graph with random updates among the allowed edges in an H_S-partite graph requires $2^{\binom{k}{2}+k}U(n)$ time per update and $2^{\binom{k}{2}+k}Q(n)$ time per query as long as at most $O(2^{\binom{k}{2}+k}(n^2 + P(n)))$ time is spent preprocessing and gives correct answers to queries with probability

labeled points. Notably, this allows us to use this inclusion-edgesclusion argument for things like counting *st* paths of length 5.

Small Illustrative Example Assume we want to count *st* paths of length 3 in a graph $G = (V,E)$ with $V = A \cup B \cup \{s,t\}$ with $A \cap B = \emptyset$ and $|A| > 0$ and $|B| > 0$ and E consisting of randomly sampled edges between E_{sA}, E_{AB}, and E_{Bt} (see Figure 1). Note that G doesn't look like an Erdős-Rényi graph. It is missing edges in E_{sB} and E_{tA} and also within the partition A and within the partition B. Imagine adding to G random 'bad' edges of this sort. We will now use the notation of *xyzw* to indicate the four vertices in order that form a fourpath. For example *saat* would refer to a path from s to some node in A to some node in A to the node t. We still want to count all paths of the form *sabt*, however, a naive count will also include: *saat*, *sbbt*, and *sbat*. The inclusion-exclusion-based techniques that exist in the literature allow us to count just the paths that use exactly one vertex from each partition, so we can count all *sabt* and *sbat* paths. However, the question remains how to count c, the number of *sabt* paths.

Our idea is as follows: Create a random edge sets E_{sB} and E_{At} with edges between E_{At} and E_{sB}, as well as $\bar{E}_{sB} = \{(s,b) | (s,b) \notin E_{sB}\}$ and $\bar{E}_{tA} = \{(t,a) | (t,a) \notin E_{tA}\}$. Now consider graphs $G_1 = E \cup E_{sB} \cup E_{At}$, $G_2 = E \cup \bar{E}_{sB} \cup E_{At}$, $G_3 = E \cup E_{sB} \cup \bar{E}_{At}$, and $G_4 = E \cup \bar{E}_{sB} \cup \bar{E}_{At}$ (see Figure 1). We use $\#G_i$ to denote the number of length-3 *st* paths that exist in graph G_i. The crucial observation is that, by construction, any path *sbat* for which $(b,a) \in E$ exists in *exactly one* of the graphs G_i, while every path *sabt* (that we want to count) appears in each graph G_i. It follows that the sum $\#G_1 + \#G_2 + \#G_3 + \#G_4 = 4c + |E_{AB}|$. It is easy (statically or dynamically) to count the number of edges that exist between A and B ($|AB|$). So, with 4 calls to counting all *st* paths of length 3 in the (correlated) Erdős-Rényi graphs $G_1, \ldots G_4$ we can compute c, the number of *sabt* paths in the graph G.

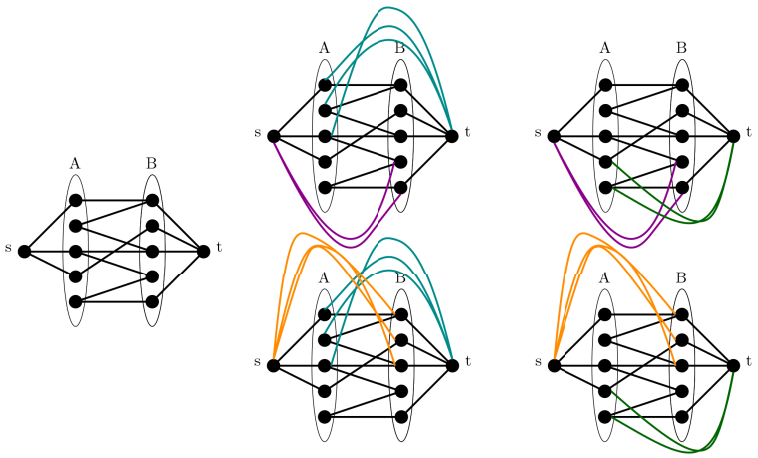

Figure 1: We have a graph with edges in E_{sA}, E_{AB}, and E_{Bt} drawn in black. Then we have four graphs G_1, G_2, G_3, and G_4 with edges between E_{sB} and E_{tA} added. We show in orange and purple two flipped edge sets for E_{sB} and in cyan and green two edge sets flipped between $E_{At}S$.

Dynamic Inclusion-Edgesclusion We will show how to generalize the idea presented in the above example in Appendix B. The core concept is flipping each 'bad' edge set. With larger subgraphs H there may be larger subraphs that remain, however, we can count those as well using a recursive algorithm. We will now state the dynamic inclusion-edgesclusion theorem (proof in Appendix B).

THEOREM 3.1. *Imagine we are given a dynamically updated graph with random updates, where some edges are marked as not-allowed. The graph will be split into k partitions and edges with 'disallowed' updates must be the complete edge sets between two partitions. These edges will not be randomly updated by the random updates. We use this structure to make our proofs easier in the body of the paper.*

Let H_S be a graph with k nodes. Assume that the problem of counting H_S in a graph with random updates among the allowed edges in an H_S-partite graph requires $2^{\binom{k}{2}+k}U(n)$ time per update and $2^{\binom{k}{2}+k}Q(n)$ time per query as long as at most $O(2^{\binom{k}{2}+k}(n^2 + P(n)))$ time is spent preprocessing and gives correct answers to queries with probability

$1-\delta$. *If the allowed edges are a constant fraction of all edges then the average-case H_S counting problem requires at least $\Omega(U(n))$ time per update or at least $\Omega(Q(n))$ time per query as long as at most $O(P(n))$ time is spent preprocessing. This new algorithm will give correct answers to queries with probability at least* $1-\delta 2^{\binom{k}{2}+k}$.

4 Lower Bounds for Counting 4-Cycles

In this section we will get a lower bound for the update time, $U(n)$ and query time, $Q(n)$ of average-case four cycle counting from biased average-case parity OuMv. These lower bounds will apply for any polynomial preprocessing. More specifically, we show that $nU(n)+Q(n)=\Omega(n^{2-o(1)})$ if the OMv hypothesis holds. Before we show that counting 4-cycles is hard in general graphs where all updates are flips of random edges, we will show that counting 4-cycles is hard on special 4-partite graphs where the edges between two of the parts are complete and all other edges are set by random flips. Given a sequence of $(M_i,\vec{u}_i,\vec{v}_i)$ for $0\leq i\leq n$ of a biased average-case parity OuMv problem with dimension n we construct a graph with $4n$ nodes, n each in A, B, C, and D, and edges $E\subseteq A\times B\ \cup\ B\times C\ \cup\ C\times D\ \cup\ D\times A$. The edges between A and B represent M in the natural way (edge (a_i,b_j) exists iff $M[i][j]=1$), the edges between D and A (resp. B and C) represent $\vec{u}$ (resp. $\vec{v}$) as described next, and there is a complete graph between C and D.

To represent $\vec{u}$ we use the $O(n^2)$ edges between A and D as follows: For any fixed j with $1\leq j\leq n$, let the parity of the number of edges (d_i,a_j) that exist for $1\leq i\leq n$ be equal to $\vec{u}[j]$. The edges in this graph are, of course, randomly updated. Additionally, they start as random. Note that when the edges start as random the parity of the number of (d_i,a_j) edges that exist is odd or even each with probability $1/2$. Further note that if we randomly update an edge between D and A this corresponds to randomly flipping a bit in $\vec{u}$. So, we are representing $\vec{u}$ as an XOR over n^2 edges, $\vec{v}$ is represented in the same way by representing $\vec{v}[j]$ by edges between B and C. We are using an XOR because we want random flips of edges to flip bits in our vector. See Figure 2 for a visual depiction.

For $1\leq i\leq n$ let the vector $\vec{p}_j$ be the vector such that $\vec{p}_j[i]=1$ iff the edge (d_j,a_i) exists in the graph (where d_j is the j^{th} node in D and a_i is defined similarly) and zero otherwise. Define $\vec{q}_j$ similarly as the vector s.t. $\vec{q}_j[i]=1$ iff (c_j,b_i) exists and zero otherwise. Note that we will set these edges such that $\vec{u}=\bigoplus_{j\in[1,n]}\vec{p}_j$ and let $\vec{v}=\bigoplus_{j\in[1,n]}\vec{q}_j$.

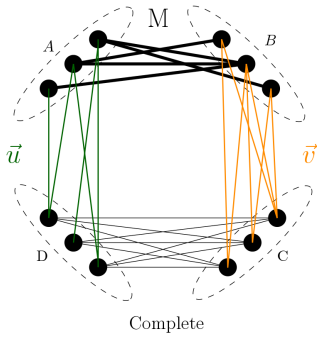

Figure 2: In this graph we depict a 4-partite graph constructed from an OuMv instance.

First let us explain why blowing up the representations of $\vec{u}$ and $\vec{v}$ are necessary. We need n^2 edges to represent M, and the most efficient representation of $\vec{u}$ and $\vec{v}$ could involve only n edges. However, if we make $O(n\lg^2(n))$ updates the expected number of updates that will flip edges representing $\vec{u}$ and $\vec{v}$ in their efficient representation is only $O(\lg^2(n))$ bit flips as the chance of picking an edge representing them is only $\Theta(1/n)$. This is not enough to randomize our vectors. However, if we represent the vectors $\vec{u}$ and $\vec{v}$ as the XOR over n representations of the vectors (e.g. the n vectors $\vec{p}_j$) we expect to make $O(n\lg^2(n))$ random flips in the vectors $\vec{u}$ and $\vec{v}$. Of course, we also expect to flip some bits in M when doing this. So, to prove that 4-cycle counting in our 4-cycle graph is hard when three of the parts are randomly updated we need to rely on Theorem 2.2. This biased model of edge flips captures our situation.

We explain next why the edge set between C and D is complete. Let $\vec{U}=\sum_{j\in[1,n]}\vec{p}_j$ and let $\vec{V}=\sum_{j\in[1,n]}\vec{q}_j$ and note that for each $1\leq i\leq n$, $\vec{u}[i]=\vec{U}[i] \mod 2$ and $\vec{v}[i]=\vec{V}[i] \mod 2$. Let $E_{C,D}$ be the edge set of C,D. For convenience let $(i,j)\in E_{C,D}$ mean there is an edge between c_i and d_j. We want to compute $\vec{u}M\vec{v}$ but when counting the number of

4-cycles that have one node from each partition we actually count

$$\sum_{(i,j)\in E_{C,D}} (\vec{p}_j M \vec{q}_i).$$

If $E_{C,D}$ is complete then this sum simplifies to $\vec{u}M\vec{v} \mod 2$. Now we will formally prove that in this restricted setting, the four cycle counting problem is hard. We will later show how to remove this restriction by simulating this setting through mutiple calls to actual Erdős-Rényi graphs. That will complete the proof.

LEMMA 4.1. *We are given a graph with 4 sets of nodes with n nodes each in A,B,C,D. We allow edges between AB, BC, CD, and DA. We enforce that CD is a complete bipartite graph. Let H be a 3-path and consider dynamic H-partite Erdős-Rényi updates where $D-A-B-C$ are the parts that are associated with the edge updates. Every update is therefore selected uniformly at random from $D\times A \cup A\times B \cup B\times C$.*

Consider a dynamic algorithm $\mathscr{A}$ that correctly counts the number of 4-cycles with exactly one node from each of the parts A,B,C,D with probability $59/60$ per query. Each update is a random update as described above. If $\mathscr{A}$ has preprocessing time $P(n) = n^{3-\varepsilon}$ for some $\varepsilon > 0$ then if the OuMv hypothesis (or OMv) holds we have that $n^2U(n) + nQ(n) = \Omega(n^{3-o(1)})$.

Proof. We sample an instance of biased average-case OuMv. Let $M_0, \vec{v}_0, \vec{u}_0$ be the uniformly random vectors at the start. Let β^i be the sequence of all biased updates in the i^{th} set of $n\lg^2(n)$ biased random updates. Let β^i_M, β^i_u, and β^i_v be the subsets of β^i that flip bits in M_i, $\vec{u}_i$, and $\vec{v}_i$ respectively.

For convenience of notation let a_j be the j^{th} node in the set A. Similarly define b_j, c_j, and d_j. For initializing the graph we sample M_0 and add the edge (a_j, b_k) iff $M_0[j][k] = 1$. Note that the edges in $A\times B$ will now look Erdős-Rényi (each edge exists iid with probability $1/2$). Now for $D\times A$ we start by sampling $n-1$ random n bit Boolean vectors $\vec{p}_j$. Then iff $\vec{p}_j[k] = 1$ we add the edge (d_j, a_k). We set the last vector $\vec{p}_n = \vec{u}_0 \oplus \bigoplus_{j=1}^{n-1} \vec{p}_j$. Note that this means $\vec{u}_0 = \bigoplus_{j=1}^{n} \vec{p}_j$. Further note that because $\vec{u}_0$ is a uniformly randomly sampled Boolean vector, the bits of $\vec{p}_n$ are sampled iid 1/0 with probability $1/2$ conditioned on all $\vec{p}_j$ $j < n$. Now iff $\vec{p}_n[k] = 1$ we add the edge (d_n, a_k). For $B\times C$ we do a similar procedure where we sample $n-1$ random vectors $\vec{q}_j$ and then sample $\vec{q}_n = \vec{v}_0 \oplus \bigoplus_{j=1}^{n-1} \vec{q}_j$. Then iff $\vec{q}_j[k] = 1$ we add the edge (c_j, b_k). Now note that all edges in $A\times B$ and $B\times C$ are included or excluded iid with probability $1/2$. So $D\times A \cup A\times B \cup B\times D$ is a three-path-partite Erdős-Rényi graph.

Now we will explain how to make an update in the graph given an update from each of β^i_M, β^i_u, and β^i_v to generate $n\lg^2(n)$ updates in our graph. For all bit flip updates $M[j][k] \in \beta^i_M$ we flip the edge (a_j, b_k). For all bit flip updates $\vec{u}[k] \in \beta_u$ we sample a uniformly random $j \in [1,n]$ and flip the edge (d_j, a_k). For all bit flip updates $\vec{v}[k] \in \beta_v$ we sample a uniformly random $j \in [1,n]$ and flip the edge (c_j, b_k). Every edge in $A\times B$ (similarly in $D\times A$ and in $B\times C$) has a probability of being flipped of $(\frac{1}{3n^2})$. All updates in biased updates are independent. We take each update in β^i in order and apply the appropriate procedure (depending on if it is from β^i_M, β^i_u, or β^i_v). So we are sampling $n\lg^2(n)$ updates iid across all allowed edges uniformly at random as promised.

Now we explain why counting four cycles in the produced graph solves OuMv on the associated vectors and matrices. Note that the number of four cycles that exist for every edge (d_j, c_ℓ) is equal to $\vec{p}_j M \vec{q}_\ell$. Now consider the sum over all (d_j, c_ℓ):

$$\sum_{j\in[0,n-1]} \left(\sum_{\ell\in[0,n-1]} \vec{p}_j M \vec{q}_\ell \right) = \left(\sum_{j\in[0,n-1]} \vec{p}_j \right) M \left(\sum_{\ell\in[0,n-1]} \vec{q}_\ell \right) = \vec{U} M \vec{V}$$

Note further that $\vec{U}M\vec{V} \mod 2 = \vec{u}M\vec{v} \mod 2$.

So, counting all the *ABCD* four cycles after $n\lg^2(n)$ updates and returning their parity gives the answer for $\vec{(u)}_i M_i \vec{(v)}_i$ for each $1 \le i \le n$, solving the biased average-case parity OuMv problem with a total number of $O(n^2\log^2(n)$ updates and $O(n)$ queries. Let $U(n)$ be the time for updates and $Q(n)$ be the time for queries. Then we have that $n^2\lg^2(n)U(n) + nQ(n) = \Omega(n^{3-o(1)})$ by Theorem 2.2. This shows hardness from OMv as well by Theorem 2.7 from [13] which states that the OMv hypothesis implies the OuMv hypothesis. ∎

We now use our dynamic inclusion-edgeclusion principle to establish the hardness.

THEOREM 4.1. *In the average-case dynamic model counting 4-cycles requires $n^{1-o(1)}$ time per update or $n^{2-o(1)}$ time per query if the pre-processing time is at most $n^{3-\varepsilon}$ for some $\varepsilon > 0$ if the OMv hypothesis is true.*

Proof. From Lemma 4.1 we know that this counting problem with probability $59/60$ requires $n^2U(n) + nQ(n) = \Omega(n^{3-o(1)})$ time for updates ($U(n)$) and queries ($Q(n)$) if edge sets DA, AB, and BC are updated dynamically at random, CD is complete, and all other edge sets are empty. Call this graph G.

Now consider picking a random set of edges in $D \times C$ of size $1/2$, call this edge set F and denote the set of remaining edges in $D \times C$ by $\bar{F}$. When a dynamic update is made to the edges between D and C flip the edges in both F and $\bar{F}$, so we maintain the property that every edge (c,d) appears in exactly one of F and $\bar{F}$. Further note that both edge sets are indistinguishable from random. Let G' be the graph with edge sets DA, AB, BC and F, whereas G'' is the graph with edge sets DA, AB, BC and $\bar{F}$. We will run the dynamic algorithm on both G' and G'' and sum their answers for the number of $ABCD$ 4-cycles. The total number of $ABCD$ 4-cycles between both graphs G' and G'' is equal to the number of $ABCD$ 4-cycles in G. So, given a graph with random dynamic updates on AB, BC, CD, and DA it is hard to count $ABCD$ 4-cycles with probability $119/120$. It requires $n^2U(n) + nQ(n) = \Omega(n^{3-o(1)})$ time for updates and queries from biased updates parity OuMv. Note that biased updates parity OuMv is hard from the OMv hypothesis.

So we have shown that counting labeled 4-cycles is hard in a dynamically updated 4-cycle-partite Erdős-Rényi dynamic graphs from OMv. Then, by Theorem 3.1 we can then say that counting 4-cycles in graphs with fully random dynamic updates (so no restrictions to specifics edge sets) requires $n^2U(n) + nQ(n) = \Omega(n^{3-o(1)})$ for updates and queries if the success probability is at least $719/720$. □

5 Lower Bounds for Counting Triangles

We will present lower bounds for counting triangles in this section. We will present counting triangles generically from worst-case offline triangle counting. Then, in this section we will give a lower bound for counting triangles through a *queried point* in average-case dynamic graphs. Note that it is different from a fixed point s (we present a fast algorithm for s-triangle counting in section 10). We get this lower bound from OMv.

5.1 Lower Bound from Worst-Case Offline Triangle Counting

First we will present the lower bound for counting triangles in Erdős-Rényi graphs. We get this lower bound from the worst-case offline 3-clique hypothesis.

DEFINITION 5.1. *The 3-clique hypothesis states that detecting a 3-clique in a worst-case graph requires $n^{\omega-o(1)}$ time where ω is the matrix multiplication constant.*

Now we can use this to say that counting triangles requires super constant time if $\omega > 2$.

THEOREM 5.1. *Let $\mathscr{A}$ be a dynamic algorithm which solves average-case triangle counting in Erdős-Rényi graphs with probability $1-2^{-10}$ with $P(n)$ preprocessing time, $Q(n)$ query time , and $U(n)$ update time. Then if the 3-clique hypothesis holds we have that*

$$P(n) + Q(n) + n^{2+o(1)}U(n) = \Omega(n^{\omega-o(1)}).$$

Therefore, if $P(n) + Q(n) = n^{\omega-o(1)-\varepsilon}$ for some $\varepsilon > 0$ we have that $U(n)$ is at least $n^{\omega-2-o(1)}$.

Proof. From Theorem 2 in [11] counting 3-cliques mod 2 with probability $1-2^{-9}$ in time $T(n)$ we can count 3-cliques in a worst-case graph with probability at least $2/3$ in time $O(T(n))$[1].

