Dynamic Boolean Formula Evaluation

Rathish Das
Cheriton School of Computer Science, University of Waterloo, Canada

Andrea Lincoln
University of California, Berkeley, CA, USA
Simons NTT Fellow, Berkeley, CA, USA

Jayson Lynch
Cheriton School of Computer Science, University of Waterloo, Canada

J. Ian Munro
Cheriton School of Computer Science, University of Waterloo, Canada

Abstract

We present a linear space data structure for Dynamic Evaluation of k-CNF Boolean Formulas which achieves $O(m^{1-1/k})$ query and variable update time where m is the number of clauses in the formula and clauses are of size at most a constant k. Our algorithm is additionally able to count the total number of satisfied clauses. We then show how this data structure can be parallelized in the PRAM model to achieve $O(\log m)$ span (i.e. parallel time) and still $O(m^{1-1/k})$ work. This parallel algorithm works in the stronger Binary Fork model.

We then give a series of lower bounds on the problem including an average-case result showing the lower bounds hold even when the updates to the variables are chosen at random. Specifically, a reduction from $k-$Clique shows that dynamically counting the number of satisfied clauses takes time at least $n^{\frac{2\omega-3}{6}\sqrt{2k}-1-o(\sqrt{k})}$, where $2 \leq \omega < 2.38$ is the matrix multiplication constant. We show the Combinatorial k-Clique Hypothesis implies a lower bound of $m^{(1-k^{-1/2})(1-o(1))}$ which suggests our algorithm is close to optimal without involving Matrix Multiplication or new techniques. We next give an average-case reduction to k-clique showing the prior lower bounds hold even when the updates are chosen at random. We use our conditional lower bound to show any Binary Fork algorithm solving these problems requires at least $\Omega(\log m)$ span, which is tight against our algorithm in this model. Finally, we give an unconditional linear space lower bound for Dynamic k-CNF Boolean Formula Evaluation.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Data Structures, SAT, Dynamic Algorithms, Boolean Formulas, Fine-grained Complexity, Parallel Algorithms

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2021.61

Funding *Rathish Das*: Supported by the Canada Research Chairs Programme and NSERC Discovery Grants.
Andrea Lincoln: 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.
Jayson Lynch: Supported by the Canada Research Chairs Programme and NSERC Discovery Grants.
J. Ian Munro: Supported by the Canada Research Chairs Programme and NSERC Discovery Grants.

1 Introduction

Boolean formula evaluation is a fundamental problem in computer science. There are many cases when one might want to evaluate the formula multiple times on related inputs. For example some SAT solving algorithms evaluate all inputs within some small Hamming Ball around certain variable settings, requiring many evaluations of a Boolean formula on very similar inputs [39, 22, 35]. Another example is systems safety monitoring where one needs to check whether certain safety constraints have been violated [32, 18]. One may wonder whether the new values in these settings can be calculated significantly faster than the time it takes to reevaluate the entire formula.

In this paper we define a dynamically updating version of evaluating formulas on Boolean variables in which variables are allowed to change value and the data-structure must maintain whether the formula is still true and, in a harder version, the total number of currently satisfied clauses. In particular, we are given a Boolean expression in k-CNF (where k is taken to be a constant) with n variables and with m clauses. We achieve a sub-linear time update cost for a generalized version of this problem, including counting the number of satisfied clauses, and show a work-efficient parallelization (Section 3). We give lower bounds on the space and time complexity of this problem, including a fine-grained average-case lower bound when all the updates are given at random (Section 4). We find it exciting that these average-case lower bounds are as strong as our worst case lower bounds, and that only a small gap remains between those and our algorithmic upper bound.

The computational complexity of evaluating Boolean formulas has been well studied, showing the problem is in ALOGTIME [17] and the related Boolean Circuit Evaluation problem is complete for NC^1 [16]. There is even a classification of Boolean Formulas under various families of allowed connectives [38]. Since we deal with formulas in k-CNF form, this restriction of the problem would lie in AC0, as it is by definition depth 2. However, when we consider the counting version in which we keep track of the number of clauses currently satisfied, it must be outside AC0 since it can trivially solve the Parity problem [25].

There has been significant work in the fine-grained complexity of dynamic problems which give conditional lower bounds for a variety of problems such as dynamic shortest path, graph connectivity, bipartite matching, max-flow, and graph diameter [37, 36, 3, 28, 20, 29]. Our results on the fine-grained complexity of Dynamic Boolean Formula Evaluation add to this body of knowledge, more importantly it brings the field of average-case fine-grained complexity to dynamic problems and data-structures. Although Alberts-Henzinger give algorithms for some average-case dynamic problems [5], but we're not aware of any work on average-case lower bounds.

Common fine-grained complexity assumptions were used to establish the average-case hardness of evaluating certain types of polynomials over finite fields in [7]. This was used to show the average-case hardness of counting the number of k-cliques in certain easy to sample random graphs [27]. This was then adapted to Erdős-Rényi random graphs in [14] and to counting bicliques in [30]. More recently, hardness for evaluating lower dimensional polynomials has been established from even weaker assumptions than those standard in fine-grained complexity and these have been used to give average-case hardness for a variety of problems such as Edit Distance and Max-Flow [21].

One of the goals of average-case fine-grained complexity is the development of fine-grained cryptography where one hopes to actually prove cryptographic security from fine-grained complexity assumptions and may offer cryptographic protocols that remain secure even if more common cryptographic assumptions turn out to be false. One-way functions and public key cryptography which is unconditionally secure against AC0 circuits was shown in [24]. In [8] fine-grained proof of work was shown from standard fine-grained complexity assumptions. Finally, one-way functions and public key encryption were built based on the average-case complexity of zero-k-clique [34].

2 Preliminaries

We study the problem of dynamically updating queries to a Boolean formula. We define this problem formally and then define the hypotheses from which we get lower bounds for this dynamic problem. Note that in this paper we treat k as a constant.

▶ **Definition 1.** *In* ***Dynamic Boolean Formula Evaluation*** *you are given a fixed k-CNF formula ϕ with m clauses and n variables. Further, the variables are given as an array $\vec{x}$ and initially set to true. The objective is to maintain a data-structure which can answer whether the formula evaluated on the current setting of the variables evaluates to true, subject to updates which flip the value of a single variable. Specifically an update is given as a single index $i \in [0, n-1]$ indicating $\vec{x}[i] = !\vec{x}[i]$.*

In the counting version of the problem, ***Counting Dynamic k-CNF Boolean Formula****, the query instead asks how many clauses are set false. In the* ***parity*** *version of the dynamic formula evaluation problem the queries ask for the parity of the number of clauses set false.*

2.1 Computational Hypotheses

We get computational lower bounds on the dynamic formula evaluation problem from variants of the k-clique problem. In this context ω is the matrix multiplication constant, that is, the smallest real number such that an algorithm exists with running time $O(n^{\omega+o(1)})$ exists for matrix multiplication. It is known that $\omega \in [2, 2.37286]$ [6].

▶ **Definition 2.** *The k-clique hypothesis states that the k-clique problem requires $n^{\omega k/3-o(k)}$ time with randomized or deterministic algorithms [2].*

The next hypothesis has to do with combinatorial algorithms. These are an informally defined set of algorithms which use only “combinatorial methods”, specifically excluding fast matrix multiplication. Although informal, these lower bounds can help inform algorithm design. In our case, the hypothesis will also allow for a better presentation of the reduction and analysis. For a discussion of this hypothesis see [1].

▶ **Definition 3.** *The combinatorial k-clique hypothesis states that for* combinatorial *algorithms the k-clique problem requires $n^{k-o(1)}$ time with randomized or deterministic algorithms [1].*

There exist decision to parity reductions for the clique problem. Goldreich defines $CC_2^{(\ell)}(G)$ as the parity of the count of the number of ℓ cliques in G.

▶ **Theorem 4** (Decision to parity ℓ-clique [26, Theorem 1])**.** *For every integer $\ell \geq 3$, there is a randomized reduction of determining whether a given n-vertex graph contains an ℓ-clique to computing $CC_2^{(\ell)}$ on n-vertex graphs such that the reduction runs in time $O(n^2)$, and makes $\exp(\ell^2)$ queries, and has error probability at most $1/3$.*

2.2 Binary-Forking Model

The Binary-forking model [12, 4, 9, 10, 11, 23], formally defined in [12], is designed to capture the performance of algorithms in the modern multicore shared-memory machines. In this model, the computation starts with a single thread, and as the computation progresses threads are created dynamically and *asynchronously.* The binary-forking model better captures the asynchronous events such as cache misses, varying clock speed, interrupts, etc., than the well-studied and stronger PRAM model [31] where computation progresses synchronously. Since modern multicore architectures employ multiple caches, processor pipelining, branch prediction, hyper-threading, etc., many asynchronous events arise in the system, thus demanding the development of a parallel computation model where computation can proceed asynchronously.

Computations in the binary-forking model can be described as max-degree 3 series-parallel directed acyclic graphs (DAGs). A node in the DAG represents a thread's instruction and each node has at most two children. The root of the DAG is the first instruction of the starting thread. If node u denotes the i-th instruction of thread t and u has only one child v, then v denotes the $(i+1)$-th instruction of the same thread t. If node u has two children v and w, then v represents the $(i+1)$-th instruction of the same thread t and w represents the first instruction of the new forked thread $t^{'}$. Note that whether w represents the 1-st thread $t^{'}$ or the $(i+1)$-th instruction of thread t is arbitrary, we just want to model the overall structure of the computation. The binary-forking model includes "join" instructions to join the forking threads which are modeled as nodes with two incoming edges.

The work of the computation is the number of nodes in the series-parallel DAG and the span of the computation is the length of the longest path in the DAG assuming unbounded resources such as processors and space.