Given an Erdős-Rényi graph we will produce a series of random updates. Given a starting Erdős-Rényi graph G and an ending Erdős-Rényi G' we will make a matrix of differences D. We set $D[u][v] = 0$ if (u,v) either exists in both G and G' or doesn't exist in both. We set $D[u][v] = 1$ if (u,v) exists in one of G or G' but not both. Let G be the initial graph for our dynamic algorithm. Let G' be the static Erdős-Rényi graph we have sampled. We will now describe a procedure to generate random updates that produce G'. We will first determine how many updates we will make to each edge, then we will randomize the order of these updates. We will save the number of updates in a matrix Δ. We will have an odd number of updates on each edge (u,v) where $D[u][v] = 1$ and an even number of updates when $D[u][v] = 0$. We initialize $\Delta = D$. Let c_D be the number of ones in D. If we want xn updates we sample a uniformly random edge (u,v) $(xn - c_D)/2$ times and then increment $\Delta[u][v]$ by two. Now we make a sequence of updates by

[1] Note that a similar result is proved in [5] with a slightly different probability.

randomizing the order of the updates represented in Δ. Further note that conditioned on the graph going from G to G' these sampled updates are drawn from uniform distribution.

Now consider starting from any graph and making xn^2 random updates to random edges. The resulting graph has a TVD from a Erdős-Rényi graph of at most $n^2(1-1/n^2)^{xn^2} < n^2e^{-x}$. So our sampling procedure above has TVD at most n^2e^{-x}.

Consider running the algorithm $\mathscr{A}$ with xn^2 random updates and one query, if $\mathscr{A}$ succeeds with probability p. This implies a success probability of $1-p-n^2e^{-x}$ for counting cliques in a Erdős-Rényi graph. Consider setting $x = \lg^2(n)$. Now, if $p = 2^{-10}$ we have an algorithm for counting 3-cliques in an Erdős-Rényi graph that succeeds with probability greater than $1-2^{-9}$, which implies a worst-case algorithm with a success probability of at least $2/3$. This algorithm involves the prepossessing of $\mathscr{A}$, $\lg^2(n)n^2$ updates, and one query. So, we have that given an algorithm $\mathscr{A}$ we have a worst-case clique counting algorithm which runs in time $P(n)+Q(n)+n^{2+o(1)}U(n)$. By the 3-clique hypothesis

$$P(n)+Q(n)+n^{2+o(1)}U(n) = \Omega(n^{\omega-o(1)}).$$

This gives us our desired result. □

5.2 Lower Bound for Counting Triangles Through a Queried Point Let $\#\triangle_a$ be the count of the number of triangles that go through the node a. Recall that in our model of average-case algorithms our adversary is allowed to pick when updates and queries are made, but not what those queries or updates are. Recall that random update flips a random edge. Additionally, for this problem of counting triangles through a queried point our queries are of the form $\#\triangle_a$ for a randomly selected node a. We will show that with $n^{2+o(1)}$ updates and $n^{2+o(1)}$ queries we can answer n vector queries on a Mv instance with a n by n matrix and n vectors of length n. This will let us prove that for the counting triangles through a queried point problem we require that $U(n)+Q(n)$ takes at least $n^{1-o(1)}$ time.

In this section we will use the OMv hypothesis. We will create a tripartite graph with node sets A, B, and C. We will represent M as the edges between A and B. Specifically $(a_i,b_j) \in E$ if $M[i][j] = 1$. We will represent $\vec{v}$ as the edges between B and U. Specifically if $\vec{v}[j] = 1$ then an odd number of edges (b_j,u_k) exist and if $\vec{v}[j] = 0$ then an even number of edges (b_j,u_k) exist. We will make two calls to graphs with opposite edge sets for A and U so that we can extract a count as if the edge set was complete. See figure 3 for a visual of this reduction. Note that with this setup the number of triangles (in both graphs) through a_i is equal to $(M\vec{v})[i]$. Now, we aren't allowed to query any specific point. However, after $\Theta(n\lg(n))$ queries for the number of triangles through a point we will get answers for all points with constant probability. So, with $\Theta(n\lg^3(n))$ queries we will get answers for all points with high probability. With the answers though a_i for all $i \in [1,n]$ we can reproduce the whole vector $M\vec{v}$.

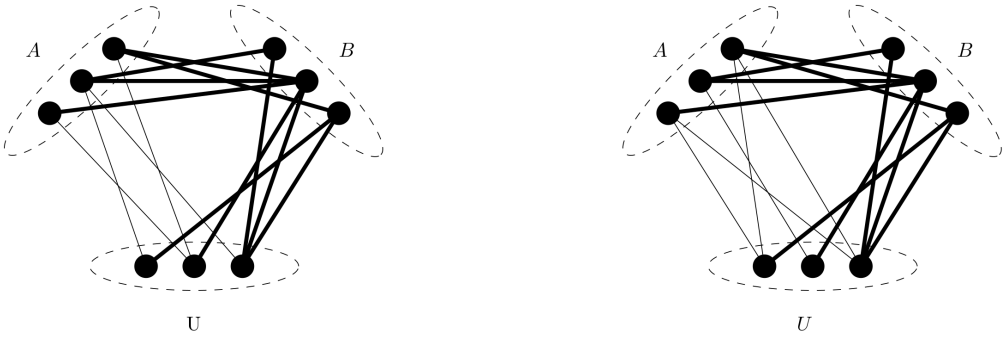

Figure 3: Two graphs with the edge sets flipped between A and U. If we count triangles in both graphs we get a count equal to the count in a single graph where all edges between A and U exist.

Before doing any of that, we will need to demonstrate that $n^2\lg^3(n)$ random updates to a graph produce a graph where each edge appears flipped with probability $1/2$ up to a very small total variation distance.

LEMMA 5.1. *We are given a graph G with n nodes where you allow some subset E of edges where one is picked uniformly at random across all edges to be flipped $n^2\lg^2(n)$ times. The resulting graph G has TVD at most $2^{-\Theta(\lg^2(n))}$*

from a graph where all the edges E exist or don't iid with probability $1/2$.

Proof. Note $|E| \leq n^2$. Consider any given edge $(u,v) \in E$ what is the chance that it is 'on' vs 'off'? Well we flip it with probability $p = 1/|E|$ each time we do a random edge flip. Let q_i be the probability (u,v) is included after i flips. Without loss of generality let $q_0 = 0$. Now define $q_i = 1/2 + \delta_i$. Now note that

$$q_{i+1} = q_i(1-p) + (1-q_i)p = (1/2+\delta_i)(1-p) + (1/2-\delta_i)p = 1/2 + (1-2p)\delta_i = 1/2 + \delta_{i+1}.$$

So $\delta_i = (1-2p)^i/2$. When $i = n^2 \lg^2(n)$ we have that $(1-p)^i/2 \leq e^{-\lg^2(n)}$. We can bound the TVD via union bound over the $|E|$ edges to get at most $n^2 e^{-\lg^2(n)}$ TVD from the uniformly random graph where every edge is included or excluded with probability $1/2$ iid. □

Now we will use this lemma and our average-case OMv results to show that triangle counting is hard.

LEMMA 5.2. *Let H be a triangle. We are given an H-partite Erdős-Rényi graph, G with three partitions of vertices A, B, and U each with n nodes. The updates to G are dynamic H-partite Erdős-Rényi. A random query is made to a random vertex x uniform across all vertices* $\#\triangle_x$.

Let $U(n)$ be the time per update and $Q(n)$ be the time per query. If at most $n^{3-\varepsilon}$ time is spent pre-processing for $\varepsilon > 0$ and if the algorithm answers correctly with probability at least $1 - \frac{1}{n^2 240}$ then $n^2 U(n) + n^2 Q(n) = \tilde{\Omega}(n^3)$.

Proof. We are given an instance of biased average-case parity OMv. Let M_0 and $\vec{v}_0$ be the initial matrix and vector. Let β^i be the $n\lg^2(n)$ updates to go from M_{i-1} and $\vec{v}_{i-1}$ to M_i and $\vec{v}_i$. Let β^i_M be the subset of the updates in β_i that update M_{i-1}. Let β^i_v be the subset of the updates in β_i that update $\vec{v}_{i-1}$.

We wll spin up two instances of the dynamic algorithm $\mathscr{A}$. Call these instances D_1 and D_2.

We will describe how to sample an initial H-partite Erdős-Rényi graph for D_1 and then how we perform updates. For the edges in $A \times U$ we simply include each edge iid with probability $1/2$. For The edges in $A \times B$ we include the edge (a_j, b_k) iff $M_0[j][k] = 1$. For edges in $B \times U$ we start by sampling $n-1$ uniformly random Boolean n bit vectors $\vec{p}_j$. Then we set $\vec{p}_n = \vec{v}_0 \oplus \bigoplus_{j=1}^{n-1} \vec{p}_j$. Now we add an edge between (u_j, b_k) iff $\vec{p}_j[k] = 1$. This initializes the graph to random three-partite Erdős-Rényi graph. Now, for updates. We will generate a series of updates α_i from β_i. For each update with probability $1/3$ we flip a random edge in $A \times U$ and add that update to α_i. With probability $2/3$ we instead take the next random update from β_i. If the update was to $M[j][k]$ we flip edge (a_j, b_k) and add that update to α_i. If the update was to $\vec{v}_{i-1}[j]$ we sample a uniformly random value $k \in [1,n]$ and flip the edge (b_j, u_k) and add that update to α_i. When we run out of updates from β_i we will make a series of queries. Notice that all edges in $A \times B \cup B \times U \cup U \times A$ are flipped with the same probability of $1/(3n^2)$. Additionally, each update is independent of the others.

Flip all edges between AU to generate the initial graph for D_2. We apply the same updates, α_i, from D_1 to D_2 we apply to both graphs. Note that this will keep the invariant that D_1 and D_2 have the same edge sets between AB and BU and have opposite edge sets between AU.

Next we will use Lemma 2.2. Note that after each set of updates α_i we will have made the requisite $n\lg^2(n)$ updates to $A \times B \cup B \times U$ required by Lemma 2.2. Roughly one third of all updates happen to AU however, as previously mentioned this maintains our invariant. Now we will make $n\lg^3(n)$ random queries in between each set of updates α_i and α_{i+1}. For any given node the chance it is picked at least once is at least

$$1 - (1 - \frac{1}{3n})^{n\lg^2(n)} > 1 - \frac{1}{e}^{\lg^2(n)/3}.$$

So, the chance that we get the answers for all the vector entries is at least $1 - n\frac{1}{e}^{\lg^2(n)/3}$. We repeat this process n times to get all n queries $M\vec{v}_1, \ldots, M\vec{v}_n$. So, if $\mathscr{A}$ succeeds with probability $1-\delta$ per query then after $\tilde{O}(n^2)$ queries and updates we will solve an instance of OMv with probability at least $1 - 2n^2\delta - 3n\frac{1}{e}^{\lg^2(n)/3}$. We need $2n^2\delta + 3n\frac{1}{e}^{\lg^2(n)/3} < 1/60$ which happens if $\delta < \frac{1}{n^2 240}$.

So, if the OMv hypothesis is true, and we spend $n^{3-\varepsilon}$ time preprocessing for $\varepsilon > 0$ then $n^2 U(n) + n^2 Q(n) = \tilde{\Omega}(n^3)$. □

THEOREM 5.2. *We are given an algorithm $\mathscr{A}$ which runs in graphs with Erdős-Rényi dynamic random graph updates and uniformly random queries across all nodes of $\#\triangle_a$ which succeeds with probability at least $1-\frac{1}{n^2 1680}$ and has preprocessing time $n^{3-\varepsilon}$ for $\varepsilon > 0$. Let $\mathscr{A}$'s update time be $U(n)$ and $\mathscr{A}$'s query time be $Q(n)$. Then if the OMv hypothesis holds $n^2U(n)+n^2Q(n) = \tilde{\Omega}(n^3)$.*

Proof. Consider taking a random sequence of updates that produces a graph as described by Lemma 5.2, lets call the series of updates that produces this graph G. We randomly intersperse updates within the partitions A, B and U. There are $3\binom{n}{2}$ internal edges and $3n^2$ edges between partitions. So, with probability $\binom{n}{2}/(\binom{n}{2}+n^2)$ we update a random edge inside a partition, and the rest of the time we use the updates from G. Call this new series of updates G'. Now, we will maintain a few different versions of G'. We maintain versions with just the nodes in a single partition G'_A, G'_B, G'_U. We also maintain versions with the nodes of two partitions $G'_{AB}, G'_{BU}, G'_{UA}$. We will run $\mathscr{A}$ on all of these graphs. To determine the number of triangles through a node x in G we can simply take the count from G' and subtract the counts from G'_{AB}, G'_{BU}, and G'_{UA} and then finally add the counts from G'_A, G'_B, and G'_U. Note that all of these graphs look like uniformly randomly updated graphs. We can union bound across all 7 of these graphs to get our probability of $1-\frac{1}{n^2 1680}$. □

So, we have our first lower bound in our standard model of average-case dynamic graph updates. Note that small changes to our problem statement have drastically changed how hard this problem is. When we are counting the triangles through a fixed point s the problem is easy. When we are counting triangles through a queried point a this problem is hard. We will now move on to the problem of counting four cycles.

6 Lower Bounds for Path Counting

In this section we will show the hardness of counting st paths of length 5. This is equivalent to counting st paths of length at most 5 (we can count all paths of length $1,2,3$, and 4 with $O(1)$ update and query time so we can add or remove these counts efficiently). We will demonstrate this hardness via reduction from OuMv. This will give us hardness from the OMv hypothesis.

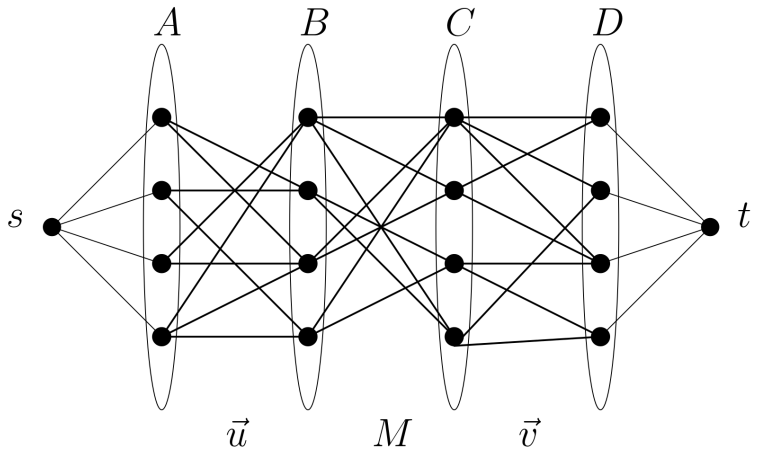

Figure 4: In our reduction the number of paths from s to A to B to C to D to t will be equal to $\vec{u}M\vec{v}$ if sA is complete and Dt is complete. As in our previous reductions we represent $\vec{u}$ and $\vec{v}$ with n^2 edges instead of n edges. The extra edges once again make the random edge flips change the vectors with a high enough proportion.

We will first show that the problem is hard when sA and Dt are complete. We will then show the problem of counting st 5-paths is hard in a st-5-path-partite Erdős-Rényi graph. So, we will show that we can make the edge sets sA and Dt look random instead of complete.

LEMMA 6.1. *Let H_S be a 5-path with two fixed points s and t are the endpoints. Consider the problem of counting the number of H_S graphs in a graph produced by randomly flipping allowed edges in a H_S-partite graph. Call this problem st5-partite st-5-path counting.*

A dynamic algorithm for st5-partite st-5-path counting which has a pre processing time of $n^{3-\varepsilon}$ for $\varepsilon > 0$ must have an update time $U(n)$ and query time $Q(n)$ that respect $n^{2+o(1)}U(n) + n^{o(1)}Q(n) = \tilde{\Omega}(n^3)$ if they produce the correct answer per query with probability at least $239/240$ if the OMv hypothesis is true.

Proof. The reduction is depicted in Figure 4. We have a single node s then sets A, B, C, and D and finally we have a single node t. Additionally we have that $|A| = |B| = |C| = |D| = n$. We restrict n to being even (this is for later convenience). We will use the same trick we have used in previous lemmas where we start by treating some edge sets as complete and then create both that edge set and its inverse and track graphs with both the edge set and the flipped edge set to return the count in the complete graph. Let the nodes a_i be the nodes in A where $i \in [1,n]$. Define the nodes b_i, c_i, and d_i similarly.

We will now describe how we sample edges in $A \times B \cup B \times C \cup C \times D$. We will call the sequence of updates to these edges in particular α_ℓ. We sample an instance of biased average-case OuMv. Let $M_0, \vec{v}_0, \vec{u}_0$ be the uniformly random vectors at the start. Let β^ℓ be the sequence of all biased updates in the i^{th} set of $n\lg^2(n)$ biased random updates. Let β^ℓ_M, β^ℓ_u, and β^ℓ_v be the subsets of β^ℓ that flip bits in M_ℓ, $\vec{u}_\ell$, and $\vec{v}_\ell$ respectively.

So, first consider the case where all edges between s and A exist and also all edges between D and t exist and these edges can't be flipped when random updates occur. After we justify the hardness in this case we will demonstrate how to return to the case we want to consider. For convenience of notation let a_j be the j^{th} node in the set A. Similarly define b_j, c_j, and d_j. For initializing the graph we sample M_0 and add the edge (a_j, b_k) iff $M_0[j][k] = 1$. Note that the edges in $A \times B$ will now look Erdős-Rényi (each edge exists iid with probability $1/2$). Now for $D \times A$ we start by sampling $n-1$ random n bit Boolean vectors $\vec{p}_j$. Then iff $\vec{p}_j[k] = 1$ we add the edge (d_j, a_k). We set the last vector $\vec{p}_n = \vec{u}_0 \oplus \bigoplus_{j=1}^{n-1} \vec{p}_j$. Note that this means $\vec{u}_0 = \bigoplus_{j=1}^{n} \vec{p}_j$. Further note that because $\vec{u}_0$ is a uniformly randomly sampled Boolean vector, the bits of $\vec{p}_n$ are sampled iid 1/0 with probability $1/2$ conditioned on all $\vec{p}_j$ $j < n$. Now iff $\vec{p}_n[k] = 1$ we add the edge (d_n, a_k). For $B \times C$ we do a similar procedure where we sample $n-1$ random vectors $\vec{q}_j$ and then sample $\vec{q}_n = \vec{v}_0 \oplus \bigoplus_{j=1}^{n-1} \vec{q}_j$. Then iff $\vec{q}_j[k] = 1$ we add the edge (c_j, b_k). Now note that all edges in $A \times B$ and $B \times C$ are included or excluded iid with probability $1/2$. So $D \times A \cup A \times B \cup B \times D$ is a three-path-partite Erdős-Rényi graph.

Now we will explain how to make an update in the graph given an update from each of β^ℓ_M, β^ℓ_u, and β^ℓ_v to generate $n\lg^2(n)$ updates in our graph. For all bit flip updates $M[j][k] \in \beta^\ell_M$ we flip the edge (b_j, c_k). For all bit flip updates $\vec{u}[k] \in \beta_u$ we sample a uniformly random $j \in [1,n]$ and flip the edge (a_j, b_k). For all bit flip updates $\vec{v}[k] \in \beta_v$ we sample a uniformly random $j \in [1,n]$ and flip the edge (c_j, d_k). Every edge in $A \times B \cup B \times C \cup C \times D$ has a probability of being flipped of $1/(2n^2)$. All updates in biased updates are independent. We take each update in β^ℓ in order and apply the appropriate procedure (depending on if it is from β^ℓ_M, β^ℓ_u, or β^ℓ_v). Call this set of updates α^ℓ.

The number of paths from s to t of length 5 is going to be the number of paths $s - a_{i_1} - b_{i_2} - c_{i_3} - d_{i_4} - t$. Let $\vec{p}_i$ be the vector where $\vec{p}_i[j] = (a_i, b_j)$. Define $\vec{q}$ similarly as the vector where $\vec{q}_i[j] = (d_i, b_j)$. Now consider that the number of paths through a_i and d_ℓ will be equal to $\vec{p}_i M \vec{q}_\ell$. So the total number of paths will be:

$$\sum_{i \in [1,n]} \sum_{\ell \in [1,\ell]} \vec{p}_i M \vec{q}_\ell = \sum_{i \in [1,n]} \vec{p}_i M \left(\sum_{\ell \in [1,\ell]} \vec{q}_\ell \right) = \sum_{i \in [1,n]} \vec{p}_i M \vec{v} = \vec{u} M \vec{v}.$$

So, we can solve the biased average-case parity OuMv problem with a dynamic five path counting algorithm when we make sA and Dt complete and have no updates to these edge sets. We will now explain how to make these edge sets random. We will make four calls to dynamic algorithms to do this.

For notation we will define E_{sA} as the edge set between sA and E_{Dt} as the edge set between D and t in our 'first' dynamic algorithm call. We will also define $\bar{E}_{sA}$ and $\bar{E}_{Dt}$ as the inverses of these edge sets (so if $(s, a_i) \in E_{sA}$ then $(s, a_i) \notin \bar{E}_{sA}$ and vice versa). Finally, as slight abuse of notation, let E be the set of edges between A and B, B and C, and finally C and D. We make calls to our dynamic st five path counting algorithm on four sets of updates where by the time of the first query and for all updates after that we maintain that the first call has a graph, G_1 made up of edges $E_{sA} \cup E \cup E_{Dt}$, the second graph, G_2 has edges $\bar{E}_{sA} \cup E \cup E_{Dt}$, for G_3 we have $E_{sA} \cup E \cup \bar{E}_{Dt}$, and finally the fourth graph, G_4 is $\bar{E}_{sA} \cup E \cup \bar{E}_{Dt}$. Now note that if we get graphs into this configuration at the start then every future update (flipping a random edge (x, y)) will maintain this structure. If we flip an edge in E_{sA} and $\bar{E}_{sA}$ then they will continue to be opposite edge sets. Now, note that if we count st five paths in all four graphs G_1, G_2, G_3, and G_4 the sum of all of these counts will equal $\vec{u} M \vec{v}$. Any path $s - a_{i_1} - b_{i_2} - c_{i_3} - d_{i_4} - t$ that existed in the graph with complete edge sets sA

and Dt will exist in exactly one of the four graphs (specifically the graph where (s,a_{i_1}) and (d_{i_4},t) exist). Now when we make updates to the graph we sample an edge in sA or Dt to flip with probability $2n/(2n+3n^2)$ and we make an update from α^ℓ with probability $3n^2/(2n+3n^2)$. We need one finally thing. We need to justify that after $\approx \lg^2(n)n^2$ updates we can generate a random edge set E and also random edge sets $E_{sA}, \cup E_{Dt}, \bar{E}_{sA}$, and $\bar{E}_{Dt}$.

To get the edge sets simply sample an Erdős-Rényi st5path-partite graph and then flip our edge sets sA and/or Dt. Note that the flipped edge set is just as likely to occur as the unflipped version and are thus indistinguishable from the Erdős-Rényi st5path-partite distribution.

So, in total, with four calls to our dynamic solver for the counting st five path problem we can solve biased average-case parity OuMv. So, by union bound if we get the correct answer at least $239/240$ of the time for the five path problem we will answer the biased average-case parity OuMv problem correctly with probability $59/60$ which gives us our bound on updates and queries. Note that by Theorem 2.2 the solving the parity biased updates problem with probability $59/60$ is hard from OMv. □

We will now use this lemma and our theorem 3.1 to get our result.

THEOREM 6.1. *Consider the problem of answering the number of st-5 paths in a graph produced by the average-case dynamic update model with probability at least $1-10^{-10}$. Call this problem average-case dynamic counting st-5 paths.*

If an algorithm has $n^{3-\varepsilon}$ pre-processing time for some constant $\varepsilon > 0$ then if its update time is $U(n)$ and the query time is $Q(n)$ we have that $n^2U(n)+nQ(n) = \tilde{\Omega}(n^3)$ if the OMv hypothesis is true.

Proof. Let H_S be a five path with two fixed points s and t are the endpoints.

In Lemma 6.1 we show that the problem of counting H_S graphs is hard in an uniformly randomly updated H_S-partite graph. We will now use Theorem 8.1 to show that counting H_S in randomly updated graphs with no H_S partite restriction are hard. Our value of k is 5 so we have an overhead of 2^{25} from Theorem 8.1. From an algorithmic perspective this washes out as a constant. We do need to track this for the probability. We bound the probability as follows $1-(60\cdot 4\cdot 2^{25})^{-1} < 1-10^{-10}$. □

7 Counting the number of s-t paths of length 1,2,3, and 4

In this section we will give fast algorithms for dynamically counting $s-t$ paths of length 1, 2, 3, and 4 in Erdős-Rényi dynamic graphs. First we note the obvious, that paths of length 1 and 2 can be counted in $O(1)$ time dynamic. Next, we will show that there is a fast algorithm even for s-t paths of length 3 and 4! You might wonder, where does it stop? What about s-t five paths in random graphs? In section 6 we showed that counting s-t paths of any constant length greater than 5 requires $\Omega(n)$ time per update or $\Omega(n^2)$ time per query.

We will start with the trivial results counting paths of length 1 and 2.

LEMMA 7.1. *With preprocessing time $O(\min(n,m))$ counting the number of st paths of length 1 and 2 in a graph with dynamic updates of edge insertion and deletion requires $O(1)$ time in the worst-case (and thus also the average-case).*

Proof. For paths of length 1 we simply check if there is an edge between s and t in time $O(1)$, if there is the answer is 1 otherwise the answer is 0. We only need to maintain the existence of single edge, so we can ignore all other updates.

For paths of length 2 we preprocess the graph to get a count of the number of two length paths, call this count c. This can be done in $O(n)$ time (simply iterate over all nodes u that could be the center node in the two length path $s-u-t$). If there are at most m edges we can instead do this in $O(m)$ time. For every edge (s,u) that exists we check if (u,t) also exists. If an edge is added or deleted with no endpoint of s or t we ignore it. If an edge is added or deleted with one endpoint in $\{s,t\}$ and the other endpoint is some other node $v \notin \{s,t\}$ then we will do an update. If we add (s,v) for example we then query in $O(1)$ time for (v,t). If (v,t) exists in the graph then increment the count by 1. If we were deleting (s,v) and (v,t) existed in the graph we would decrement the count by 1. If we add or delete (t,v) we do the same process with s and t switched. So we check for the edge (v,s). This takes $O(1)$ time per update. □

Next we consider paths of length 3. The extra difficulty here is that (u,v) edges now matter. When adding a (u,v) edge it is easy to update our count, simply check if (s,u), (s,v), (u,t), and (v,t) exist. However, what can we do when

we add (s,u) for example? We need to know the number of two length paths from u to t. To do this we will now show that in average-case graphs we can maintain this information efficiently.

LEMMA 7.2. *Let G be a graph that is updated in an average-case dynamic fashion. G has a fixed vertex set V. Counting the number of length two paths from a fixed point s and all points $u \in V$ can be tracked in $O(1)$ time dynamically in the average-case dynamic setting with preprocessing time $O(n^2)$. A query is a given point u and we must return the number of two length paths to s.*

Proof. The preprocessig process will save the counts of all two length paths between s and all points $u \in V$ in variable c_u. To compute these values we can take $O(1)$ time for all pairs of nodes v,u to check if the path $s-v-u$ exists. If the path exists then we increment c_u.

Now, if we add or delete the edge (u,v) we may need update the counts of two length paths from s to u and from s to v. If (s,u) exists and we added the edge increment c_v if we deleted the edge then decrement c_v. If (s,v) exists and we added the edge increment c_u if we deleted the edge decrement c_u. All of these operations take $O(1)$ time.

If we add the edge (s,u) we may need to spend $O(n)$ time to update nodes v adjacent to u. However, the chance we update an edge adjacent to s is $1/n$. This gives an overall expected time of $O(1)$.

So all updates take expected time $O(1)$. □

Now we can count the number of paths from s to t of length 3.

LEMMA 7.3. *Counting the number of length three paths between s and t can be done in $O(1)$ update and query time in the average-case dynamic model with preprocessing time $O(n^2)$.*

Proof. We will track the count of three length paths in c. Let $w_{s,u}$ be the count of the number of length two paths from s to u. Let $w_{u,t}$ be the count of the number of length two paths from t to u. We will maintain these values using Lemma 2.2. We need to initialize c to do this we take $O(1)$ time for all pairs of nodes u and v and check if $s-u-v-t$ exists. If the path exists we increment c. This will initialize c to the correct value.

If we add the edge (u,v) increment c if (s,u) and (s,t) exist. If we delete (u,v) decrement c if (s,u) and (s,t) exist.

If we add the edge (s,u) (or symmetrically (t,u)) we now want to increment c by the number of two length paths from u to t. So we increment by $w_{u,t}$ which we have been tracking with the algorithm from Lemma 7.2. If we delete the edge (s,u) (or symmetrically (t,u)) we now want to decrement c by $w_{s,u}$. All of these operations take $O(1)$ time. Reading the count c takes $O(1)$ time. □

We now want to count paths from s to t of length 4. Note that now if we add or delete (u,v) we need to check if $s-u-v-w-t$ paths exist. We can almost do this by checking if (s,u) exists and adding the count of all two length paths from v to t. However, to get an accurate count we must decrement by the number of two length paths where w is actually u, because, $s-u-v-u-t$ is not a length four path. If we add an edge (s,u) then we need to count the number of three length paths from u to t. We do this last step in $O(n)$ time and here we use the fact that the probability of adding or deleting an edge adjacent to s or t is $1/n$. So, the additional complexity for this problem will come from wanting to avoid paths with multiple instances of the same vertex and the difficulty of tracking length 3 paths from all nodes to s and t.

LEMMA 7.4. *Counting the number of length four paths between s and t can be done in expected time $O(1)$ update and query time in average-case dynamic graphs where every edge is 'flipped' with equal probability on each update. This algorithm uses preprocessing time $O(n)$ when started on an empty graph, and $O(n^\omega)$ otherwise.*

If you simultaneously maintain all four-paths between s and all nodes in the graph using n copies of this data structure then the updates can be maintained in expected time $O(n)$. The queries, (s,v), take $O(1)$ time to return the number of paths between s and v for any vertex v in the graph. This algorithm requires preprocessing time $O(n^2)$ when starting on an empty graph and $O(n^\omega)$ otherwise.

Proof. Once again we will track the count of our paths in c. We will use Lemma 7.2 once again. For pre-processing we will want the count for all nodes v and v' the number of two length paths between them. We can do this by taking

the adjacency matrix A and squaring it using fast matrix multiplication in n^ω time. We also need to pre-process the number of four paths. We can similarly count this with fast matrix multiplication. We count the number of four walks from s to t with A^4 and then we need to remove the walks that repeat nodes. There are three such walks to remove $s-u-v-u-t$, $s-u-s-v-t$ and $s-u-t-v-t$ (note a walk could fall in two categories by being $s-u-s-u-t$ for example). We can count all $s-u-v-u-t$ walks in $O(n)$ time. We can count all $s-u-s-v-t$ and $s-u-t-v-t$ in linear time. We can count the double counted paths $s-u-s-u-t$ and $s-u-t-u-t$ in $O(n)$ time. So we can count all the valid paths in $O(n^\omega)$ time.

We have two cases. In the first we add (or delete) an edge (u,v) in the second we add or delete the edge (s,u) or (u,t). Let $w_{s,u}$ be the number of two length paths from s to u and define $w_{t,u}$ similarly as the number of two length paths from t to u. We track all of these values with Lemma 7.2.

For the first case, where we are adding (u,v) to the graph we add four values to c. First if (s,u) exists then we add $w_{t,v}$, we subtract one from this number if (u,t) exists (to remove the count of the path *suvut* we would otherwise be counting). Second if (s,v) exists then we add $w_{t,u}$, we subtract one from this number if (u,t) exists. Third, if (t,u) exists then we add $w_{s,v}$, we subtract one from this number if (u,s) exists. Fourth, if (t,v) exists then we add $w_{s,u}$, we subtract one from this number if (v,s) exists. If we are deleting the edge (u,v) subtract the amount we would have added had we added (u,v). This process takes $O(1)$ time.

For the second case, consider adding the edge (s,u). Now for all nodes $v \notin \{s,t,u\}$ check if (u,v), if it does add $w_{t,v}$ to the count. We need to remove paths counted here of the form *suvut*, so once again, we decrease c by one if (u,t) exists. We did this process for every node v so this takes $O(n)$ time. However, the probability that we have flipped an edge adjacent to either s or t is $O(1)/n$, for an expected time per update of $O(1)$. For deleting the edge (s,u) decrease c by the amount we would have added if we added (s,u). If you add or delete the edge (t,u) do the same procedure as above, but switch s and t.

This gives us an expected time of $O(1)$ for updates and queries.

Next we will consider the version where we make n copies of this data structure and track the number of four length paths between s and u for all $u \in v$. To pre-process the lemma 7.2 data structures we take n^ω time, we simply want to maintain the same thing for all entries. The count of two paths between all points is shared across all instances. By putting A^4 we count four walks between all pairs of points, including all pairs (s,u). For the rest of the pre-processing as we mention earlier all types of walks can be removed in $O(n)$ time, so we can do this for all (s,u) pairs in $O(n^2)$ time. This gives a total of $O(n^\omega + n^2)$ time which is $O(n^\omega)$. We track four paths for each pair (s,u) separately, but can use a shared data structure for two paths. Now when updates occur we simply call the earlier described data structure for all entries, with a shared access to the data structure for all pairs of points. When a query occurs we simply read off the count of four paths. □

We will now present a quick proof that *worst-case st* four path requires $m^{1/2}$ time per update. We present this lower-bound to note the differences between worst-case and average-case algorithms.

LEMMA 7.5. *Given a graph with n nodes and edge updates (both insertions and deletions) an algorithm for dynamically counting st four paths requires $m^{1/2-o(1)}$ time per update from the OMv hypothesis.*

Proof. From [12] we have that counting *st* three paths requires $m^{1/2-o(1)}$ time per update. Now consider inserting a node s' and adding a single edge from s' to s. Now ask for $s't$ four paths. Four paths from s' to t are three paths from s to t. So, *st* four path must take at least as long. □

Now, you might hope for an efficient algorithm in the average-case with $O(1)$ update time for paths of length 5. However, in section 6 we will show that no efficient algorithm exists for counting paths of length 5 if OMv holds in the worst-case. This will give us a tight understanding of the transition of hardness based on path length in the average-case from constant to linear update times.

Counting s4-cycles Finally we will address the question of counting four cycles with a fixed point. We will present this here because we get our algorithms through our path algorithms.

COROLLARY 7.1. *Counting the number of four cycles with a fixed point s paths can be done in expected time $O(1)$ update and query time when in the average-case dynamic model. This algorithm uses preprocessing time $O(n^\omega)$.*

Proof. Use Lemma 7.4 but set s and t to be the same point. This is a four cycle with a fixed point s. □

Finally we will note the lower bound in the *worst-case* for this problem (as opposed to the average-case).

LEMMA 7.6. *Given a graph with n nodes and arbitrary edge updates (both insertions and deletions) an algorithm for dynamically counting s four cycles requires $m^{1/2-o(1)}$ time per update from the OMv hypothesis.*

Proof. From [12] we have that counting *st* three paths requires $m^{1/2-o(1)}$ time per update. Now consider inserting an edge from s to t, all three paths from s to t are now counted by an algorithm for counting four cycles. However we are also, potentially, counting some cycles we would like to remove. We will now discuss how to remove all of these undesirable counts efficiently.

First we want to remove the count of all four cycles that don't involve both s and t. To do this we maintain three additional dynamic graphs. Let G be the original graph. Consider the same graph but with t removed, call this G_t. Let the count of the number of four cycles in G be c_G. Let the number of four cycles in G_t be c_{G_t}. The number of four cycles involving both s and t is $c_G - c_{G_t}$.

There is one more bad case. we want to count for cycles where s and t are adjacent. That is, we want to count four cycles where the edges are $(s,u)(u,v)(v,t)(t,s)$. There can also exist four cycles where t and s are opposite each-other $(s,u)(u,t)(t,v)(v,s)$. We can use Lemma 7.1 to count all two paths from s to t. Call this count c_2. The number of these bad four cycles where s and t are opposite is equal to $\binom{c_2}{2}$ where we define $\binom{0}{2} = \binom{1}{2} = 0$. So, if we return the value $c_G - -c_{G_t} - c_2$ we will return the number of length three *st* paths. □

8 Counting the Number of s-t Paths of Length 5

In this section we will present an algorithm for *st* five path. This algorithm is tight to the lower bound we presented earlier.

We present two algorithms. One algorithm minimizes query time. The other algorithm minimizes update time. These algorithms address both parts of the lower bound we will later present ($n^2U(n) + nQ(n) = \tilde{\Omega}(n^3)$). So, we present one algorithm that hits the bound for the update time and one that hits the update time for the query.

THEOREM 8.1. *Let the average-case st-5-path counting problem be the problem of dynamically counting five paths starting at s and ending at t where every update picks two nodes uniformly at random and flips the edge between those two nodes.*

There exists an algorithm for average-case st-5-path counting with preprocessing time $O(n^\omega)$, update time $O(1)$, and query time $O(n^\omega)$ (recall that ω is the matrix multiplication constant).

There exists an algorithm for average-case st-5-path counting with preprocessing time $O(n^2)$ when started on a empty graph and $O(n^\omega)$ otherwise , update time $O(n)$, and query time $O(1)$.

Proof. We will use lemma 7.2 to track all length two paths from s and t. Let u be an arbitrary node and let $w_{s,u}$ be the number of length 2 paths from s to u. According to lemma 7.2 both the updates and queries take both $O(1)$ time while using only $O(n^2)$ preprocessing. Thus, using this data structure only adds a constant amount of time to the update and query time. To initialize all the saved values $w_{s,u}$ we simply square the adjacency matrix, A, to count all two length paths from s to all nodes u, i.e. we take the entry $A^2[s][u]$, which takes time $O(n^\omega)$. We will initialize the count of five paths using the following method (we also use it for the version with fast updates and slow queries).

First we will describe how to count *st* five paths in $O(n^\omega)$ time using the above data structure. This gives the algorithm with constant update and $O(n^\omega)$ query time. We will use E to refer to the edge set at the time of the query (or during preprocessing). The idea is to (1) first count all *walks* (i.e., node repetitions are allowed) from s to t of length 5 and then (2) subtract out all length-5 walks that are not paths. (1) We will count all length-5 walks from s to t in $O(|E|)$ time by running a BFS and counting for each node the number of length 1 walks to s, then running another layer of the BFS to count the number of length 2 walks to s by using the length-1 counts of the neighbors', etc, until we have counted all the length-5 walks. Let C be the resulting number. (2) Now we simply need to subtract out the number of walks that are not paths. As there are no self-loops in the graph, there are three possible ways that the same node u can appear on a s-t path: in the second and fifth locations ($su??ut$)[2], the second and forth locations ($su?u?t$), and

[2]We use the question mark to indicate arbitrary nodes.

the third and fifth locations ($s?u?ut$). Note that the both of the last two patterns can occur at the same time ($suvuvt$). We now explain how to count the number of each pattern that occurs.

First ($su??ut$): Let us name the central nodes ($suxyut$). Note that u, x, and y forms a triangle in the walk. Thus, by determining the number of triangles of nodes that are incident to both s and t we can subtract out all these types of walks. More specifically, we cube the adjacency matrix A using matrix multiplication in $O(n^\omega)$ time. This will give a count of all triangles through each node. Let S be the set of all nodes u that have edges to both s and t. We can compute S in $O(n)$ time by checking every vertex. Summing up the triangle count for each node in S_1 gives exactly the number of paths of the first pattern. Call this count $C_{2,5}$. Note that there is no double counting between two nodes u and v because they are mutually exclusive as a path can have only one node as the second node in its path.

Second ($su?u?t$): Let us name the unspecified nodes ($suxuyt$). Notice that x can be any node in the neighborhood of u, N_u. Notice that the number of possible nodes y is simply w_{ut}, which we have been tracking using the data structure of lemma 7.2. So the number of paths of the second pattern going through u is simply $|N_u|w_{ut}$. We can compute the maintain the in and out degrees of all nodes in $O(1)$ time per update , and compute all these multiplied in $O(n)$ time after that. As before there is no double counting between two nodes u and v because they are mutually exclusive. Summing $|N_u|w_{ut}$ over all nodes u gives the desired count, which we call $C_{2,4}$.

Third ($s?u?ut$): Let us name the unspecified nodes ($sxuyut$). This pattern is handled symmetrically to the second pattern by replacing t with s, i.e., it just requires to sum up $|N_u|w_{su}$ over all u. Let us call the resulting count $C_{3,5}$. It takes time $O(n^2)$ to determine the resulting value.

Now notice that our counts for $C_{2,4}$ and $C_{3,5}$ both count one kind of walk twice, namely the $suvuvt$ walk. We want to count the number of these so we can subtract it off to remove the double counting. For all edges (u,v) such that (s,u) and (v,t) both exist we have one such path. Similarly if (s,v) and (u,t) both exist we have another path. Thus, each edge (u,v) can contribute to zero, one, or two double counted $suvuvt$ walks and we can determine in $O(n^2)$ time this number for all edges. Call this count summed over all edges $C_{2,3,4,5}$. Now the number of length-5 s-t paths is $C - C_{2,5} - C_{2,4} - C_{3,5} + C_{2,3,4,5}$.