3 Algorithm

In this section we first describe our main algorithm which achieves an update time of $O(m^{1-\frac{1}{k}})$, and then we describe how to parallelize it in the PRAM model with total work $O(m^{1-\frac{1}{k}})$ and span (parallel time) $O(\log m)$. We assume k is a constant.

3.1 Main Algorithm

The high level idea is to take as input a formula ϕ and handle the variables which appear in many clauses (high frequency) and variables that appear in relatively few clauses (low frequency) with different methods. For a low frequency variable we simply update each clause in which it is involved. For high frequency variables we group all clauses with roughly the same structure together and simply track the number of clauses with that structure. Fundamentally, we will try to track for each clause how many variables are set true and how many are set false.

▶ **Theorem 5.** *Given a k-CNF formula ϕ with m clauses and n variables there is an algorithm which takes $O(km)$ preprocessing time and every further update takes $O(m^{1-1/k})$ time to solve the dynamic formula evaluation problem as well as the counting and parity variants of the dynamic formula evaluation problem.*

Proof. Given the formula ϕ with m clauses split the variables into two sets: H for high frequency variables and L for low frequency variables. Variables in H appear in at least λ clauses. Variables in L appear in fewer than λ clauses. Let $\vec{a}$ be the current assignment, without loss of generality assume the starting assignment is the all true assignment ($\vec{a} = \vec{1}$).

Our goal will be to spend $O(\lambda)$ time updating each variable in L when it flips value. We want to build a structure to make updates of variables in H faster than the number of clauses that contain them. First, we will have $k+1$ variables $count_0, \dots, count_k$. The variable $count_i$ will track the count of the number of clauses with i literals set true. Now we will build k arrays $A_1, \dots, A_k$. The A_i array will keep track of clauses with i high frequency variables. An entry $A_i[x_1][x_2]\dots[x_i][j]$ will hold a number. That number will be the count of the number of clauses that contain literals $x_1, \dots, x_i \in H$ and have j variables (not necessarily from $x_1, \dots, x_i$) set true. Note, although we construct the A_i based only on high frequency variables, there may be many clauses with a mix of high and low frequency variables and these arrays help track all of those clauses. So each A_i has a total of $(2|H|)^i \cdot k$ entries, each entry has a number in $[0, m]$. There are $2|H|$ literals for the high-frequency variables (both x and $\bar{x}$), each of the i chosen literals can be any one of those $2|H|$ literals, and for each of these combinations we can have up to k literals true in a clause. Since ordering is equivalent, this is actually an over-count of what is needed, but we work with this number for convenience. For convenience and uniqueness, we give all variables an arbitrary order and only fill out entries where $x_1 < x_2 < \ldots < x_i$. We maintain another helpful Boolean array S that tracks the current settings of all variables: $S[x_i]$ indicates the current value of x_i. Yet another array allows the variables in L to have quick update times. We build the array B of size $2|L|$ where $B[x]$ contains a representation of the at most λ clauses in which the literal $x \in L$ appears. This is $2|L|$ because we have both x and $\bar{x}$. We want these representations to include a marking for which of the variables are high frequency variables.

Preprocessing. We fill in B with the representations of all clauses for low frequency variables. Recall that we mark each clause with which of the variables in the clause are high frequency. Set $S[x_i] = True$ for all x_i. For all the A_i we go through all m clauses in ϕ and update the counts. Consider the following example and a clause $c = (x_1 \vee x_2 \vee \ldots \vee x_k)$ where the x_i are literals. Suppose our initial assignment $\vec{a} = \vec{1}$ sets ℓ literals to true and i of the literals are high frequency variables, $x_1, \ldots, x_i \in H$, then increment $A_i[x_1] \ldots [x_i][\ell]$ by one. We also want to update $count_0, \ldots, count_k$; note that $count_\ell$ is simply a sum of all entries $A_i[*] \ldots [*][\ell]$ for all i. We can also compute these in the preprocessing stage in $O(km + (2|H|)^k \cdot k^2)$ time to initilize the arrays and by simply evaluating each clause. We will later see that this is $O(km)$ with our selection of the size of $|H|$.

Now every entry $A_i[x_1] \ldots [x_i][\ell]$ is a count of how many clauses with the associated literals have ℓ of those literals set to true. We want to maintain this.

Updates for variables in L. If we flip a variable $x_L \in L$ then go to $S[x_i]$ to see the current setting, call it b. Flip it so that now $S[x_L] = \bar{b}$. Next read $B[x_L]$ and $B[\bar{x}_L]$ to get all clauses containing x_L. We will deal with each clause differently depending on how many high frequency variables it has.

If the clause c has no high frequency variables, then read the k entries S for the settings of the variable to determine how many literals were set true before the flip. Let ℓ denote this number. Now, by flipping x_L we either increment or decrement the number of literals in c that are set true (either $+1$ or -1 depending on the original setting and the literal). Suppose the new number of literals set true is $\ell + \Delta$, then we decrement $count_\ell$ and increment $count_{\ell+\Delta}$ (the clause is moving from being an ℓ literal true clause to an $\ell + \Delta$ clause). All of this takes $O(k)$ time.

If the clause c has $i > 0$ high frequency variables, then we have to make changes to A_i as well. First, we read the k entries for variable settings in S to evaluate the clause. Suppose the high frequency variables are $x_1, \ldots, x_i$ and that c had ℓ literals set true before the flip. Suppose that after the flip of x_L the clause c has $\ell + \Delta$ literals set true. Then we decrement $A_i[x_1] \ldots [x_i][\ell]$ and increment $A_i[x_1] \ldots [x_i][\ell + \Delta]$. As before we decrement $count_\ell$ and increment $count_{\ell+\Delta}$. This takes $O(k)$ time. With at most λ clauses, the total time is $O(k\lambda)$.

The variable $count_0$ now holds the correct total for the number of unsatisfied clauses.

Updates for variables in H. Now suppose we are flipping a variable in H, x_H. For a variable in H we cannot look at all of its clauses. However, we can look at all of the relevant entries in A_i arrays. First in $O(1)$ time read our previous setting $b = S(x_H)$ and update $S(x_H) = \bar{b}$.

Now update the A_i arrays. For all $i \in [1,k]$ we will read all the entries that involve our variable x_H. Note that there are $2 \cdot (2|H|)^{i-1} \cdot k$ entries with x_H in each A_i, and so $O((2n)^{k-1})$ entries in total since $|H| < n$ and k is assumed to be constant. Without loss of generality assume that $b = True$ and we are flipping the variable from false to true. Also, for convenience of notation assume that x_H is the first variable in our arbitrary order (if instead x_H appears elsewhere simply access with the variables in order). Then for all $(2n)^{i-1}$ choices of variables $x_2, \ldots, x_i$ we will do the following (from $v = 0$ to $v = k$):

$$A_i[x_H][x_2]\ldots[x_i][v] = \begin{cases} 0 & \text{if } v = k \\ A_i[x_H][x_2]\ldots[x_i][v+1] & \text{if } v < k \end{cases}$$

and (from $v = k$ to $v = 0$)

$$A_i[\bar{x}_H][x_2]\ldots[x_i][v] = \begin{cases} 0 & \text{if } v = 0 \\ A_i[\bar{x}_H][x_2]\ldots[x_i][v-1] & \text{if } v > 0 \end{cases}.$$

The above procedure simply moves the appropriate number of clauses up or down a bucket depending on whether the literal is now true or false. If x_H appears, it used to be true and is now false. So, every clause with x_H now has one fewer literal true, we can move all those clauses into the bucket of the array one below it. If $\bar{x}_H$ appears, it used to be false and is now set true. So we can move all those clauses to the buckets above them. We also have to update our counts. If you move t clauses from $A_i[*]\ldots[*][v]$ to $A_i[*]\ldots[*][v+\Delta]$ then decrement $count_v$ by t and increment $count_{v+\Delta}$ by t.

The variable $count_0$ now holds the correct total for the number of unsatisfied clauses.

Each entry $A_i[*]\ldots[*][v]$ takes $O(1)$ time to update (or $O(\lg(n))$ time in the RAM model instead of word RAM). We need to update a total of $O(k(2|H|)^{k-1})$ entries.

Overall Runtime and Space. The space usage is $m+n+k(2|H|)^k$. The runtime is (assuming k is a constant) $O(\lambda + |H|^{k-1})$. Now, we can bound $|H| < km/\lambda$ and so the runtime is $O(\lambda + (m/\lambda)^{k-1})$. Optimizing, we choose $\lambda = (m/\lambda)^{k-1}$, and get $\lambda = m^{(k-1)/k} = m^{1-1/k}$. For a total claimed runtime of $O(m^{1-1/k})$. ◀

This algorithm gives a non-trivial improvement over the naive algorithm which requires $\Theta(m)$ time updates. We will discuss lower bounds in Section 4, where we are able to show this runtime is close to optimal if the k-clique hypothesis is true (for example a runtime of $O(n)$ are ruled out).

3.2 A Parallel Algorithm

We now show how to parallelize the algorithm of Section 3.1 in the PRAM CREW (concurrent read exclusive write) model [31]. Our results also hold in a more restrictive parallel model called the binary-forking model [12].

▶ **Theorem 6.** *Given a k-CNF formula ϕ with m clauses and n variables there is a parallel algorithm (in PRAM CREW model and the binary-forking model) which for each variable-update performs $O(m^{1-\frac{1}{k}})$ work in expectation and achieves $O(\log m)$ span (parallel runtime) with high probability (in the number of clauses) to solve the dynamic formula evaluation problem.*

Proof. We use the same arrays in the parallel algorithm as in the serial algorithm to achieve the work and span bounds.

Updates for low frequency variables. There are at most λ clauses where low frequency variable x_L might appear. For each clause c where x_L appears, we do the following. Without loss of generality assume that clause c has i high-frequency variables $x_1, x_2, \ldots, x_i$ and has ℓ true literals before flipping x_L. Let $\ell + \Delta$ be the number of true literals in c after the flip.