Second we will present the version with $O(n)$ update time and constant-time queries. Let G be the random graph with vertex set V and let G_s be the graph with s removed. Now consider running $n-2$ copies of the algorithm from Lemma 7.4, $\mathscr{A}_{7.4}$, one for each node $u \in V \setminus \{s,t\}$, that counts the number $w_{u,t}$ of length-4 paths from u to t in G_s. When started on an empty graph all these data structures can be initialized in time $O(n^2)$, otherwise it takes time $O(n^\omega)$. When receiving an update, we pass the update to all $n-2$ copies, except if the updated edge is incident to s. This takes $O(1)$ time per graph for $O(n)$ time total to run the algorithm for all $n-2$ copies. Then we add up the 4-path counts $w_{u,t}$ for each neighbor u of s and store the result C. This takes time $O(n)$ as well. Now when queried we can simply return C in $O(1)$ time. So, with $O(n^2)$, resp. $O(n^{1+\omega})$ preprocessing, $O(n)$ time updates, and $O(1)$ time queries we can solve the average-case five path problem.

□

9 Counting Four Cycles

The main idea to count the number of four cycles is to track the number $w_{u,v}$ of length-2 paths between any pair u, v of nodes in the graph. Note that the number of four cycles with u and v on opposite corners of the cycle (u,x,v,y) is equal to $\binom{w_{u,v}}{2}$, where we define $\binom{1}{2} = 0$ for our purposes. The sum C of these values for all node pairs (u,v) counts every cycle four times: Consider the $abcd$ four cycle we will count this in $w_{a,b}$, $w_{b,a}$, $w_{b,d}$, and $w_{d,b}$. Thus we simply return $C/4$ as query answer.

We show below how to implement this algorithm so as to match our lower bound for counting average-case four cycles.

THEOREM 9.1. *Let the worst-case four-cycle counting problem be the problem of dynamically counting four cycles where every update is a deletion or insertion of an edge.*

There exists an algorithm for worst-case four cycle counting with preprocessing time $O(n^\omega)$, worst-case update time $O(n)$ and worst-case query time $O(1)$.

There exists an algorithm for average-case four cycle counting with preprocessing time $O(n^\omega)$, worst-case update time $O(1)$ and worst-case query time $O(n^\omega)$ (note ω is the matrix multiplication constant).

Proof. We first present an algorithm with $O(n)$ update and $O(1)$ query time. In a data structure we maintain (1) the

degree, i.e., the size of the neighborhood, of every node as well as (2) the number $w_{u,v}$ of length-2 paths between any pair u, v of nodes in the graph, and (3) the sum $C^* := \sum_{u,v\in V, u\neq v} \binom{w_{u,v}}{2}$. Note that $C^* := \sum_{u,v\in V, u\neq v} \binom{w_{u,v}}{2}$ equals four times the total number of four cycles: Think of each $\binom{w_{u,v}}{2}$ as counting all four cycles where the first vertex is u and the third is v. Thus, at each query we return $C^*/4$. Next we describe how to initialize and maintain each of these values.

(1) The degree of every node can be trivially initialized in $O(n^2)$ total time and maintained after an update in constant time per operation.

(2) When starting we can set all $w_{u,v}$ values in $O(n^\omega)$ time. Maintaining all of these for every node takes $O(n)$ time per update as only the endpoints of the updated edge and their neighbors can be affected. Specifically, if edge (u,w) was updated then only the counts $w_{u,v}$ for every neighbor v of w and the counts $w_{x,w}$ for every neighbor x of u needs to be updated.

(3) During preprocessing we need to count all four cycles in the initial graph. We can do this in $O(n^\omega)$ time by taking the adjacency matrix A to the fourth power to count all length-4 walks from all nodes u back to u. Let C be the total number of length-4 walks. This counts some non-cycles specifically (a) $u-v-w-v-u$ and (b) $u-v-u-w-u$. Pattern (a) consists of a path of length 2 originating at u and pattern (b) consists of two neighbors v and w of u. With the above information we can determine the number C_a and C_b of these two patterns for each node u in $O(n)$ time for a total of $O(n^2)$ time for all nodes. Note that walks with pattern $u-v-u-v-u$, called pattern (c), is counted in both C_a and C_b of these, and there are as many such walks for u as its degree. Thus, we can determine the number C_c of all walks of pattern (c) in$O(n)$ time. It follows that $C-C_a-C_b+C_c$ equals $\sum_{u,v\in V, u\neq v} \binom{w_{u,v}}{2}$, and it can be computed in total time $O(n^\omega)$. Note that in the same time we can even keep a per-node count, i.e. the number of four-cycles containing a node u per node u.

Now consider an update operation. If an edge (u,v) is updated, it can change the values of $w_{u,x}$, $w_{x,u}$, $w_{x,v}$, and $w_{v,x}$ for all possible nodes x and the above data structure will provide these values. When updating a value $w_{u,x}$, let $old_{u,x}$ be the old value and $w_{u,x}$ be the new value. When doing this update we will update C^* by $\binom{w_{u,x}}{2} - \binom{old_{u,x}}{2}$. As at most $4n$ w-values changed, need to make at most $O(n)$ such updates to C^*, and these take $O(1)$ time to compute. This gives us our $O(n)$ update and $O(1)$ query time algorithm.

We next present an algorithm with update time $O(1)$ and query time $O(n^\omega)$. Updates simply update the graph, no further data structure is kept. At query time we run the following static four cycle counting algorithm which run in time n^ω as follows: We start by determining the number C of all length-4 walks that start at a node u and return to u in time n^ω for all nodes u by taking the adjacency matrix to the fourth power. We count some four cycles we shouldn't in this way, specifically, $u-a-v-a-u$ for any neighbor a of u and neighbor $v\neq u$ of a. The number C' of these is $\sum a\in V|N_a|(|N_a|-1)$, where N_a is the neighborhood of a. We can compute C' in n^2 time from scratch. Thus, $C-C'$ gives the number of four cycles and it can be computed a in $O(n^\omega)$ time. □

We will now note that there is a faster algorithm when the graph is sparse.

COROLLARY 9.1. *There is an average-case algorithm for counting four cycles dynamically when the graph starts out empty that takes preprocessing $P(n) = m^{5/3}$, amortized update time $U(n) = O(\min(n, m^{2/3}))$, and query time $Q(n) = O(1)$.*

Proof. First note that $m^{2/3} < n$ when $m < n^{3/2}$. So, we can start by using the worst-case algorithm of Hanauer, Henzinger and Hua [12]. This algorithm counts four cycles with $O(m^{5/3})$ preprocessing time, $O(m^{2/3})$ update time and $O(1)$ query time. When we have at least $n^{3/2}$ edges we will then run the algorithm from Theorem 9.1. This takes $O(n^\omega)$ pre-processing time. We can amortize this cost across all $n^{3/2}$ prior updates for a total amortized time of $O(n^{\omega-3/2}+n)$ per update. Note that $\omega < 2.37$ [3], so the amortized time is $O(n)$. □

10 Counting s-triangles in a fully dynamic random graph

We next show that the number of triangles through a fixed node s can be counted even more efficiently.

THEOREM 10.1. *Fix a node s. Let the average-case s-triangle counting problem be the problem of dynamically counting the number of triangles containing s where every update picks two nodes uniformly at random and flips the edge between those two nodes.*

There exists an algorithm for average-case s-triangle counting algorithm with preprocessing time $O(n^2)$, update time $O(1)$, and query time $O(1)$.

Proof. Let C be the number of triangles containing s. During preprocessing we spend $O(n^2)$ time to compute C. We next describe the update operation. In the following we say *update a counter by a value a* to mean that the counter is incremented by a if the update was an insertion and the counter is decremented by a if the update was a deletion.

Case 1: An edge (u,v) with $u \neq s$ and $v \neq s$ is updated. To update C check whether both the edges (u,s) and (v,s) exist and if so, update C by 1.

Case 2: An edge (u,s) was updated. To update C determine the intersection of the neighbors of u and of s. Let a be this number. Update C by a.

The correctness of the algorithm follows immediately. Note that the worst-case running time in Case 1 is $O(1)$ and in Case 2 it is $O(n)$. Furthermore the probability that Case 2 occurs is $2/n$. Thus the expected time per update operation is constant. □

11 Counting the Number of Triangles Through a Queried Point

Here we will present an algorithm for counting triangles through a node that is given as query parameter, i.e., not a fixed node. These algorithms will all work in the worst-case as well.

THEOREM 11.1. *Given a graph with n nodes where updates may delete or add an edge and queries ask for the number of triangles that go through a point u ($\#\triangle_u$) then there are two algorithms.*

There exists an algorithm with preprocessing time $O(n^\omega)$, update time $O(n)$, and query time $O(1)$.

There exists an algorithm with preprocessing time $O(n^2)$, update time $O(1)$, and query time $O(n^2)$.

Proof. For our first algorithm we start by creating an array C where $C[u]$ contains a count of all triangles through u. We populate this array by taking the adjacency list A of the graph and cubing it as the entry $A^3[u][u]$ contains a count of all triangles through u. When edge (u,v) is updated, we maintain A in $O(1)$ time by updating the two associated entries. Additionally we will update the counts in $O(n)$ time by checking for all x if x,u,v is a triangle. If x,u,v was a triangle before the edge was deleted we decrement $C[x],C[u]$, and $C[v]$. If x,u,v is a triangle after the edge is inserted we increment $C[x],C[u]$, and $C[v]$. This takes $O(1)$ time per node x and thus takes $O(n)$ time overall. Our queries take $O(1)$ time looking up the entry in our table C.

For our second algorithm we do no preprocessing, we simply create the adjacency matrix A. We do nothing but keep A updates for each update, thus taking $O(1)$ time. For every query for the triangles through x we simply check for all u and v if x,u,v is a triangle. This takes $O(1)$ time per pair (u,v) and thus takes $O(n^2)$ time. □

References

[1] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. *CoRR*, abs/1402.0054, 2014.

[2] David Alberts and Monika Rauch Henzinger. Average-case analysis of dynamic graph algorithms. *Algorithmica*, 20(1):31–60, 1998.

[3] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021.

[4] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *J. Comput. Syst. Sci.*, 47(3):549–595, 1993.

[5] Enric Boix-Adserà, Matthew Brennan, and Guy Bresler. The average-case complexity of counting cliques in erdős-rényi hypergraphs. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1256–1280. IEEE Computer Society, 2019.

[6] Béla Bollobás and Oliver Riordan. The diameter of a scale-free random graph. *Combinatorica*, 24(1):5–34, 2004.

[7] Colin Cooper, Alan M. Frieze, Kurt Mehlhorn, and Volker Priebe. Average-case complexity of shortest-paths problems in the vertex-potential model. *Random Struct. Algorithms*, 16(1):33–46, 2000.

[8] Daniel da Silva. Proprietades geraes et resolucao das congruencias biniomis. 1854.

[9] Mina Dalirrooyfard, Andrea Lincoln, and Virginia Vassilevska Williams. New techniques for proving fine-grained average-case hardness. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 774–785. IEEE, 2020.

[10] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[11] Oded Goldreich. On counting t-cliques mod 2. *Electron. Colloquium Comput. Complex.*, 27:104, 2020.

[12] Kathrin Hanauer, Monika Henzinger, and Qi Cheng Hua. Fully dynamic four-vertex subgraph counting. *CoRR*, abs/2106.15524, 2021.

[13] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30. ACM, 2015.

[14] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining triangle queries under updates. *ACM Trans. Database Syst.*, 45(3):11:1–11:46, 2020.

[15] Andrew McGregor and Sofya Vorotnikova. Triangle and four cycle counting in the data stream model. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 445–456, 2020.

[16] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614, 2011.

[17] Charalampos E Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *2008 Eighth IEEE International Conference on Data Mining*, pages 608–617. IEEE, 2008.

A Proofs About OMv and OuMv

In this paper we want to work with average-case problems. OMv is a popular dynamic conjecture. We present the average-case hardness of a parity variant of OMv given the worst-case hardness of traditional OMv. We do the same with OuMv.

Reminder of Lemma 2.1 *Let $x \geq \log n$ be an integer. With x calls to parity OuMv we can answer an instance of OuMv with probability at least $1 - n2^{-x}$. Thus the parity OuMv hypothesis is implied by the OMv hypothesis.*

With x calls to parity OMv we can answer an instance of OMv with probability at least $1 - n2^{-x}$. Thus the parity OMv hypothesis is implied by the OMv hypothesis.

Proof. Consider an instance of OuMv with M and n tuples (u_i, v_i). Recall that the input is defined over zero one matrices and vectors. We want to know if $u_i^T M v_i > 0$. To solve this problem with parity OuMv we will make calls with x randomly altered matrices R_j with $1 \leq j \leq x$. Let D_M be a distribution over $\{0,1\}^{n \times n}$ matrices defined as follows: R_j is created by flipping each one to zero with probability 1/2. We will show first that this defines D_M such that if $R_j \sim D_M$ then $R_j[k][\ell] = 0$ if $M[k][\ell] = 0$. However, if $M[k][\ell] = 1$ then $R_j[k][\ell]$ is 0 with probability $1/2$ and 1 with probability $1/2$ iid for each entry.

Consider a specific value i where $u_i^T M v_i = 0$, note that $u_i R_j v_i = 0$ with probability 1 in this case. The probability is 1 because R_j has a strict subset of the 1s in M and all entries are zero or one. So, $u_i R_j v_i$ can only decrease in value

Now consider a specific value i where $u_i^T M v_i \geq 1$. Now note that this sum is

$$\sum_{k \text{ s.t. } u_i[k]=1} \left(\sum_{\ell \text{ s.t. } v_i[\ell]=1} M[\ell][k] \right).$$

Further note that $u_i^T R_j v_i$ is

$$\sum_{k \text{ s.t. } u_i[k]=1} \left(\sum_{\ell \text{ s.t. } v_i[\ell]=1} R_j[\ell][k] \right).$$

Now note that every value of $M[\ell][k]$ in the first sum that is a 1 is in R_j a one or zero with probability $1/2$. Consider the parity of $u_i^T R_j v_i$. Note that we can pick some particular value of $M[\ell][k]$ included in the first sum which is a one. Regardless of the parity of the rest of the sum, this last value is summed into it and is a one with probability $1/2$ and a zero with probability $1/2$. So, if $R_j \sim D_M$ then the parity of $u_i^T R_j v_i$ is 1 with probability $1/2$ if $u_i^T M v_i \geq 1$.

Consider just one R_j gives us a procedure with one sided error, namely when $u_i^T M v_i \geq 1$. However, we produced x instances $R_1, \ldots, R_x$ where $R_j \in D_M$ and run the parity OuMv algorithm for each of them. Now, if for any j we have $u_i^T R_j v_i \equiv 1 \mod 2$ then we know $u_i^T M v_i \geq 1$. By combining answers of all these x calls we get a correct answer for our original OuMv problem (consisting of n queries) with probability at least $1 - n2^{-x}$. The OuMv hypothesis is implied by the OMv hypothesis by Theorem 2.7 from [13].

We will now prove the reduction from OMv to parity OMv. We will be using broadly the same techniques. Given a M and v_i we will produce the randomly altered matrices R_j with $1 \leq j \leq x$ as above. We will make a series of queries to parity OMv to $R_j v_i$ and use this to produce the desired output of OMv. Now note that the value of $(Mv_i)[k]$ is

$$\sum_{\ell \text{ s.t. } v_i[\ell]=1} M[\ell][k].$$

Further note that $R_j v_i$ is

$$\sum_{\ell \text{ s.t. } v_i[\ell]=1} R_j[\ell][k].$$

Now once again if $(Mv_i)[k]$ is zero then $(R_j v_i)[k]$ is zero with probability 1. Next note that if $(Mv_i)[k]$ is non-zero (thus we want to return a vector with a one in the k^{th} position) then there is a $1/2$ chance that $(R_j v_i)[k] = 1 \mod 2$, i.e. that it returns the correct answer. So, for each position in the vector after x calls the chance it is wrong is at most 2^{-x} we have n indexes into our vector for a total probability of correctness of at least $1 - n2^{-x}$. □

Note that if we pick $x = \lg^2(n)$ in the above theorem, we give the correct answer with high probability.

We now present a proof that the hardness of worst case parity OuMv implies the hardness of uniform average-case OuMv. This proof is basically an "online version" of the Blum, Luby, and Rubinfeld proof that matrix multiplication is hard on average [4]. We can show that parity OuMv and parity OMv are both hard on average with the inclusion/exclusion technique explained below.

Reminder of Lemma 2.2 *An algorithm for uniform average-case parity OuMv that succeeds with probability $1 - \varepsilon$ in time $T(n)$ implies a worst-case algorithm for parity OuMv in time $O(T(n))$ that succeeds with probability $1 - 8\varepsilon$.*

An algorithm for uniform average-case parity OMv that succeeds with probability $1 - \varepsilon$ in time $T(n)$ implies a worst-case algorithm for parity OMv in time $O(T(n))$ that succeeds with probability $1 - 4\varepsilon$.

Proof. Use Lemma 2.1 it suffices to show that worst-case parity OuMv implies hardness for the uniform average-case of OuMv. Thus, consider an instance of worst-case parity OuMv, a matrix M with n tuples (u_i, v_i) given in an online manner. Now consider drawing a random matrix R from the uniform distribution of $\{0,1\}^{n \times n}$. Additionally consider

drawing random vectors x_i and y_i uniformly at random from $\{0,1\}^n$. Now consider the following problems:

$$(u_i \oplus x_i)^T (M \oplus R)(v_i \oplus y_i) = a \tag{A.1}$$
$$(u_i \oplus x_i)^T (M \oplus R)(y_i) = b \tag{A.2}$$
$$(u_i \oplus x_i)^T (R)(v_i \oplus y_i) = c \tag{A.3}$$
$$(u_i \oplus x_i)^T (R)(y_i) = d \tag{A.4}$$
$$(x_i)^T (M \oplus R)(v_i \oplus y_i) = e \tag{A.5}$$
$$(x_i)^T (M \oplus R)(y_i) = f \tag{A.6}$$
$$(x_i)^T (R)(v_i \oplus y_i) = g \tag{A.7}$$
$$(x_i)^T (R)(y_i) = h \tag{A.8}$$

Now note that $a \oplus b \oplus c \oplus d \oplus e \oplus f \oplus g \oplus h = u_i^T M v_i$, which is the value we want to extract. Further note that each problem we list above is a uniform average-case parity OuMv problem (by itself, obviously there are correlations between any two of the above instances). So, an algorithm for uniform average-case parity OuMv which succeeds with probability $1-\varepsilon$ implies an algorithm for worst case parity OuMv that succeeds with probability at least $1-8\varepsilon$ by the union bound.

We will now make the same argument for OMv, which is slightly easier. We will generate uniformly random R_i and y_i as described above and then produce the following problem:

$$(M \oplus R)(v_i \oplus y_i) = \vec{a} \tag{A.9}$$
$$(M \oplus R)(y_i) = \vec{b} \tag{A.10}$$
$$(R)(v_i \oplus y_i) = \vec{c} \tag{A.11}$$
$$(R)(y_i) = \vec{d} \tag{A.12}$$

We will now note that $\vec{a}+\vec{b}+\vec{c}+\vec{d} = Mv_i$, which is the value we want to compute. Now note that once again each problem looks individually random (but there are correlations between the problems). So an algorithm for uniform average-case OMv that succeeds with probability $1-\varepsilon$ will succeed with probability at least $1-4\varepsilon$ by the union bound (we get 4 here instead of the previous 8 because we only need to make 4 calls). ∎

Now, we can consider what a fast algorithm for the uniform average-case OuMv problem implies about worst-case OuMv.

Reminder of Theorem 2.1 *Given a dynamic algorithm $\mathscr{A}$ for uniform average-case parity OuMv which succeeds with probability at least $16/17$ and runs in time $T(n)$ we can solve worst-case OuMv with probability at least $1-2^{-\Omega(\lg^2(n))}$ in time $\tilde{O}(T(n))$.*

Given a dynamic algorithm $\mathscr{A}'$ for uniform average-case parity OMv which succeeds with probability at least $16/17$ and runs in time $T(n)$ we can solve worst-case OMv with probability at least $1-2^{-\Omega(\lg^2(n))}$ in time $\tilde{O}(T(n))$.