We update $A_i[x_1]\ldots[x_i][\ell], A_i[x_1]\ldots[x_i][\ell+\Delta]$, count_ℓ, and $\text{count}_{\ell+\Delta}$ for each clause c where x_L appears.

It might happen that multiple clauses trigger updates to the same entry A_i or count_ℓ, since multiple clauses may have the same set of high-frequency variables and the same number of true literals. When multiple updates try to edit the same entry in parallel, write-write conflicts (race conditions) arise which lead to incorrect answers. We resolve race condition using $O(\lambda)$ extra space as follows.

Since x_L is a low-frequency variable, x_L could appear in λ clauses at most. Suppose that each $A_i[x_1]\ldots[x_i][\ell]$ has an ID. We allocate two arrays each of size $O(\lambda)$. One array is used to compute the count_ℓs and the other array is used to compute the A_is. For each of the λ clauses, we allocate a designated $O(1)$ space in each array. Recall that count_ℓ denotes the number of clauses with ℓ literals set true. We would like to update count_ℓ after flipping variable x_L.

We now describe how we decrement count_ℓ and increment $count_{\ell+\Delta}$ for each clause c. Each c computes corresponding $count_\ell$ and $count_{\ell+\Delta}$ (as in the serial version) in $O(k)$ time (k is a constant here) and write them in the designated location in the array allocated to compute count_ℓ (see Figure 1 in the Appendix A). We perform *semisort* [13] (i.e. collect equal values in groups) on the array so that all the count_i comes before all the count_j where $i < j$. The randomized semisort algorithm [13] performs $O(\lambda)$ work in expectation and takes $O(\log \lambda)$ parallel time with high probability. After that we do a binary reduction to get the correct count_is. We do this binary reduction in $O(\log \lambda)$ parallel time. Note that we avoid race conditions by updating count_i in different locations.

Similar analysis holds to calculate the A_is where we store the the value of the A_is according to their IDs. Hence, a low frequency is updated in $O(\log \lambda)$ parallel time and uses $O(\lambda)$ extra space.

Updates for high frequency variables. We do not parse the clauses where the high frequency variable occurs. Instead, we update the corresponding A_is and count_is as follows.

(from $v = 0$ to $v = k$):

$$A_i[x_H][x_2]\ldots[x_i][v] = \begin{cases} 0 & \text{if } v = k \\ A_i[x_H][x_2]\ldots[x_i][v+1] & \text{if } v < k \end{cases}$$

and (from $v = k$ to $v = 0$)

$$A_i[\bar{x}_H][x_2]\ldots[x_i][v] = \begin{cases} 0 & \text{if } v = 0 \\ A_i[\bar{x}_H][x_2]\ldots[x_i][v-1] & \text{if } v > 0 \end{cases}.$$

In short, if x_H appears it used to be true and is now false. So, every clause with x_H now has one fewer literal true, we can move all those clauses into the bucket of the array one below it. If $\bar{x}_H$ appears it used to be false and is now set true. So we can move all those clauses to the buckets above them.

We can do all these updates in parallel in $O(1)$ parallel time. We update the count_ℓ after we update the A_is. We update $O(k \cdot 2|H|^{k-1})$ entries of A_i where x_H is present. If we move t clauses from $A_i[*]\ldots[*][v]$ to $A_i[*]\ldots[*][v+\Delta]$ then we decrement $count_v$ by t and increment $count_{v+\Delta}$ by t. As multiple A_i entries may update the same count_i entry, we use an array of $O(k \cdot 2|H|^{k-1})$ extra space to avoid the race condition for parallel updates. Like before, we semisort the array based on the v of $A_i[*]\dots[*][v]$. Then we perform a parallel sum of the the entries with the same v and then add/subtract this sum to count_i. Combining all these the span to update a high frequency variable is $O(\log |H|) = O(\log n)$.

As we set $\lambda = m^{1-\frac{1}{k}}$, the span to update a low frequency variable is $O(\log \lambda) = O(\log m)$ and a high frequency variable is $O(\log n)$. Since we do not increase the work asymptotically, the work in this parallel algorithm remains the same as in the serial version. Hence, total work performed is $O(\lambda) = O(m^{1-\frac{1}{k}})$ in expectation. The span is $O(\log m)$ w.h.p. due to semisort. ◀

We could replace the randomized semisort algorithm with a deterministic sorting algorithm. Cole-Ramachandran's [19] deterministic sorting algorithm performs $O(x \log x)$ work and takes $O(\log x \log\log x)$ span to sort x items in the binary-forking model (in PRAM CREW model also) giving the following theorem.

▶ **Theorem 7.** *Given a k-CNF formula ϕ with m clauses and n variables there is a deterministic parallel algorithm (in PRAM CREW model and the binary-forking model) which for each variable update performs $O(m^{1-\frac{1}{k}} \log m)$ work and achieves $O(\log m \log\log m)$ span (parallel running time) to solve the dynamic formula evaluation problem.*

4 Lower Bounds for Dynamic Formula Evaluation

In this section we present lower bounds for the dynamic formula evaluation problem on k-CNF formulas. First, we present an unconditional linear lower bound on the space of the data-structure. Next, we present a series of conditional lower bounds on the running times of preprocessing, updates, and queries. These are based on the k-Clique Hypothesis. This culminates in an average-case lower bound for random updates to the variables based on counting k-cliques in Erdős-Rényi graphs. Finally, we present a conditional lower bound on the span of the algorithm in the binary forking model. Recall that we are treating k as a constant in this section.

4.1 Linear Space Lower Bound

We show a space lower bound linear in the number of variables by a reduction from INDEX [33]. Details can be found in the Appendix.

▶ **Theorem 8.** *Every randomized algorithm for dynamic-formula-evaluation which correctly decides that the formula for each variable flip (with n variables, $O(n)$ clauses, and clause size at least 2) is satisfied or not with probability strictly larger than $1/2$, uses $\Omega(n)$ space in the worst case.*

4.2 The k-Clique Hypotheses and a General Reduction

We give these lower bounds from hypotheses related to the ℓ-clique problem. We will use k for the k-CNF Dynamic Counting Boolean Formula problem we reduce to. First, let us present a generic reduction from all ℓ-clique instances to a dynamic formula evaluation formula ϕ on $n = |V|$ variables, where each given instance of ℓ-clique will correspond to a single setting of the variables X.

▶ **Lemma 9.** *Consider a graph G, where we have not yet decided what edges exist or don't. There are $|V|$ vertices and $\binom{|V|}{2}$ potential edges. There exists a formula ϕ_ℓ with $n = \binom{|V|}{2}$ variables, width $k = \binom{\ell}{2}$ clauses, and a total of $m = \binom{|V|}{\ell}$ clauses such that when the n variables X are set with a 1 if the edge exists in G and 0 if the edge doesn't exist then the number of false clauses in $\phi_\ell(X)$ corresponds to the number of k cliques in G.*

Next we want a useful lemma which will help us bound the preprocessing time. We will use many smaller instances of the clique problem so that our number of update calls to our data-structure is large compared to its initial size. Variations on this lemma are folklore, but a proof is provided for completeness.

▶ **Lemma 10.** *Assume (existence/counting/parity) (combinatorial) ℓ-clique requires $|V|^{c\ell(1-o(1))}$ time for some constant c. Let $T(\cdot)$ be the time to solve a single instance when given a list of instances. Then giving correct answers to g^ℓ instances each of size $|U| = \ell|V|/g$ takes $g^\ell T(|U|)$ time which is at least $|V|^{c\ell(1-o(1))}$ time.*

Consider value $0 < d < c \leq 1$ where we insist that $(\ell|V|/g)^\ell = \Theta(|V|^{d\ell})$ then $T(|U|)$ requires at least $|U|^{(\frac{c-1+d}{d})\ell(1-o(1))}$ time.

Proof. Partition the vertex set into $V_1, \ldots, V_g$ each partition of size $|V|/g$. Now we form g^ℓ instances each of size at most $|U| = \ell|V|/g$. These instances are all possible choices of at most ℓ of our vertex sets merged together. Using inclusion exclusion we can count the number of cliques in the original graph using calls to these problems. This inclusion and exclusion also allows parity to solve parity. We will find a clique in these instances if and only if there is at least one clique in the original graph.

Given this reduction we can solve a (existence/counting/parity) (combinatorial) ℓ-clique problem with answers to g^ℓ instances each of size $|U| = \ell|V|/g$, giving the first statement.

For the second statement note that $g = \Theta(|V|^{1-d})$. Then note that $|U| = \Theta(|V|^d)$. Now we can say that $T(|U|)$ must be at least $|V|^{(c-1+d)\ell(1-o(1))}$ time. So, in terms of $|U|$, $T(|U|)$ must be at least

$$|U|^{(\frac{c-1+d}{d})\ell(1-o(1))}.$$ ◀

Note that when $c = 1$ you get no loss in efficiency regardless of the constant $d < 1$ you pick. This case of $c = 1$ happens for combinatorial ℓ-clique. It is why we use it as an example. It simplifies the reduction to not worry about the loss that comes from a large d.

4.3 From the Combinatorial k-Clique Hypothesis

Using the reduction in the prior section, we give a lower bound from the combinatorial ℓ-clique hypothesis. Note how we choose the value of d for the reduction from Lemma 10. We want d to be small so that our preprocessing time is small. This is what will cause inefficiency in the future reductions when $c \neq 1$; we loose efficiency in the reduction when d is small.