Proof. By Lemma 2.2 $\mathscr{A}$ implies an algorithm for *worst-case parity OuMv* that succeeds with probability $9/17$ and runs in time $O(T(n))$. If we run this algorithm for worst-case parity OuMv $\lg^4(n)$ times and take the most common answer, we will get the correct answer with probability at least $1-2^{-\Omega(\lg^3(n))}$ in time $\tilde{O}(T(n))$. We call this new algorithm $\mathscr{B}$.

Now consider Lemma 2.1, by making x calls to $\mathscr{B}$ we can solve *OuMv* with probability at least $1-n2^{-x}-x2^{-\Omega(\lg^3(n))}$. Now consider $x=\lg^3(n)$. In this case, we succeed with probability at least $1-2^{-\Omega(\lg^2(n))}$ while running in time $\tilde{O}(T(n))$. This proves the first half of our theorem statement.

We proceed analogously for the OMv case. By Lemma 2.2 $\mathscr{A}'$ implies an algorithm for *worst-case parity OMv* that succeeds with probability $9/17$ and runs in time $O(T(n))$. Now note that we can run this algorithm for worst-case parity OMv $\lg^4(n)$ times and take the most common answer, with probability at least $1-2^{-\Omega(\lg^3(n))}$ we will get the correct answer in time $\tilde{O}(T(n))$, call this new algorithm $\mathscr{B}'$.

Now consider Lemma 2.1, by making x calls to $\mathscr{B}'$ we can solve OMv with probability at least $1-n2^{-x}-x2^{-\Omega(\lg^3(n))}$. Now consider $x=\lg^3(n)$. In this case, we succeed with probability at least $1-2^{-\Omega(\lg^2(n))}$ while running in time $\tilde{O}(T(n))$. This proves the second half of our theorem statement. ∎

All proofs above assumed that we are given a *fixed* uniformly random M and a series of n random vectors $\vec{u}_i$ and $\vec{v}_i$. However, in our actual setting when we transform the random vectors $\vec{u}_i$ and $\vec{v}_i$ into $\vec{u}_{i+1}$ and $\vec{v}_{i+1}$ we will make $\Theta(n\lg^2(n))$ random updates to not just $\vec{u}_i$ and $\vec{v}_i$ but also M. Thus, we need to show that this problem with *changing* M is still hard. That is, we want to show that we can compute $\vec{u}_{i+1}M\vec{v}_{i+1}$ using an M' with a small number of changes. We can generalize this. We want to show that if we make $\Theta(n\lg^2(n))$ random updates to $\vec{u}_i$, $\vec{v}_i$, and M we can still solve the original problem. We will start with the definition of the OuMv and OMv version of biased updates.

REMINDER OF DEFINITION 2.3 *We call the following problem* biased average-case OuMv. *Let n be a positive integer. The initial matrix M_0 and vectors $\vec{u}_0$ and $\vec{v}_0$ are chosen uniformly at random from all possible such n-dimensional Boolean vectors and $n\times n$-dimensional Boolean matrices. Now, $M_{i+1},\vec{u}_{i+1},\vec{v}_{i+1}$ are created from $M_i,\vec{u}_i,\vec{v}_i$ by flipping $n\lg^2(n)$ bits. Each bit is flipped with the following distribution: each bit in M_i is flipped with probability $\frac{1}{3n^2}$, each bit in $\vec{u}_i$ and $\vec{v}_i$ is flipped with probability $\frac{1}{3n}$. In the* biased average-case OuMv *problem for $1\le i\le n\lg^3(n)$ right after the construction of $M_i,\vec{u}_i,\vec{v}_i$ we must return $(\vec{u}_iM_i\vec{v}_i)$ with Boolean multiplication. In the* biased average-case parity OuMv *problem for $1\le i\le n\lg^3(n)$ right after the construction of $M_i,\vec{u}_i,\vec{v}_i$ we must return $(\vec{u}_iM_i\vec{v}_i) \mod 2$.*

DEFINITION A.1. *We call the following problem* biased average-case OMv. *Let n be a positive integer. The initial matrix M_0 and vector $\vec{v}_0$ are chosen uniformly at random from all possible such n-dimensional Boolean vectors and $n\times n$-dimensional Boolean matrices. Now, $M_{i+1},\vec{v}_{i+1}$ are created from $M_i,\vec{v}_i$ by flipping $n\lg^2(n)$ bits. Each bit is flipped with the following distribution: each bit in M_i is flipped with probability $\frac{1}{2n^2}$, each bit in $\vec{v}_i$ is flipped with probability $\frac{1}{2n}$. In the* biased average-case OMv *problem for $1\le i\le n\lg^3(n)$ right after the construction of $M_i,\vec{v}_i$ we must return $(M_i\vec{v}_i)$ with Boolean multiplication. In the* biased average-case parity OMv *problem for $1\le i\le n\lg^3(n)$ right after the construction of $M_i,\vec{v}_i$ we must return $(M_i\vec{v}_i) \mod 2$.*

REMINDER OF THEOREM 2.2 *If a dynamic algorithm $\mathscr{A}$ solves biased average-case OuMv in time $U(n)$ per biased update and $Q(n)$ per query with probability at least $59/60$ then worst-case OuMv can be solved in time $U(n)n^{2+o(1)}+Q(n)n$.*

If a dynamic algorithm $\mathscr{A}'$ solves biased average-case OMv in time $U(n)$ per biased update and $Q(n)$ per query with probability at least $59/60$ then worst-case OMv can be solved in time $U(n)n^{2+o(1)}+Q(n)n$.

Proof. We use the reduction of Theorem 2.1 from worst-case OuMV to (uniform) average-case parity OuMv problem and, thus, assume that we are given an instance of (uniform) average-case parity OuMv with matrix M fixed and the sampled vectors $(\vec{u}_i,\vec{v}_i)$ from $i=0,1,\ldots,n$ given in an online manner. We will show how to turn this into three parity OuMv problems S_1, S_2, and S_3, each with the same set of vectors $(\vec{u'}_i,\vec{v'}_i)$ but with different but correlated i-th matrices such that the XOR of their individual answers gives the answer for the given instance of the (uniform) average-case parity OuMv problem. Each S_k with $k=1,2,3$ will have a small TVD from a *biased* average-case parity OuMv problem. We will use Lemma C.1 to show that this implies a high probability of success on the *biased* average-case parity OuMv distribution.

The initial matrix and vectors M_0, $\vec{v}_0$, and $\vec{u}_0$ are also the initial matrix and vectors for each S_k. Now for $0\le i\le n-1$ given the vector $(\vec{u'}_j,\vec{v'}_j)$ and $(\vec{u}_i,\vec{v}_i)$ (where $\vec{u}_i,\vec{v}_i$) is the next vector pair we have a query on) from the uniform average-case parity OuMv we produce many vector pairs $(\vec{u'}_j,\vec{v'}_j)$ and the j-th matrices for each S_k as follows. We first need to determine the three numbers s_u, s_v, and s_M, where s_u, resp. s_v corresponds to the number of flipped bits that will be applied to $(\vec{u'}_{j-1},\vec{v'}_{j-1})$ to generate $(\vec{u'}_j,\vec{v'}_j)$ and s_M will be used for constructing the three matrices. They are generated as follows: Initially set s_u, s_v, and s_M all to 0 and then $n\lg^2(n)$ times increment one of them, each of them with equal probability. Then perform a check described below. We use a different random procedure to sample the vectors if it succeeds vs fails.

Recall that we are give a (uniform) random sampled $\vec{u}_i$. Additionally, we will maintain the invariant that $\vec{u'}_{j-1}$ is sampled from a distribution that has a TVD of at most $2^{-\Omega(n^2)}$ from uniform vectors (See Lemma C.2. Let b_u be the number of indices where $\vec{u'}_{j-1}$ and $\vec{u}_{i+1}$ differ. Let b_v be defined analogously. We call s_u (resp. s_v) *valid for b_u (resp. b_v)* if (a) $s_u\ge b_u$ and (b) the parity of s_u equals the parity of b_u. With probability $1/4$ s_u and s_v will match the

parity constraints and $s_u < b_u$ or $s_v < b_v$ happens with probability at most $2^{-\omega(1)}$, i.e., with probability $1/4 - 2^{-\omega(1)}$ both are valid. We will sample an S and describe slightly different procedures bellow depending on if S is valid or not. Our goal is that individually, each of S_1, S_2, S_3 are series of updates sampled from a distribution with a TVD of zero from the biased updates distribution. Additionally, whenever we sample a valid S we will answer the query of $(\vec{u}_i, \vec{v}_i)$ and we can increment i to solve the next vector pair. When S is invalid we will effectively be solving a random problem unrelated to our query. Note that after sampling s_u and s_v at most $\lg^4(n)$ times they are valid for both b_u and b_v with probability at least $1 - 2^{-\lg^3(n)}$.

In the case where we have selected valid s_u and s_v, we create the actual updates for $\vec{u'}_{j-1}$ and $\vec{v'}_{j-1}$ to create $\vec{u'}_j = \vec{u}_i$ and $\vec{v'}_j = \vec{v}_i$. Consider $\vec{u'}_{j-1}$ and the set of indices that change when going to $\vec{u'}_i$. On each of those indices of $\vec{u}_i$ we will generate an odd number of flips and on all other indices we will generate an even number of flips. We sample a set of updates to the indices of $\vec{u}_i$ conditioned on having s_u and respecting the transformation from $\vec{u}_i$ to $\vec{u}_{i+1}$ as follows: we add one flip for each index that changes between $\vec{u}_i$ and $\vec{u}_{i+1}$, and then we sample $(s_u - b_u)/2$ indices in $[1,n]$ uniformly at random and add two bit flips at the sampled indices. Finally, we randomize the order of the flips in β_u. Applying these bit flips to $\vec{u}_i$ gives the vector $\vec{u'}_{i+1}$. We denote the set of updates for $\vec{u}_i$ by β_u. Then, we perform the same procedure for v to generate β_v creating vector $\vec{v'}_{i+1}$. Finally to create the list of updates for both, we merge the lists β_u and β_v and randomize the order of updates between them. Call this list of updates $\beta_{u,v}$.

In the case where S was invalid (either s_u or s_v was invalid, instead sample s_u uniformly random updates iid and them to β_u. Sample s_v uniformly random updates iid and them to β_v. Note that regardless of what S we sample we use it and fill S with flips.

Now we will analyze the TVD of the flips we perform in β_u to the uniformly sampled flips. We will use $Bin_\mu[x]$ to refer to the Binomial distribution where a success happens with probability μ and there are x samples. First, by Lemma C.2 we have that a vector that results from the sequence of $Bin_{1/3}[n\lg^2(n)]$ bit flips on $\vec{u}_i$ and a uniformly random vector have TVD of at most $2^{-\Omega(\lg^2(n))}$. Note that (1) we can use the linearity of XOR here to start from $\vec{u}_i$ instead of the empty vector and (2) the length of a sequence of $Bin_{1/3}[n\lg^2(n)]$ bit flips on $\vec{u}_i$ has the same distribution as the number s_u picked by one iteration of the above procedure. This is relevant for the case where S is valid. In the case where S is invalid we are sampling flips in a distribution that is indistinguishable from the biased updates distribution we describe. So, the distribution of the vectors $\vec{u'}_{i+1}$ and $\vec{v'}_{i+1}$) that result from the above procedure and the distribution over two uniform random vectors have TVD of at most $3 \cdot 2^{-\Omega(\lg^2(n))} = 2^{-\Omega(\lg^2(n))}$. Additionally, the distribution over updates to achieve these outcomes is has a TVD of at most $2^{-\Omega(\lg^2(n))}$.

Now we need to sample the bit flips in M. We have two cases based on whether s_M is even or odd. If the number of samples is even then sample $s_M/2$ iid uniformly random updates to M three times, call these possible updates Δ_a^i, Δ_b^i, and Δ_c^i. We will additionally define the set F_i to be the empty set of updates for the even case. If s_M is odd then sample $\lfloor s_M/2 \rfloor$ entries for Δ_a^i, Δ_b^i, and Δ_c^i. We additionally sample as single bit flip in the matrix M and let F_i be the set of just this flip in the odd case. We create three sampled flips $f_1 = \Delta_a^i \oplus \Delta_b^i \oplus F_i$, $f_2 = \Delta_a^i \oplus \Delta_c^i \oplus F_i$, and $f_3 = \Delta_b^i \oplus \Delta_c^i \oplus F_i$, to be used by S_1, resp. S_2, resp. S_3, one per instance. Note that no matter whether s_M is even or odd, we produced three instances with $n\lg^2(n)$ uniformly random samples. So, the produced updates in M are indistinguishable (in each instance) from uniformly random samples.

Now we show that if we make calls to the three biased updates instances and sum their outputs we will get $\vec{u}_{i+1} M \vec{v}_{i+1}$. Note that the matrix used by the first instance for operation $i+1$ is $M \oplus \bigoplus_{j=1}^{i+1}(\Delta_a^j \oplus \Delta_b^j \oplus F_i)$, for the second it is $M \oplus \bigoplus_{j=1}^{i+1}(\Delta_a^j \oplus \Delta_c^j \oplus F_i)$, and for the third it is $M \oplus \bigoplus_{j=1}^{i+1}(\Delta_b^j \oplus \Delta_c^j \oplus F_i)$. Thus,

$$\vec{u}_{i+1}(M \oplus \bigoplus_{j=1}^{i+1}(\Delta_a^j \oplus \Delta_b^j \oplus F_i))\vec{v}_{i+1} \oplus \vec{u}_{i+1}(M \oplus \bigoplus_{j=1}^{i+1}(\Delta_b^j \oplus \Delta_c^j \oplus F_i))\vec{v}_{i+1} \oplus \vec{u}_{i+1}(M \oplus \bigoplus_{j=1}^{i+1}(\Delta_c^j \oplus \Delta_a^j \oplus F_i))\vec{v}_{i+1} =$$

$$3\vec{u}_{i+1} M \vec{v}_{i+1} \oplus 2(\vec{u}_{i+1} \bigoplus_{j=1}^{i+1}(\Delta_a^j \oplus \Delta_b^j \oplus \Delta_c^j)\vec{v}_{i+1}) \oplus 3\vec{u}_{i+1}(\bigoplus_{j=1}^{i+1} F_i)\vec{v}_{i+1} =$$

$$(\vec{u}_{i+1} M \vec{v}_{i+1} \oplus \vec{u}_{i+1}(\bigoplus_{j=1}^{i+1} F_i)\vec{v}_{i+1}) \mod 2.$$

We will save the values of all sampled F_i, these have between zero and one entries each and each such entry consist of 0 or 1 and a location in M. Computing $c = \vec{u}_{i+1}(\bigoplus_{j=1}^{i+1} F_i)\vec{v}_{i+1}$ takes $O(i)$ time in the following way. We initialize c to be zero at the beginning of the reduction and whenever the j-th vector pair of the uniform average-case parity OuMv instance arrives and we have determined the new F_j then we perform the following operation: If F_j is non-empty and has location $M[a][b]$ then we XOR our current value of c with $\vec{u}_{i+1}[a]\vec{v}_{i+1}[b]$. We can then return the value $\vec{u}_{i+1}M\vec{v}_{i+1} \mod 2$ by XORing c to the value $(\vec{u}_{i+1}M\vec{v}_{i+1} \oplus \vec{u}_{i+1}(\bigoplus_{j=1}^{i+1} F_i)\vec{v}_{i+1})$ received by XORing the query answers of S_1, S_2, and S_3. Computing c for all n OuMv queries requires $O(n^2)$ time across all n pairs $(\vec{u}_{i+1}, \vec{v}_{i+1})$. This shows the correctness of our reduction.

Lemma C.1 shows that if two distributions have TVD ε then an algorithm that succeeds on one distribution with probability p must succeed on the other with probability at least $p - \varepsilon$. We will use this to overcome the distance of our problem from the TVD of at most $2^{-\Omega(\lg^2(n))}$ from the uniform distribution. Assume we have an algorithm for biased average-case parity OuMv with failure probability at most $1/60$. If we apply it to one of the instances S_k, it has failure probability at most $1/60 + 2^{-\Omega(\lg^2(n))}$. Thus, the total failure probability of the algorithm when applied to S_1, S_2, and S_3 is at most $3/60 + 3 \cdot 2^{-\Omega(\lg^2(n))}$, which is a smaller probability of failure than $1/3$. Thus, the algorithm could be used to answer the given (uniform) average-case parity OuMv problem with probability at least 2/3. Our reduction uses $O(n^2 \log^n) = O(n^{2+o(1)})$ updates and n queries. Combined with Theorem 2.1 the result follows.

We restate the above proof and give the proof for biased average-case parity OMv in Appendix A.

For OMv we will have a similar procedure. Let s_M be the number of all flips in M. We will generate Δ_a^i, Δ_b^i, Δ_c^i, and F_i in the same way as before. Now our equation for these changes with delta is:

$$((M + \bigoplus_{j=1}^{i+1}(\Delta_a^j \oplus \Delta_b^j \oplus F_i))\vec{v}_{i+1} \oplus (M + \bigoplus_{j=1}^{i+1}(\Delta_b^j \oplus \Delta_c^j \oplus F_i))\vec{v}_{i+1} \oplus (M \oplus \bigoplus_{j=1}^{i+1}(\Delta_c^j \oplus \Delta_a^j \oplus F_i))\vec{v}_{i+1}) \mod 2 = \tag{A.13}$$

$$M\vec{v}_{i+1} \oplus \bigoplus_{j=1}^{i+1} F_i\vec{v}_{i+1} \mod 2. \tag{A.14}$$

Now, the time to compute $\vec{c} = \bigoplus_{j=1}^{i+1} F_i\vec{v}_{i+1}$ is $O(n)$. We will form the n bit vector $\vec{c}$ by initializing it to zero. Then if F_i is empty we do nothing to $\vec{c}$. If F_i contains a flip at $M[a][b]$ we set $\vec{c}[a]$ to $\vec{c}[a] \oplus \vec{v}_{i+1}[b]$. Note that $i + n = O(n)$. Once again we can subtract off the value of $\vec{c}$ to get out our desired answer. We do $O(n)$ computation $O(n)$ times for a total of $O(n^2)$ computation.

So, we can use the biased updates approach for OMv as well. The math on the probabilities of failure remain the same and we solve OMv if we have a success probability of at least $1 - 1/60$. ∎

B Inclusion Edgesclusion Proofs

In this section we prove Theorem 3.1. It shows the following: Assuming that counting labeled copies of H_S in a dynamic H_S-partite Erdős-Rényi graph is hard, then counting the same subgraph unlabeled H_S in dynamic Erdős-Rényi graphs is also hard. This reduction has an overhead that grows as 2^{k^2} for subgraphs H_S with k nodes. As a result, this reduction is fully efficient when $k = O(1)$. This reduction becomes inefficient when $k = \Omega(\sqrt{\lg(n)})$. In this paper we look at graphs with $k \leq 6$, so this reduction is efficient for our purposes.

First we make a notational comment: In this section we are counting the number of copies/occurrences of a small graph H_S in a larger graph G. To do this we will count *subgraphs* L of H_S recursively. Unfortunately, the problem of counting the number of occurrences of H_S in a larger graph is often called 'subgraph counting'. To avoid confusion we will use the phrase 'counting copies of H_S in G' to refer to the problem of counting occurrences of H_S and reserve the term *subgraphs* for proper subgraphs of H_S. This will make the section clearer.

Reminder of Theorem 3.1 *Imagine we are given a dynamically updated graph with random updates, where some edges are marked as not-allowed. The graph will be split into k partitions and edges with 'disallowed' updates must be the complete edge sets between two partitions. These edges will not be randomly updated by the random updates. We use this structure to make our proofs easier in the body of the paper.*

Let H_S be a graph with k nodes. Assume that the problem of counting H_S in a graph with random updates among the allowed edges in an H_S-partite graph requires $2^{\binom{k}{2}+k}U(n)$ time per update and $2^{\binom{k}{2}+k}Q(n)$ time per query as long

as at most $O(2^{\binom{k}{2}+k}(n^2+P(n)))$ time is spent preprocessing and gives correct answers to queries with probability $1-\delta$. If the allowed edges are a constant fraction of all edges then the average-case H_S counting problem requires at least $\Omega(U(n))$ time per update or at least $\Omega(Q(n))$ time per query as long as at most $O(P(n))$ time is spent preprocessing. This new algorithm will give correct answers to queries with probability at least $1-\delta 2^{\binom{k}{2}+k}$.