▶ **Theorem 11.** *Assume ℓ is a constant. If the combinatorial ℓ-clique hypothesis is true then any combinatorial algorithm $\mathcal{A}$ for the Dynamic Counting Boolean Evaluation problem, $U(n, m, k)$, with polynomial preprocessing time $P(n, m, k) = poly(n) \cdot poly(m) \cdot poly(k)$ requires $m^{(1-k^{-1/2})(1-o(1))}$ time. We can also state the lower bound in terms of the number of variables n, in which case $U(n, m, k)$ requires at least $n^{(1-o(1))(\sqrt{2k}-2)/2}$ time.*

4.4 From the k-Clique Hypothesis

We will now use the same reduction style as the previous section. However, now c from Lemma 10 will be $c = \omega/3$ instead of 1. Here ω is the matrix multiplication exponent constant. The analysis proceeds similarly to that of Section 4.3. Details are in the Appendix.

▶ **Theorem 12.** *Assume ℓ is a constant. If the ℓ-clique hypothesis is true then any algorithm $\mathcal{A}$ for the Dynamic Counting Boolean Evaluation problem, $U(n, m, k)$, with polynomial preprocessing time $P(n, m, k) = O(m)$ requires $m^{\left(\frac{2\omega-3}{3}-k^{-1/2}\right)(1-o(1))}$ time. We can also state the lower bound in terms of the number of variables n, in which case $U(n, m, k)$ requires at least $n^{\frac{2\omega-3}{6}\sqrt{2k}-1-o(\sqrt{k})}$ time.*

4.5 Random Input Implications

Our goal is to show that with a worst-case fixed formula, specifically our k-clique formula, it is still hard to evaluate updates to the formula when the *updates are random.* We will show lower bounds against algorithms which always correctly implement updates, but, are allowed error when responding to queries. The queries this dynamic formula evaluation algorithm must support are queries on the number of falsified clauses.

For this result we will use a theorem from [26][1]. At a high level, they show that the counting/parity l-clique problem requires just as long over the uniform average-case distribution as it does in the worst-case when l is a constant. First we will define the model of average-case inputs we are considering.

▶ **Definition 13.** *Let Average-Case counting dynamic formula evaluation (AC#DFE) be the problem where you are given a worst-case formula to evaluate but random (so average-case) updates on the variables. More specifically, you are given a worst-case formula ϕ, and a series of updates where each update flips the assignment of a uniformly random chosen variable. A data structure D for AC#DFE with error probability ϵ takes as input a worst-case ϕ and after each random update answers the count of the number of false clauses in the formula with probability $1-\epsilon$.*

▶ **Definition 14** (Erdős-Rényi graphs)**.** *Create a graph with $|V|$ vertices and no edges to start with. Now, for every pair of vertices in the graph iid include an edge between them with probability $1/2$. Thus, every potential edge in the graph appears with probability $1/2$ independently from all other edges.*

Now we can describe one theorem from [26]. First we start by using a piece of their notation. Define $CC_2^{(\ell)}(G)$ as the parity of the count of the number of ℓ cliques in G.

▶ **Theorem 15** (Average-Case ℓ-clique [26, Theorem 2])**.** *For every integer $\ell \geq 3$, there is a randomized reduction of computing $CC_2^{(\ell)}$ on the worst-case n-vertex graph to correctly computing $CC_2^{(\ell)}$ on at least a $1 - \exp(-\ell^2)$ fraction of the n-vertex graphs such that the reduction runs in time $O(n^2)$, makes $\exp(\ell^2)$ queries, and has error probability at most $1/3$.*

So, up to sub-polynomial factors, average-case counting and parity ℓ-clique are just as hard on average as worst-case parity ℓ-clique. Now, we can use these reductions to show lower bounds for our problem with random inputs. First, we will show that if you start

[1] For an alternate presentation of the parity proof of this result see [15]. The formulation from [26] is easier to build on in this case.

from a graph and make $\binom{n}{2}\lg^2(n)$ random edge flips the resulting graph is drawn from a distribution that has total variation distance less than $2^{-\Omega(\lg^2(n))}$ from an Erdős-Rényi graph. Note that, on an intuitive level, this means that when we sample from our edge flipping distribution we "look like" we are sampling an Erdős-Rényi graph a $1-2^{-\Omega(\lg^2(n))}$ fraction of the time. Since our reduction represents edges as variables, flipping a random variable is like flipping an edge in the original graph, and we want to argue that after not too many flips we have a random looking graph.

▶ **Lemma 16.** *Given a graph G let the procedure F be the process of selecting a uniformly random pair of vertices (u, v) and either deleting edge (u, v) it if it exists or inserting (u, v) if it doesn't exist.*

If we run the procedure F $\Theta\left(\binom{n}{2}\lg^2(n)\right)$ times on any graph G the resulting distribution over graphs has total variation distance at most $n^{2-\Omega(\lg(n))}$ from the distribution over Erdős-Rényi graphs.