B.1 High Level Idea and Definitions To prove the theorem we will show the contrapositive: If one can efficiently count the number of unlabeled copies of H_S in dynamic Erdős-Rényi graphs, then one can efficiently count the number of labeled copies of H_S in dynamic H_S-partite Erdős-Rényi graphs. The problem of counting labeled copies of H_S (when H_S has k nodes) in G requires that we are given a partition of the vertices of G into k sets $V_1,\ldots,V_k$. Throughout this section we will assume that the graph G has such a partition (it isn't necessarily k-partite, it simply has these marked sets). The proof proceeds in two steps: (A) First (Section B.2) we use the inclusion-edgesclusion technique to argue that if one can efficiently count the number of *unlabeled* copies of H_S in dynamic *ErdRen graphs*, then one can efficiently count the number of *unlabeled* copies of H_S *with exactly one vertex per partition* in dynamic Erdős-Rényi graphs. (B) Then (Section B.3 and Lemma B.5) we show if one can efficiently count the number of *unlabeled* copies of H_S with exactly one vertex per partition in dynamic Erdős-Rényi graphs, then one can efficiently count the number of *labeled* copies of H_S in *dynamic H_S-partite Erdős-Rényi graphs.*

The high level idea of our approach is this. Given two non-overlapping sets of nodes A and B. A and B won't necessarily be the whole graph. If you randomly choose the edges between A and B, then both the random edge set you chose and its inverse are equally likely! So E_{AB} and $\bar{E}_{AB}$ look just as plausible. Additionally, consider two graphs G and G' such that G has the edges between A and B defined by E_{AB} and G' has the edges between A and B defined by $\bar{E}_{AB}$. Now imagine counting some subgraph H_S in G and G' and returning the sum over both graphs. We will use the inclusion-exclusion technique to count only versions of H_S that have *exactly one node* in each set of nodes A and B. This sum will include twice the counts of all subgraphs H_S that use no edge between A and B, as well as a count of all subgraphs H_S that use *any single* edge between A and B. (Note that it cannot include subgraphs with multiple edges between A and B as it can include only exactly one node of A and B.) So, if we wanted to count all subgraphs H_S that don't use an edge between A and B we could (1) ask for the counts ct and ct' over G and G', (2) compute the number ct^- of subgraphs H_S that use any edge (a,b), and (3) return $(ct+ct'-ct^-)/2$. Note that (2) requires basically counting for each edge (a,b) over subgraphs H' that have one fewer edge, namely $H_S \setminus (a,b)$. It turns out this general approach can be used recursively. Note that we can use the standard inclusion exclusion technique to force exactly one vertex of our counted subgraphs to be in each partition so we only need to handle cases where there are zero or one edges between partitions.

In the following we consider the graph G and then consider for each edge set, V_i, all possibilities of the edge set being on and off. We are able to recurse down to a point of considering just the count of how many edges exist between the partitions. We will be using the same general approach as [9], however, we cannot use their lemmas off the shelf as they do not address the dynamic setting nor the setting where some partitions are single fixed nodes (like *st*-path).

To do this we need to define the various edge-set flipped graphs. We will borrow the definition from [9].

DEFINITION B.1. (SLIGHTLY ALTERED DEFINITION 3 FROM [9]) *Let G be a Erdős-Rényi graph with every edge included with probability $1/2$ (note G can also be the outcome of many random edge flips). Let G have vertex partitions $V_1,\ldots,V_k$ and the edges between partitions are called $E_{i,j}$ $\forall i,j \in [1,k]$ where $i<j$. Note that in the case of fixed nodes we will allow some of the edge partitions to be single nodes.*

For $i \neq j$ label all $|V_i|\cdot|V_j|$ possible edges between V_i and V_j with numbers in $[1,2]$ as follows. Edges that exist in G are labeled 1. The rest of the edges are labeled 2. For $\ell \in [1,2]$, let $E^{\ell}_{i,j}$ be the set of all edges of label ℓ.

Let $G^{(\ell_1)(\ell_2)\ldots(\ell_{\binom{k}{2}})}$ be the graph formed by choosing edge sets $E^{\ell_1}_{1,2}$, $E^{\ell_2}_{1,3},\ldots,E^{\ell_{\binom{k}{2}}}_{k-1,k}$. Let X_G be the set of all possible $2^{\binom{k}{2}}$ graphs $G^{(\ell_1)(\ell_2)\ldots(\ell_{\binom{k}{2}})}$.

Thus for each pair of vertex sets V_i, V_j with $i \neq j$ each graph in X_G either contains all of the edges between that vertex pair in G or all edges that are not in G. We will use X_G to build our recursive step. Furthermore we will use the following notion of a labeled subgraph of H_S for our recursion.

DEFINITION B.2. *Recall that H_S with k nodes is labeled if the nodes are given the labels $v_1,\ldots,v_k$. A labled subgraph, L, of H_S is a subgraph of k' nodes of H_S that retain the same labels as the original nodes in H_S.*

For example say H_S were an *st* 5 path, $v_1 - v_2 - v_3 - v_4 - v_5 - v_6$ where $S = \{(v_1, s), (v_6, t)\}$. Then L might be a two-path $v_1 - v_2 - v_3$ and the disconnected edge $v_5 - v_6$. But, L could also be the distinct labeled subgraph of a two-path $v_1 - v_2 - v_3$ and the disconnected edge and $v_4 - v_5$.

B.2 Traditional Inclusion Exclusion The first step in our reduction will involve ensuring that we are only counting unlabeled subgraphs H_S that have exactly one vertex per set. This concept originates from Daniel da Silva in 1854 [8]. This technique used in [9] (see Lemma 5.5) and [5] (see Lemma 3.10). This is a standard technique. With 2^k calls on a graph with k nodes you can count all copies of that graph that appear with exactly one vertex in each partition of a larger graph. We will present a version specific to our problem here.

THEOREM B.1. *Let G be a graph with k sets of vertices $V_1, \ldots, V_k$ that partition the vertices (G is not necessarily k-partite). Let T_i be the set of all subgraphs G' formed by the union of i of the vertex sets. (So T_2 when $k = 3$ has three graphs with vertex sets $V_1 \cup V_2$, $V_2 \cup V_3$ and $V_3 \cup V_1$.) Let $f_{H_S}(G')$ be the number of unlabeled copies of a graph H_S in G where H_S has k nodes. Let $C_{H_S}(T_i) = \sum_{G' \in T_i} f_{H_S}(G')$. That is, $C_{H_S}(T_i)$ is the count of all instances of H_S in all graphs that are formed by the union of i of G's vertex sets. Note that $C_{H_S}(T_k) = f_{H_S}(G)$.*

Then the number of unlabeled copies of H_S is equal to $\sum_{j=0}^{k-1} (-1)^j C_{H_S}(T_k)$.

Proof. Consider a specific copy of H_S, say that it has nodes in exactly ℓ partitions $V_{i_1}, \ldots, V_{i_\ell}$. Let us determine how many times it is counted and with what signs.

The first case is $\ell = k$, note that such an H_S must have exactly one vertex per partition as H_S has exactly k nodes. Now, such an H_S is counted in $C_{H_S}(T_k)$ once and appears in no other counts.

Next, consider an $\ell \in [1, k-1]$ (note that $\ell = 0$ is nonsensical, H_S has k nodes and must appear in some partition). A given copy of H_S appears only in $C_{H_S}(T_i)$ for $i \geq \ell$. How many graphs in S_i does this H_S appear in? Any versions where all its ℓ partitions are selected and the remaining $i - \ell$ partitions are selected from the remaining $k - \ell$ partitions. The parity of the counts of the S_i values flip, so in the final sum this individual H_S is counted

$$\pm \sum_{i=\ell}^{k} (-1)^i \binom{k-\ell}{i-\ell} = \pm \sum_{p=0}^{k-\ell} (-1)^p \binom{k-\ell}{p} = 0$$

times. So, the count $\sum_{j=0}^{k-1} (-1)^j C_{H_S}(T_k)$ returns only H_S that have at least one node in each partition, which implies exaclty one node per partition. □

Now, for intuition lets discuss why this isn't enough. If H_S were a clique, we would be done. However, if H_S is not a clique, the count we return after inclusion exclusion still includes many H_S that fail to follow the labeling. This is not surprising as we counted only unlabeled subgraphs. Intuitively, we are counting subgraphs H_S which use edges that do not fulfill the labeling requirement. For example, if H_S were an *st* 3-path, $v_1 - v_2 - v_3 - v_4$ where $S = \{(v_1, s), (v_4, t)\}$ we might be counting a path $v_1 - v_3 - v_2 - v_4$ which is an *st* three path, but, isn't the kind of *st* three path we want to count.

In worst-case graphs we can use inclusion-exclusion along with empty edge sets between some pairs of partitions to count the desired labeled subgraphs with only a unlabeled subgraph counting algorithm and inclusion-exclusion. However, this idea won't work in the average-case, because empty edge partitions are very far from an Erdős-Rényi graph. However, in the average-case we can't do this. The graph wouldn't look Erdős-Rényi! The name for a graph where all the 'disallowed' edge sets (i.e. edge sets that do not appear in H_S) are empty but all other edge sets look Erdős-Rényi is a Erdős-Rényi H_S-partite graph. So our crucial observation here is that an unlabeled counter for H_S in Erdős-Rényi H_S-partite graphs can be used to count labeled H_S in Erdős-Rényi H_S-partite graphs. The reason is as follows: As unlabeled subgraphs that we counted in these graphs have exactly one vertex per partition and Erdős-Rényi H_S-partite graphs contain no edges between pairs of partitions that have no edge in the corresponding labeling of H_S, every counted unlabeled subgraph must fulfill the labeling of H_S. *Thus the count of unlabeled subgraphs with exactly one vertex per partition in Erdős-Rényi H_S-partite graphs equals the count of labeled subgraphs in Erdős-Rényi H_S-partite graphs.*

In the next section we will prove that an algorithm for counting unlabeled copies of H_S with exactly one vertex per partition in Erdős-Rényi graphs can be used to count labeled copies of H_S in H_S-partite Erdős-Rényi graphs, which is our goal (B).

B.3 Unlabeled to Labeled Counting Consider a labeled graph H_S. We want to count the number of labeled copies of H_S in an Erdős-Rényi H_S-partite graphs. For this we will use a counter of *unlabeled* copies of H_S in Erdős-Rényi graphs which are not H_S-partite. Recall that if the graph were H_S-partite traditional inclusion-exclusion would trivially solve our problem.

Our proof works by recursion. During the recursive proof we use L to denote the labeled subgraph of H_S whose occurrences we want to count in G. The base case of the recursion consists of graphs L that consist of only one edge. As we show next this base case is easy to track both dynamically and statically.

LEMMA B.1. (BASE CASE LEMMA) *We are given a graph G with k partitions $A_1, \ldots, A_k$ (some of them may be single nodes) with n nodes and m edges. We can update the stored counts of the number of edges between all pairs of A_i and A_j with $i \neq j$ in $O(1)$ time per edge insertion/deletion dynamically, given that we use $O(k^2)$ space to store the information. In the static setting we can return counts of the number of edges between all pairs of either A_i and A_j in $O(m)$ time.*

Proof. In a dynamic algorithm we track the number of edges between all $O(k^2)$ pairs of partitions A_i and A_j in a table of size k^2. Every edge inserted or deleted updates the count of the sets containing its endpoints in constant time.

In the static case we can scan over all edges and count the number of edges between all possible pairs of partitions. □

Note that this allows us to track the number of occurrences of any labeled subgraph L that consists of exactly one edge.

Recursion We will devote the rest of the section to explain the recursion up from the base-case. Let us briefly discuss the idea behind the recursion. The first idea is that we sum the counts of unlabeled H_S across multiple graphs in X_G where between some subset of edge sets V_{i_ℓ} and V_{j_ℓ} we include all possible combinations of the graphs using the edges sets $E^1_{i_\ell,j_\ell}$ or $E^2_{i_\ell,j_\ell}$. This sum will, metaphorically, make the edge set between V_{i_ℓ} and V_{j_ℓ} complete. We can use this to efficiently count the unlabeled H_S copies that don't match our desired labels (because we want to subtract that count out) as long as we know the counts of all the smaller labeled subgraphs of H_S. We recurse on this idea, to get the count of some labeled subgraph L of H_S. We use the unlabeled counts of H_S in many graphs in X_G and the counts of labeled subgraphs of L that are smaller than L in G.

We first define the subgraphs of H_S. Note that H_S consists of k labeled vertices $v_1, \ldots v_k$.

DEFINITION B.3. *Let $Z_{H_S}(\ell, e)$ be the set of labeled subgraphs of H_S with ℓ nodes and e edges.*

If $J \in Z_{H_S}(\ell, e)$ and $L \in Z_{H_S}(\ell', e')$ then $J \subset L$ (in words J is a subgraph of L) if every labeled vertex $v_j \in J$ also appears in L with the same label and every edge $(v_i, v_j) \in J$ also appears in L.

Outline. The lemma below show how to count the number of subgraphs using double recursion, once on the number of vertices, and once on the number of edges. More specifically, to show the claim for a graph H_S with v nodes and e edges, we will first recurse on all subgraphs with fewer than v vertices (vertex recursion). We use their counts to count all subgraphs with v nodes and at most $v-2$ edges. We use this result as the base case for a second recursion (edge recursion), where we count all subgraphs with v nodes and $e+1 > v-2$ edges. In the latter recursion we recurse on subgraphs with v nodes and at most e edges.

Vertex recursion. First observe that the recursion on subgraphs with fewer vertices is trivial if the graph to count is disconnected.

LEMMA B.2. *Let G be a graph with n nodes, m edges and k labeled partitions of the vertices $V_1, \ldots, V_k$ (recall that G is not necessarily k-partite). If we have the counts of all labeled subgraphs of H_S in G with exactly one vertex in each partition of the graph of size less than s vertices, then we can compute the number of labeled subgraphs in G that are the union of two disconnected, vertex-disjoint labeled subgraphs of H_S of total size s or less.*

Proof. Let one be labeled subgraph $L_{\hat{S}}$ and the other be labeled subgraph $L'_{\hat{S}'}$. Given that they share no vertices, we can simply multiply the number of subgraphs $L_{\hat{S}}$ and $L'_{\hat{S}'}$. □

Next we show how to use this fact in our vertex recursion. More specifically, given a count of all subgraphs of fewer than v vertices we use it to count all subgraphs with v vertices and up to $v-2$ edges.

LEMMA B.3. *Let G be a graph with n nodes, m edges and k labeled partitions of vertices $V_1,\ldots,V_k$ (some partitions may be single nodes). Given the count of all labeled subgraphs of H_S in all graphs in X_G with exactly one vertex in each partition of the graph with less than v vertices, we can count all labeled subgraphs with v vertices and at most $v-2$ edges in $\tilde{O}(m)$ time.*

Proof. A subgraph of v vertices and at most $v-2$ edges must be disconnected. Thus, we can use Lemma B.2 to compute the count of the subgraph by multiplying the counts of its connected components. □

Edge recursion. Next we show how to count subgraphs with v vertices and $e+1$ edges, assuming we know the counts for all subgraphs with v vertices and up to e edges [3].

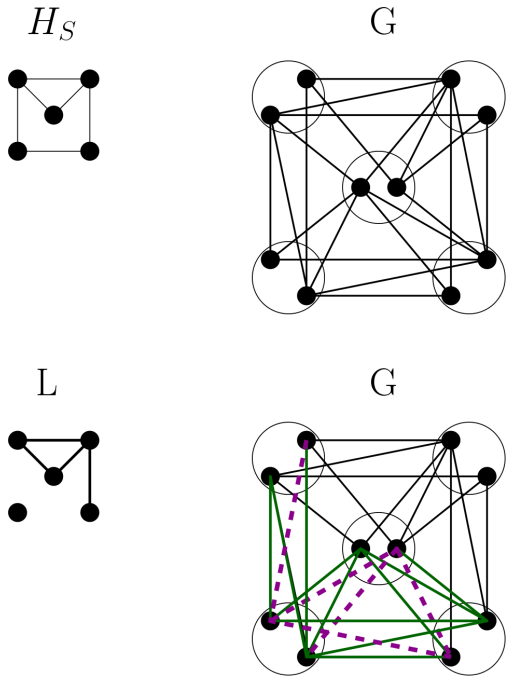

Figure 5: Full lines represent edges in the graph G, so edge sets labeled with 1 e.g. $E^1_{i,j}$. The dashed lines represent the edges in the inverse of the edge sets (i.e. $E^2_{i,j}$). The first graph G represents G with no changes made. The second copy has edges in black which we don't flip when trying to count copies of L. In the second copy green edges represent the edges that exist in the original G (that is $E^1_{i,j}$ edge sets) but which exist in between partitions that we flip when learning the count of L. In Lemma B.4 we take unlabeled counts of subgraphs in all input graphs in X_G where edges in G between partitions that correspond to edges in L are set to $E^1_{i,j}$. In our picture, this is the black (thin line) edges. Then, in our example there are four pairs of vertices in L (which correspond to partitions in G) that do not have an edge between them in L. All 2^4 possible choices of four tuples of edge sets between these partitions will appear in X_G.

LEMMA B.4. *Let G be a graph with the partitions of the vertices into k sets and let L be a labeled subgraph of the labeled graph H_S such that there is at most one node per partition with v vertices and $e+1$ edges for any $v>0$ and $e\geq 0$. Assume we are given the counts of the number of unlabeled subgraphs of H_S which have exactly one vertex in each partition in all graphs in X_G (see Definition B.1). Additionally, assume that we are given the counts of all* labeled *subgraphs of H_S consisting of less than or equal to v vertices with $[0,e]$ edges in G.*

Using both of these counts we can count the number of not-necessarily induced copies of the labeled subgraph L in G in time $O((k!+k\log n)\cdot 2^{k^2})$.

[3] Basically, we will use Lemma 5.7 from [9], but we need to reprove it as [9] requires that there are exactly n vertices in a partition and our graphs might not fulfill this requirement.

Proof. Let H_S have $v_H = k$ vertices and $e_H \geq e+1$ edges. Let the subgraph L be given as a list of $v \leq k$ vertices labeled as being in partitions $i_1, \ldots, i_v$ and $e+1$ edges between these partitions in total. Let S_L be the set of all pairs (x,y) such that there is an edge between partition i_x and i_y in L, let $\bar{S}_L$ be the set of all pairs with an edge in H_S but not in S_L, and let R be the remaining pairs in $[1,k] \times [1,k]$. Note that this gives a partition of $[1,k] \times [1,k]$ such that $S_L \cup \bar{S}_L$ equals the edge set of H_S. Thus, $|R| = \binom{k}{2} - e_H$ and $|R \cup \bar{S}_L| = \binom{k}{2} - e - 1$.

Our general approach is that we (1) first overcount by determining the number of unlabeled copies of H_S in a suitable subset of X_G and then (2) we correct that count by subtracting out all copies of H_S in graphs G' where the vertex assignment of vertices of H_S to vertices in G' does not agree with the labeling of L. We describe each of these steps next in detail.

(1) Consider the subset of instances in X_G where the edges between partitions in S_L (for example E_{i_1,i_2}) are all set to be the version labeled (1) ($E^{(1)}_{i_1,i_2}$). Call this subset $S_G[L]$. What graph instances are in $S_G[L]$? These are all graphs that contain $E^{(1)}_{i_1,i_2}$ for every (i_1,i_2) in S_L, but for all $(i_1,i_2) \in R \cup \bar{S}_L$ the graphs contains either $E^{(1)}_{i_1,i_2}$ or $E^{(2)}_{i_1,i_2}$. Note that there are $2^{\binom{k}{2}-e-1}$ many such graphs.

We can enumerate all graphs in $S_G[L]$ in time $O(2^{k^2})$ and for each graph G' retrieve the number of unlabeled copies of H_S in G' as each G' belongs to X_G. Let us call their sum $c_{S_G[L]}$. Note that each unlabeled copy of H_S in a graph G' of $S_G[L]$ corresponds to a labeling of H_S as each vertex of the copy is assigned to a vertex partition of G'. This is the reason why we count different labelings of H_S in the rest of the proof.

What will the count $c_{S_G[L]}$ contain? It will count the number of labelings of H_S that appear if the graph G were to have a complete bipartite graph between all pairs of partitions in $R \cup \bar{S}_L$, but, note that each labeling of H_S is counted with multiplicity equal to the number of graphs in $S_G[L]$ in which it appears. But this depends on how many of its edges belong to $R \cup \bar{S}_L$: If ℓ of its edges belong to $R \cup \bar{S}_L$, it is counted $2^{\binom{k}{2}-e-1-\ell}$ times as for $\binom{k}{2} - e - 1 - \ell$ many pairs of partitions in $R \cup \bar{S}_L$ the edge label (1) or (2) is not fixed. Thus we showed the following proposition.

PROPOSITION B.1. *If ℓ of edges of H_S belong to $R \cup \bar{S}_L$, then each labeling of H_S is counted in $2^{\binom{k}{2}-e-1-\ell}$ graphs of $S_G[L]$.*

Given that L is a labeled subgraph of H_S, at least one labeling of H_S will share all $e+1$ edges and v vertices of L. So our count $c_{S_G[L]}$ will include a count of the labeled subgraphs L, but we must subtract from $c_{S_G[L]}$ all labelings of H_S that do not agree with the labeling of L. The resulting number will be the number of copies of L in G. We describe next how to do this.

(2) Given the counts in G of all small subgraphs we can count how many labelings of H_S exist that match up only partially with L and remove these from the count $c_{S_G[L]}$. Our approach is as follows: There are $O(2^{k^2})$ labeled subgraphs L' of L. For each of them we determine first how often it appears in G. As L' is labeled, we can get this count $c_{L'}$ recursively by the assumption of the lemma. For each such occurrence L'_G of L' we need to determine in how many ways it can be "extended" into a labeling of H_S, i.e. how many labelings of H_S exist that agree with the labeling of L' on all its vertices. We do this in two steps:

(a) Let $J_{L,L'}$ be a complete graph on k vertices where all edges in $L - L'$ are excluded and let $c_{J_{L,L'}}$ be the count of the number of labelings of H_S that exist in $J_{L,L'}$. Note that we can compute this in time $O(k!)$ by exhaustive search over all possible labelings of H_S that embed in $J_{L,L'}$.

(b) Next we determine for each labeling of H_S embedded in $J_{L,L'}$ (i) in how many graphs of $S_G[L]$ it occurs and (ii) what its count is in these graphs. Let $e_{L'} \leq e$ be the number of edges and $v_{L'} \leq v$ be the number of vertices in L'. (i) Note that by construction $e_H - e_{L'}$ of edges of H_S belong to $R \cup \bar{S}_L$ and, thus, by Proposition B.1 each labeling is counted in $2^{\binom{k}{2}-e-1-e_H+e_{L'}}$ graphs of $S_G[L]$. All these graphs have edges between all the partitions for which H_S has edges. (ii) Given one of these graphs G' we are left with counting how often the labeling of H_S is counted, i.e., how many different occurrences of H_S there are in G'. Recall that each such occurrence $H_{S,G'}$ needs to use exactly the edges of the occurrence of L'_G in G' for the vertex partitions for which L' contains an edge, i.e., for each edge (i_1,i_2) of $H_S \cap L'$ there is only one choice of edge for $H_{S,G'}$, namely it must use exactly the same edge as L'_G. However, for all remaining edges of H_S all possible combinations of choices of vertices in the partitions not appearing in L' are possible. More precisely, let $V_{j_1}, \ldots, V_{j_{k-v_{L'}}}$ be the partitions of G' that correspond to labels of vertices of H_S that do not appear in L'. Each choice of one vertex from each of these partitions gives rise to a different occurrence $H_{S,G'}$ of

H_S in G'. Thus, there are $M_{L'} := \Pi_{x=1}^{k-v_{L'}} |V_{j_x}|$ many such occurrences in G'. Said differently, there are exactly $M_{L'}$ many labelings of H_S that agree with the labeling of L' in G'. Note that $M_{L'}$ only depends on k, on the labeling of L', and on the sizes of different sets V_j of the partition, but not the specific structure of H_S, which makes it easy to compute M in time $O(\log M_{L'}) = O(k \log n)$. Combining (i) and (ii) it follows that each labeling of H_S embedded in $J_{L,L'}$ contributes $2^{\binom{k}{2}-e-1-e_H+e_{L'}} \cdot M_{L'}$ to $c_{S_G[L]}$.

As there are $c_{L'}$ many subgraphs L' in G and for each of them there are $c_{J_{L,L'}}$ many possible labelings of H_S such that H_S contains all edges of L' but none of $L \setminus L'$ we need to subtract

$$c_{L'} \cdot c_{J_{L,L'}} \cdot M_{L'} \cdot 2^{\binom{k}{2}-e-1-(e_H-e_{L'})}$$

from the count $c_{S_G[L]}$.

So, for all $O(2^{k^2})$ labeled subgraphs of L we can compute their contribution to $c_{S_G[L]}$ and subtract out this contribution. This leaves only a count of labelings of H_S that overlap with L exactly. Let $V_{j_1}, \ldots, V_{j_{k-v}}$ be the partitions of G' that correspond to labels of vertices of H_S that do not appear in L and let $M_L := \Pi_{x=1}^{k-v} |V_{j_x}|$. To compute the number of subgraphs L we simply divide this number by $c_{G_{L,L}} \cdot M_L \cdot 2^{\binom{k}{2}-e_H}$.

The total time for this computation is $O(2^{k^2} \cdot k! + 2^{k^2} \cdot k \log n)$. □

Note that if $k = o(\sqrt{\lg(n)})$, then the running time $O((k! + k \log n) \cdot 2^{k^2})$ is sub-polynomial. We now use this lemma to count the labeled copies of H_S in Erdős-Rényi H_S-partite graphs as follows. Specifically, our base case are single vertices (sizes of partitions) and labeled edges (number of edges between two partitions). We can then count all one-edge three-node labeled subgraphs using Lemma B.3. Then we can count all three-node two-edge labeled subgraphs using Lemma B.4. In general given that we have counted all subgraphs with $v-1$ or less vertices and all subgraphs with v vertices and at most $e-1$ edges we use Lemma B.4 until we have counted all subgraphs with v vertices and at most $\binom{v}{2}$ edges. Then we use Lemma B.3 to count subgraphs with $v+1$ vertices and at most $v-2$ edges. We can repeat this procedure to count all labeled subgraphs of H_S, including labeled H_S itself.

Now, let us recall our goal: we want to say that an algorithm $\mathscr{A}$ for counting unlabeled H_S graphs (dynamically/statically) in Erdős-Rényi graphs implies an algorithm $\mathscr{D}$ for counting labeled H_S graphs (dynamically/statically) in H_S-partite Erdős-Rényi graphs. Our lowest level recursive steps work for this (they simply count the number of edges in between two partitions which can be done efficiently dynamically and statically). However, our more involved recursion step, Lemma B.4, requires that we have the unlabeled counts of H_S in all graphs X_G. Statically, this is easy, we simply produce all graphs in X_G in time $O(n^2 2^{\binom{k}{2}})$. Dynamically, we proceed as follows. We maintain two types of data structures: (1) We dynamically maintain the number of edges between any pair of vertex sets in the partition (i.e. the base case data structure from Lemma B.1. (2) Let X_G^* be the set of graphs consisting of each graph G' of X_G and some suitable subgraphs of G' (explained below in the proof of Theorem 3.1). We run an instance of algorithm $\mathscr{A}$ for each graph G' of X_G^*. All such graphs are generated during preprocessing and an instance of $\mathscr{A}$ is initialized on each of them. Whenever there is an update in G, we perform the same update in $\mathscr{A}$ for each graph in X_G^*. Note that this guarantees that at each point in time the graphs on which $\mathscr{A}$ is run are exactly the graphs in X_G^*.

For each update in G we perform k^2 updates in the type-(1) data structures, each taking constant time, and $|X_G^*|$ many updates in the type-(2) data structures, each taking as much time as an update in $\mathscr{A}$. The preprocessing time is $O(n^2 |X_G^*|)$ plus $|X_G^*|$ times the preprocessing time of $\mathscr{A}$.

Now at query time we run the recursive algorithm sketched above and stated formally in the following lemma. During the recursion it requires to know the unlabeled count of subgraphs H_S which is given by the type-(2) data structures and the number of edges between any pair of vertex sets, given by the type-(1) data structures.

LEMMA B.5. *Let H_S have e edges and k vertices. Let $\mathscr{B}$ be an average-case algorithm for counting* unlabeled *copies of H_S that have exactly one vertex per partition in dynamic Erdős-Rényi graphs with $Q(n)$ query time which gives the correct with probability at least $1 - \varepsilon / \left(2^{\binom{k}{2}}\right)$.*

There is an algorithm to determine the number of labeled copies of H_S in Erdős-Rényi dynamic graphs with edge probability $1/2$ with query time $O(2^{\binom{k}{2}} Q(n))$. It will give the correct answer with probability at least $1 - \varepsilon$.

Proof. To count the number of labeled copies of H_S in a H_S-paritie Erdős-Rényi graphs we will make queries into all graphs $G' \in X_G$. We will ask that an instance of $\mathscr{B}$ be run on all $G' \in X_G$. We are not analyzing the preprocessing time here, we do that analysis in Theorem 3.1. Here we will analyze how, given a dynamically updated count of unlabled copies of H_S with one node per partition we can count unlabled copies of H_S.

First we use the base case from Lemma B.1, recall that this has $O(1)$ update time dynamically. Then, as described in the lemmas above we will recurse upwards. Let $[v,e]$ be an abuse of notation for the set of all counts of labeled subgraphs with v vertices and e edges. The base case gives us $[1,0]$ and $[2,1]$. We can produce $[2,0]$ using Lemma B.2. From here if we have $[v,e]$ and $e < \binom{v}{2}$ we will produce $[v,e+1]$ using Lemma B.4. If $e = \binom{v}{2}$ then we will produce $[v,i]$ for all $i < v-2$ using Lemma B.3. Note that if there are no labeled subgraphs with certain $[v,e]$ pairs, then this case is trivial as we have nothing to return. We can recurse upwards using this until we have counts for all labeled subgraphs H_S. This solves the question at hand.

We now analyze the running time. The recursion procedures take $O(2^{k^2} \cdot (Q(n) + 2^{k^2}))$ time per query. We make a query into each graph in X_G and we do computation for each subgraph L of H_S and there are at most 2^{k^2} subgraphs L of H_S.

Now, for probability of correctness. If $\mathscr{B}$ succeeds with probability at least $1 - \varepsilon/2^{\binom{k}{2}}$ then by union bound we give the correct answer with probability at least $1-\varepsilon$. ∎

Note that we never needed to store anything other than the base case values of the number of edges between pairs of vertex sets in the vertexpartitions (which is easy to maintain dynamically, see Lemma B.1). Our recursive calls simply require the information about the counts of copies of H_S in all graphs in X_G, but no history of information other than that. This makes dynamically maintaining the count very easy: We maintain two data structures, namely (1) the number of edges between pairs of partitions. If we keep track of the number of edges between partitions in G then we simply need to use a dynamic algorithm to give counts of copies of unlabeled H_S in all graphs in X_G. This allows us to count the labeled copies of H_S in G.

Now we are able to proof Theorem 3.1. We will re-state the theorem again.

REMINDER OF THEOREM 3.1 *Imagine we are given a dynamically updated graph with random updates, where some edges are marked as not-allowed. The graph will be split into k partitions and edges with 'disallowed' updates must be the complete edge sets between two partitions. These edges will not be randomly updated by the random updates. We use this structure to make our proofs easier in the body of the paper.*

Let H_S be a graph with k nodes. Assume that the problem of counting H_S in a graph with random updates among the allowed edges in an H_S-partite graph requires $2^{\binom{k}{2}+k}U(n)$ time per update and $2^{\binom{k}{2}+k}Q(n)$ time per query as long as at most $O(2^{\binom{k}{2}+k}(n^2+P(n)))$ time is spent preprocessing and gives correct answers to queries with probability $1-\delta$. If the allowed edges are a constant fraction of all edges then the average-case H_S counting problem requires at least $\Omega(U(n))$ time per update or at least $\Omega(Q(n))$ time per query as long as at most $O(P(n))$ time is spent preprocessing. This new algorithm will give correct answers to queries with probability at least $1-\delta 2^{\binom{k}{2}+k}$.

Proof. In this proof we will show that given a fast algorithm $\mathscr{A}$ for counting unlabeled H_S in Erdős-Rényi graphs we can produce a fast algorithm for counting labeled H_S in H_S-partite Erdős-Rényi graphs. Assume $P'(n)$, $U'(n)$, and $Q'(n)$ are the preprocessing, update, and query time of algorithm $\mathscr{A}$. First we will use Theorem B.1 which makes 2^k calls to an algorithm for counting unlabeled H_S to produce an algorithm $\mathscr{B}$ for counting unlabeled H_S with exactly one vertex per partition. To apply these techniques we need not just the counts of unlabeled copies of H_S in graphs X_G, but also these counts in a larger set of graph which we denote as X_G^* which is defined as follows.

First recall that G, and thus also G' the set of vertices V is partitioned into k set $V_1, V_2, \ldots, V_k$. For every subset $T \subset [1,k]$, $T \neq \emptyset$, let G'_T be the subgraph of G' induced by the vertex set $\cup_{i \in T} V_i$. We also need to run $\mathscr{A}$ on all subgraphs G'_T of G with $T \neq \emptyset$. As there are 2^k subgraphs of G', there are $2^{\binom{k}{2}+k}$ graphs on which $\mathscr{A}$ is run. Let X_G^* denote the set of all these graphs. Note that all of them Erdős-Rényi dynamic graphs. For the inclusion-exclusion technique we need to run an instance of $\mathscr{A}$ on every graph of X_G^* which counts unlabeled copies of H_S and use it to count unlabeled H_S which have exactly one vertex per partition. As described before every edge update to G triggers at most $|X_G^*| = 2^{\binom{k}{2}}2^k$ edge updates to maintain the graphs for all $2^{\binom{k}{2}+k}$ instances of $\mathscr{A}$. Note that $2^{\binom{k}{2}+k} < 2^{k^2+k}$.

Now we can use Lemma B.5, which requires (1) a dynamic algorithm for the base cases, which is given in

Lemma B.1, and (2) an instance of algorithm $\mathscr{B}$ for each graph X_G^*. These are exactly the type-(1) and type-(2) data structures described above that can be maintained in time $2^{\binom{k}{2}+k}U'(n)+k^2$ per update with preprocessing time $2^{\binom{k}{2}+k}(n^2+P'(n))+k^2$. Every time there is a query we execute the recursion from lemma B.5 from scratch. This allows us to count the number of labeled copies of H_S in an H_S-partite Erdős-Rényi graph dynamically in time $2^{\binom{k}{2}+k}Q'(n)$ for the following reason: At query time the algorithm of lemma B.5 calls algorithm $\mathscr{B}$ $2^{\binom{k}{2}}$ times, algorithm $\mathscr{B}$ calls $\mathscr{A}$ 2^k times, and each call to $\mathscr{A}$ takes time $Q'(n)$. As the calls to $\mathscr{A}$ dominate the running time, the total time for answering a query is $2^{\binom{k}{2}+k}Q'(n)$. If $\mathscr{A}$ succeeds with probability $1-\delta$ then $\mathscr{B}$ succeeds with probability at least $1-\delta 2^k$ by a union bound. By Lemma B.5 our final algorithm succeeds with probability at least $1-2^{\binom{k}{2}}\varepsilon$ if $\mathscr{B}$ succeeds with probability $1-\varepsilon$. So the success probability of query is at least $1-2^{\binom{k}{2}+k}\delta$. Let us denote the resulting algorithm by $\mathscr{D}$.

So assume that $\mathscr{A}$ has preprocessing time $P'(n)=O(P(n))$, update time $U'(n)=o(U(n))$ and query time $Q'(n)=o(U(n))$. Then $\mathscr{D}$ would take $O(2^{\binom{k}{2}+k}(n^2+P(n)))$ preprocessing time, $o(2^{\binom{k}{2}+k}U(n))$ update time and $o(2^{\binom{k}{2}+k}Q(n))$ query time which is a contradiction, as in our theorem statement we state that $\mathscr{D}$ requires at least $2^{\binom{k}{2}+k}U(n)$ time per update and $2^{\binom{k}{2}+k}Q(n)$ time per query as long as at most $O(2^{\binom{k}{2}+k}(n^2+P(n)))$ time is spent preprocessing. □

Let us quickly summarize what we did in this section. We used a recursive calls to count larger and larger labeled subgraphs of H_S in a graph G using only an unlabeled counter. We then use those counts to count labeled H_S graphs using an unlabeled H_S counter. In the worst-case this reduction is very easy. However, in the average-case we need to maintain the property that the edge sets 'look random'. We solve this problem by making repeated calls to graphs that each individually look indistinguishable from random because

C Total Variation Distance and Algorithms

In this appendix we will explain how you can use TVD to use the success probability of any algorithm on one distribution on a very similar different distribution. We will then bound the difference between two distributions we use in our biased updates proof.

LEMMA C.1. *We are given algorithm $\mathscr{A}$ and two distributions D and D' over inputs to $\mathscr{A}$. Let ε be the TVD from D to D'. If $\mathscr{A}$ has success probability p on inputs drawn from the distribution D then $\mathscr{A}$ has a success probability of at least $p-\varepsilon$ on inputs drawn from D'.*

Proof. Let S_D be the support of D and let $S_{D'}$ be the support of D'. Let $\hat{S}=S_D\cup S_{D'}$. Let F be the subset of $\hat{S}$ on which $\mathscr{A}$ fails.

Now let $f=1-p$, the failure probability of $\mathscr{A}$ on the distribution D. Let q be the success probability of $\mathscr{A}$ on inputs drawn from D'. Let $f'=1-q$ the failure probability of $\mathscr{A}$ on inputs drawn from D'.

Now recall that TVD between D and D' is defined as the sum:

$$\sum_{x\in\hat{S}} |\left(Pr_{y\sim D}[y=x]\right)-\left(Pr_{y\sim D'}[y=x]\right)|.$$

Further note that $f-f'$ is the sum:

$$\sum_{x\in F} \left(Pr_{y\sim D}[y=x]\right)-\left(Pr_{y\sim D'}[y=x]\right).$$

Now note that $F\subseteq\hat{S}$ so:

$$f'-f=\sum_{x\in F}\left(Pr_{y\sim D'}[y=x]\right)-\left(Pr_{y\sim D}[y=x]\right)\leq\sum_{x\in\hat{S}}|\left(Pr_{y\sim D}[y=x]\right)-\left(Pr_{y\sim D'}[y=x]\right)|=\varepsilon.$$

So $f'-f\leq\varepsilon$ which tells us that $p-q\leq\varepsilon$. So the success probability of $\mathscr{A}$ on D' is at least $p-\varepsilon$. □

Now onto bounding the TVD between distributions in our biased updates proof. We start by proving random updates and random vectors are similar. We will use $Bin_\mu[x]$ to refer to the Binomial distribution where a success happens with probability μ and there are x samples.

LEMMA C.2. *Let D be the uniform distribution over n bit Boolean vectors. Let T_p be the distribution of random vectors formed by making $Bin_p[n\lg^2(n)]$ bit flips on uniformly random locations in the vector starting from the all zeros vector.*

When p is constant the TVD of D and $T_{1/2}$ is $2^{-\Omega(\lg^2(n))}$. When p is constant the TVD of D and $T_{1/3}$ is also $2^{-\Omega(\lg^2(n))}$.

Proof. Consider $\vec{u} \sim T_p$. Now consider one given bit in this vector, $\vec{u}[j]$. After f flips call the probability that $\vec{u}[j] = 0$ by q_f and call the TVD for that bit δ_f. Note that $q_f = 1/2 + \delta_f$. Now note that $q_0 = 1$ and $\delta_0 = 1/2$. Next note that the probability that $\vec{u}[j] = 0$ after i flips will be equal to:

$$(\text{probability } j \text{ not flipped in } i^{th} \text{ flip})(q_{i-1}) + (\text{probability } j \text{ flipped in } i^{th} \text{ flip})(1 - q_{i-1})$$

We can write this as

$$q_i = (1 - p/n)(1/2 + \delta_{i-1}) + p/n(1/2 - \delta_{i-1}) = 1/2 + (1 - 2p/n)\delta_{i-1} = 1/2 + \delta_i.$$

So $\delta_{i+1} = (1 - 2p/n)\delta_i = (1 - 2p/n)^{i+1}$. So after $n\lg^2(n)$ flips we have that $\delta_{n\lg^2(n)} \leq e^{-p\lg^2(n)} = 2^{-\Omega(\lg^2(n))}$. ∎

Finally we will address the parities of β_u and β_v from Theorem 2.2.

LEMMA C.3. *Let $x \sim Bin_p[n\lg^2(n)]$ and $y \sim Bin_p[n\lg^2(n)]$ (so x and y are sampled from the same binomial distribution). Let A be the probability distribution over tuples $(x \mod 2, y \mod 2)$.*

Let B be the probability distribution over over tuples $(a \mod 2, b \mod 2)$.

The TVD of A and B when $p = 1/2$ is zero. The TVD of A and B when $p = 1/3$ is $2^{-\Omega(n\lg^2(n))}$.

Proof. For distribution $Bin_p[n\lg^2(n)] \mod 2$ note that we can make a similar argument to Lemma C.2. We start with a parity of zero with probability 1. Let $q_i = 1/2 + \delta_i$ be the probability of a zero for the distribution $Bin_p[i] \mod 2$. Then note that $q_{i+1} = (1-p)q_i + p(1-q_i)$. So $\delta_{i+1} = (2p-1)\delta_i$. When $p = 1/2$ δ_i goes to zero immediately. When $p = 1/3$ we have that $\delta_i = 1/2(-1/3)^i$. So when $f = n\lg^2(n)$ we will have a probability of zero of $q_f = 1/2 + 1/2(-1/3)^{n\lg^2(n)}$. The distribution A will have probability q_f^2 for $(0,0)$, $q_f(1-q_f)$ for $(0,1)$ and $(1,0)$, and finally $(1-q_f)^2$ for $(1,1)$.

For distribution B we will start by analyzing L. Note that L is the same distribution as D (the xor of two random vectors is a random vector). Now note that C is one with probability $1/2$ and zero with probability $1/2$. Each index has a $1/2$ chance of being a 1 iid, when you XOR a random variable that is zero and one each with probability $1/2$ with another such variable the probability is still $1/2$. So the distribution B is simply uniformly $1/4$ probability across all four of these tuples: $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$.

Recall that the definition of TVD is the sum of the absolute value of the difference in probability distribution on all outcomes. So, the TVD of these distributions is the sum of the differences in probability across the four outcomes $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. So the TVD between these is $|q_f^2 - 1/4| + 2|q_f(1-q_f) - 1/4| + |(1-q_f)^2 - 1/4|$. Recall $q_f = 1/2 + 1/2(-1/3)^{n\lg^2(n)}$ so the TVD of A and B is at most $2^{-\Omega(n\lg^2(n))}$. ∎

$1-\delta$. *If the allowed edges are a constant fraction of all edges then the average-case H_S counting problem requires at least $\Omega(U(n))$ time per update or at least $\Omega(Q(n))$ time per query as long as at most $O(P(n))$ time is spent preprocessing. This new algorithm will give correct answers to queries with probability at least* $1-\delta 2^{\binom{k}{2}+k}$.

4 Lower Bounds for Counting 4-Cycles

In this section we will get a lower bound for the update time, $U(n)$ and query time, $Q(n)$ of average-case four cycle counting from biased average-case parity OuMv. These lower bounds will apply for any polynomial preprocessing. More specifically, we show that $nU(n)+Q(n)=\Omega(n^{2-o(1)})$ if the OMv hypothesis holds. Before we show that counting 4-cycles is hard in general graphs where all updates are flips of random edges, we will show that counting 4-cycles is hard on special 4-partite graphs where the edges between two of the parts are complete and all other edges are set by random flips. Given a sequence of $(M_i,\vec{u}_i,\vec{v}_i)$ for $0\leq i\leq n$ of a biased average-case parity OuMv problem with dimension n we construct a graph with $4n$ nodes, n each in A, B, C, and D, and edges $E\subseteq A\times B\ \cup\ B\times C\ \cup\ C\times D\ \cup\ D\times A$. The edges between A and B represent M in the natural way (edge (a_i,b_j) exists iff $M[i][j]=1$), the edges between D and A (resp. B and C) represent $\vec{u}$ (resp. $\vec{v}$) as described next, and there is a complete graph between C and D.

To represent $\vec{u}$ we use the $O(n^2)$ edges between A and D as follows: For any fixed j with $1\leq j\leq n$, let the parity of the number of edges (d_i,a_j) that exist for $1\leq i\leq n$ be equal to $\vec{u}[j]$. The edges in this graph are, of course, randomly updated. Additionally, they start as random. Note that when the edges start as random the parity of the number of (d_i,a_j) edges that exist is odd or even each with probability $1/2$. Further note that if we randomly update an edge between D and A this corresponds to randomly flipping a bit in $\vec{u}$. So, we are representing $\vec{u}$ as an XOR over n^2 edges, $\vec{v}$ is represented in the same way by representing $\vec{v}[j]$ by edges between B and C. We are using an XOR because we want random flips of edges to flip bits in our vector. See Figure 2 for a visual depiction.

For $1\leq i\leq n$ let the vector $\vec{p}_j$ be the vector such that $\vec{p}_j[i]=1$ iff the edge (d_j,a_i) exists in the graph (where d_j is the j^{th} node in D and a_i is defined similarly) and zero otherwise. Define $\vec{q}_j$ similarly as the vector s.t. $\vec{q}_j[i]=1$ iff (c_j,b_i) exists and zero otherwise. Note that we will set these edges such that $\vec{u}=\bigoplus_{j\in[1,n]}\vec{p}_j$ and let $\vec{v}=\bigoplus_{j\in[1,n]}\vec{q}_j$.

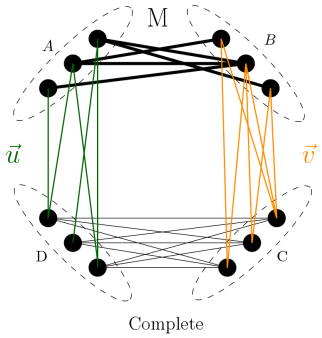

Figure 2: In this graph we depict a 4-partite graph constructed from an OuMv instance.

First let us explain why blowing up the representations of $\vec{u}$ and $\vec{v}$ are necessary. We need n^2 edges to represent M, and the most efficient representation of $\vec{u}$ and $\vec{v}$ could involve only n edges. However, if we make $O(n\lg^2(n))$ updates the expected number of updates that will flip edges representing $\vec{u}$ and $\vec{v}$ in their efficient representation is only $O(\lg^2(n))$ bit flips as the chance of picking an edge representing them is only $\Theta(1/n)$. This is not enough to randomize our vectors. However, if we represent the vectors $\vec{u}$ and $\vec{v}$ as the XOR over n representations of the vectors (e.g. the n vectors $\vec{p}_j$) we expect to make $O(n\lg^2(n))$ random flips in the vectors $\vec{u}$ and $\vec{v}$. Of course, we also expect to flip some bits in M when doing this. So, to prove that 4-cycle counting in our 4-cycle graph is hard when three of the parts are randomly updated we need to rely on Theorem 2.2. This biased model of edge flips captures our situation.

We explain next why the edge set between C and D is complete. Let $\vec{U}=\sum_{j\in[1,n]}\vec{p}_j$ and let $\vec{V}=\sum_{j\in[1,n]}\vec{q}_j$ and note that for each $1\leq i\leq n$, $\vec{u}[i]=\vec{U}[i] \mod 2$ and $\vec{v}[i]=\vec{V}[i] \mod 2$. Let $E_{C,D}$ be the edge set of C,D. For convenience let $(i,j)\in E_{C,D}$ mean there is an edge between c_i and d_j. We want to compute $\vec{u}M\vec{v}$ but when counting the number of

randomizing the order of the updates represented in Δ. Further note that conditioned on the graph going from G to G' these sampled updates are drawn from uniform distribution.

Now consider starting from any graph and making xn^2 random updates to random edges. The resulting graph has a TVD from a Erdős-Rényi graph of at most $n^2(1-1/n^2)^{xn^2} < n^2e^{-x}$. So our sampling procedure above has TVD at most n^2e^{-x}.

Consider running the algorithm $\mathscr{A}$ with xn^2 random updates and one query, if $\mathscr{A}$ succeeds with probability p. This implies a success probability of $1-p-n^2e^{-x}$ for counting cliques in a Erdős-Rényi graph. Consider setting $x = \lg^2(n)$. Now, if $p = 2^{-10}$ we have an algorithm for counting 3-cliques in an Erdős-Rényi graph that succeeds with probability greater than $1-2^{-9}$, which implies a worst-case algorithm with a success probability of at least $2/3$. This algorithm involves the prepossessing of $\mathscr{A}$, $\lg^2(n)n^2$ updates, and one query. So, we have that given an algorithm $\mathscr{A}$ we have a worst-case clique counting algorithm which runs in time $P(n)+Q(n)+n^{2+o(1)}U(n)$. By the 3-clique hypothesis

$$P(n)+Q(n)+n^{2+o(1)}U(n) = \Omega(n^{\omega-o(1)}).$$

This gives us our desired result. □

5.2 Lower Bound for Counting Triangles Through a Queried Point Let $\#\triangle_a$ be the count of the number of triangles that go through the node a. Recall that in our model of average-case algorithms our adversary is allowed to pick when updates and queries are made, but not what those queries or updates are. Recall that random update flips a random edge. Additionally, for this problem of counting triangles through a queried point our queries are of the form $\#\triangle_a$ for a randomly selected node a. We will show that with $n^{2+o(1)}$ updates and $n^{2+o(1)}$ queries we can answer n vector queries on a Mv instance with a n by n matrix and n vectors of length n. This will let us prove that for the counting triangles through a queried point problem we require that $U(n)+Q(n)$ takes at least $n^{1-o(1)}$ time.

In this section we will use the OMv hypothesis. We will create a tripartite graph with node sets A, B, and C. We will represent M as the edges between A and B. Specifically $(a_i,b_j) \in E$ if $M[i][j] = 1$. We will represent $\vec{v}$ as the edges between B and U. Specifically if $\vec{v}[j] = 1$ then an odd number of edges (b_j,u_k) exist and if $\vec{v}[j] = 0$ then an even number of edges (b_j,u_k) exist. We will make two calls to graphs with opposite edge sets for A and U so that we can extract a count as if the edge set was complete. See figure 3 for a visual of this reduction. Note that with this setup the number of triangles (in both graphs) through a_i is equal to $(M\vec{v})[i]$. Now, we aren't allowed to query any specific point. However, after $\Theta(n\lg(n))$ queries for the number of triangles through a point we will get answers for all points with constant probability. So, with $\Theta(n\lg^3(n))$ queries we will get answers for all points with high probability. With the answers though a_i for all $i \in [1,n]$ we can reproduce the whole vector $M\vec{v}$.

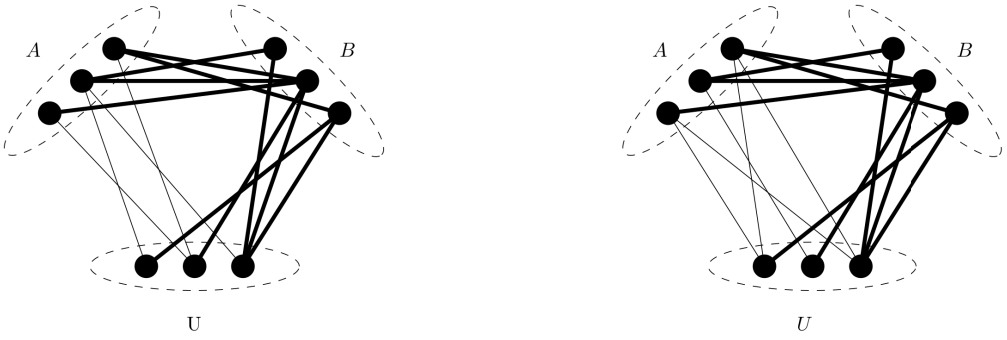

Figure 3: Two graphs with the edge sets flipped between A and U. If we count triangles in both graphs we get a count equal to the count in a single graph where all edges between A and U exist.

Before doing any of that, we will need to demonstrate that $n^2\lg^3(n)$ random updates to a graph produce a graph where each edge appears flipped with probability $1/2$ up to a very small total variation distance.

LEMMA 5.1. *We are given a graph G with n nodes where you allow some subset E of edges where one is picked uniformly at random across all edges to be flipped $n^2\lg^2(n)$ times. The resulting graph G has TVD at most $2^{-\Theta(\lg^2(n))}$*

THEOREM 5.2. *We are given an algorithm $\mathscr{A}$ which runs in graphs with Erdős-Rényi dynamic random graph updates and uniformly random queries across all nodes of $\#\triangle_a$ which succeeds with probability at least $1-\frac{1}{n^2 1680}$ and has preprocessing time $n^{3-\varepsilon}$ for $\varepsilon > 0$. Let $\mathscr{A}$'s update time be $U(n)$ and $\mathscr{A}$'s query time be $Q(n)$. Then if the OMv hypothesis holds $n^2U(n)+n^2Q(n) = \tilde{\Omega}(n^3)$.*

Proof. Consider taking a random sequence of updates that produces a graph as described by Lemma 5.2, lets call the series of updates that produces this graph G. We randomly intersperse updates within the partitions A, B and U. There are $3\binom{n}{2}$ internal edges and $3n^2$ edges between partitions. So, with probability $\binom{n}{2}/(\binom{n}{2}+n^2)$ we update a random edge inside a partition, and the rest of the time we use the updates from G. Call this new series of updates G'. Now, we will maintain a few different versions of G'. We maintain versions with just the nodes in a single partition G'_A, G'_B, G'_U. We also maintain versions with the nodes of two partitions $G'_{AB}, G'_{BU}, G'_{UA}$. We will run $\mathscr{A}$ on all of these graphs. To determine the number of triangles through a node x in G we can simply take the count from G' and subtract the counts from G'_{AB}, G'_{BU}, and G'_{UA} and then finally add the counts from G'_A, G'_B, and G'_U. Note that all of these graphs look like uniformly randomly updated graphs. We can union bound across all 7 of these graphs to get our probability of $1-\frac{1}{n^2 1680}$. □

So, we have our first lower bound in our standard model of average-case dynamic graph updates. Note that small changes to our problem statement have drastically changed how hard this problem is. When we are counting the triangles through a fixed point s the problem is easy. When we are counting triangles through a queried point a this problem is hard. We will now move on to the problem of counting four cycles.

6 Lower Bounds for Path Counting

In this section we will show the hardness of counting st paths of length 5. This is equivalent to counting st paths of length at most 5 (we can count all paths of length $1,2,3$, and 4 with $O(1)$ update and query time so we can add or remove these counts efficiently). We will demonstrate this hardness via reduction from OuMv. This will give us hardness from the OMv hypothesis.

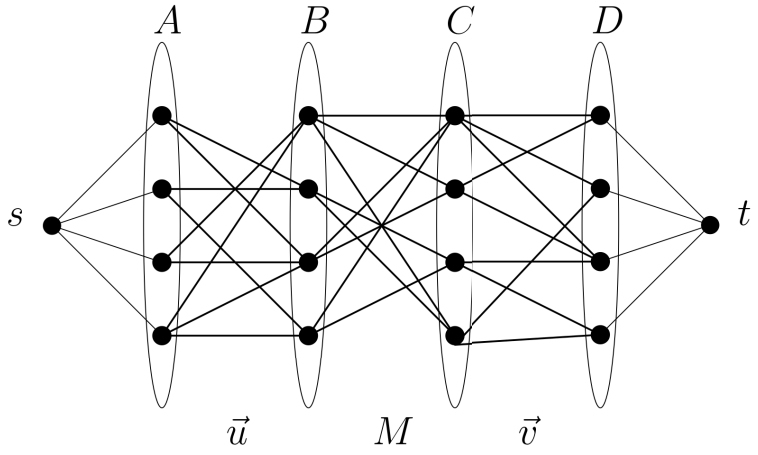

Figure 4: In our reduction the number of paths from s to A to B to C to D to t will be equal to $\vec{u}M\vec{v}$ if sA is complete and Dt is complete. As in our previous reductions we represent $\vec{u}$ and $\vec{v}$ with n^2 edges instead of n edges. The extra edges once again make the random edge flips change the vectors with a high enough proportion.

We will first show that the problem is hard when sA and Dt are complete. We will then show the problem of counting st 5-paths is hard in a st-5-path-partite Erdős-Rényi graph. So, we will show that we can make the edge sets sA and Dt look random instead of complete.

LEMMA 6.1. *Let H_S be a 5-path with two fixed points s and t are the endpoints. Consider the problem of counting the number of H_S graphs in a graph produced by randomly flipping allowed edges in a H_S-partite graph. Call this problem st5-partite st-5-path counting.*

LEMMA B.3. *Let G be a graph with n nodes, m edges and k labeled partitions of vertices $V_1,\ldots,V_k$ (some partitions may be single nodes). Given the count of all labeled subgraphs of H_S in all graphs in X_G with exactly one vertex in each partition of the graph with less than v vertices, we can count all labeled subgraphs with v vertices and at most $v-2$ edges in $\tilde{O}(m)$ time.*

Proof. A subgraph of v vertices and at most $v-2$ edges must be disconnected. Thus, we can use Lemma B.2 to compute the count of the subgraph by multiplying the counts of its connected components. □

Edge recursion. Next we show how to count subgraphs with v vertices and $e+1$ edges, assuming we know the counts for all subgraphs with v vertices and up to e edges [3].

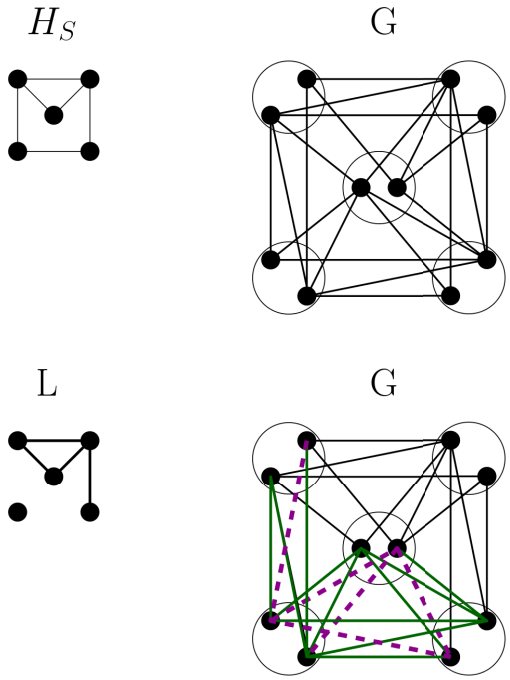

Figure 5: Full lines represent edges in the graph G, so edge sets labeled with 1 e.g. $E^1_{i,j}$. The dashed lines represent the edges in the inverse of the edge sets (i.e. $E^2_{i,j}$). The first graph G represents G with no changes made. The second copy has edges in black which we don't flip when trying to count copies of L. In the second copy green edges represent the edges that exist in the original G (that is $E^1_{i,j}$ edge sets) but which exist in between partitions that we flip when learning the count of L. In Lemma B.4 we take unlabeled counts of subgraphs in all input graphs in X_G where edges in G between partitions that correspond to edges in L are set to $E^1_{i,j}$. In our picture, this is the black (thin line) edges. Then, in our example there are four pairs of vertices in L (which correspond to partitions in G) that do not have an edge between them in L. All 2^4 possible choices of four tuples of edge sets between these partitions will appear in X_G.

LEMMA B.4. *Let G be a graph with the partitions of the vertices into k sets and let L be a labeled subgraph of the labeled graph H_S such that there is at most one node per partition with v vertices and $e+1$ edges for any $v>0$ and $e\geq 0$. Assume we are given the counts of the number of unlabeled subgraphs of H_S which have exactly one vertex in each partition in all graphs in X_G (see Definition B.1). Additionally, assume that we are given the counts of all* labeled *subgraphs of H_S consisting of less than or equal to v vertices with $[0,e]$ edges in G.*

Using both of these counts we can count the number of not-necessarily induced copies of the labeled subgraph L in G in time $O((k!+k\log n)\cdot 2^{k^2})$.

[3] Basically, we will use Lemma 5.7 from [9], but we need to reprove it as [9] requires that there are exactly n vertices in a partition and our graphs might not fulfill this requirement.