The high level idea for the reduction is to take the worst-case reduction from the prior section and at the last step add the randomized reduction. Specifically, in the worst case we show that solving a list of instances $G_1, \ldots, G_x$ is still hard. Now, for the average-case we will use the [26] reduction to turn each G_i into $\exp(\ell^2)$ queries on random graphs, call these $G_i^{(1)}, \ldots, G_i^{(\exp(\ell^2))}$. Note that these graphs each look random, but have correlations. So, we can't ask the same AC#DFE data structure about two of these. However, we can spin up $\exp \ell^2$ instances of data structures for AC#DFE: $D_1, \ldots, D_{\exp(\ell^2)}$. We will, implicitly, give the data structure D_j graphs $G_1^{(j)}, \ldots, G_x^{(j)}$. These will be totally uncorrelated (each are randomized under different random bits). This will correspond to random updates on variables (which in our worst-case reduction co respond to edges in the graph).

We will show in this next theorem that even with random updates, the same lower bounds hold (up to factors in time of 2^{2k}, which if k is a constant is simply a constant).

▶ **Theorem 17.** *Let k be a constant. Say there is a data structure D that can solve the AC#DFE problem on worst-case k-CNF formulas ϕ with error $\epsilon < \exp(-2k)/3$ with $P_D(n, m, k)$ pre-processing time and $U_D(n, m, k)$ time per query and update. We will note the lower bounds for combinatorial D from the combinatorial ℓ-clique hypothesis. We will also note the lower bound for general D from the ℓ-clique hypothesis.*

- *If D is a combinatorial data structure, the combinatorial ℓ-clique hypothesis is true, and $P_D(n, m, k) = poly(n) \cdot poly(m) \cdot poly(k)$, then $U_D(n, m, k)$ requires $m^{(1-k^{-1/2})(1-o(1))}$ time. We can also state the lower bound in terms of the number of variables n, in which case $U_D(n, m, k)$ requires at least $n^{(1-o(1))(\sqrt{2k}-2)/2}$ time.*
- *If the ℓ-clique hypothesis is true and $P_D(n, m, k) = O(m)$ then $U_D(n, m, k)$ requires $m^{\left(\frac{2\omega-3}{3}-k^{-1/2}\right)(1-o(1))}$ time. We can also state the lower bound in terms of the number of variables n, in which case $U_D(n, m, k)$ requires at least $n^{\frac{2\omega-3}{6}\sqrt{2k}-1-o(\sqrt{k})}$ time.*

Proof. First, let us describe the general reduction regardless of the hypothesis we start with. We take the list of $|V|^{\ell(1-d)}$ sub-problems $G_1, \ldots, G_{|V|^{\ell(1-d)}}$ and on each instance individually we run the reduction from Theorem 15. So we take each problem, G_i, of size $|V|^d$ and use Theorem 15 to make a list of $s = \exp(\ell^2)$ queries $G_i^{(1)}, \ldots, G_i^{(s)}$. Now, we spin up s different instances of D, call them $D_1, \ldots, D_s$. Consider sampling the updates from $G_i^{(j)}$ to $G_{i+1}^{(j)}$ uniformly at random from all random series of updates of length $s = \Theta(n^2 \lg^2(n))$ that flip edges to $G_{i+1}^{(j)}$. Because every variable corresponds to an edge in the graph flipping a variable and flipping an edge is a one to one relationship. So, after this series of flips that look uniformly random up to a TVD of $n^{\Omega(\lg(n))}$ (by Lemma 16) we get the answer to the number of ℓ-cliques in $G_{i+1}^{(j)}$! By the claims of the Theorem statement each D_j has an error rate of

$\epsilon < \exp(-2k)/3$ on the random updates of the average-case distribution of AC#DFE our TVD tells us that the error rate of D_j on this "nearly random" set of updates is at most $\epsilon + n^{\Omega(\lg(n))}$. Now note that if each instance $D_1, \dots, D_s$ gives us the correct answer we can correctly answer our query on all G_i by Theorem 15. So, we get a correct answer on the worst-case G_i with probability $2/3 - \exp\left(\ell^2\right) n^{-\Omega(\lg(n))}$, note this is large enough that this procedure can be repeated to boost this probability. Note this means we get the same lower bounds as we did in the worst-case, but with an overhead of $\exp\left(\ell^2\right)$ which is a constant from our perspective.

First we discuss the combinatorial k-clique case. Start by taking the reduction from Theorem 11. So if D is a combinatorial data structure, the combinatorial ℓ-clique hypothesis is true, and $P_D(n,m,k) = \mathit{poly}(n) \cdot \mathit{poly}(m) \cdot \mathit{poly}(k)$, then $U_D(n,m,k)$ requires $m^{(1-k^{-1/2})(1-o(1))}$ time. We can also state the lower bound in terms of the number of variables n, in which case $U_D(n,m,k)$ requires at least $n^{(1-o(1))(\sqrt{2k}-2)/2}$ time.

Finally, we will address the ℓ-clique hypothesis. As before we will get the same lower bounds, up to a factor of $\exp(\ell^2)$ which is a constant from our perspective. So if the ℓ-clique hypothesis is true and $P_D(n,m,k) = O(m)$ then $U_D(n,m,k)$ requires $m^{\left(\frac{2\omega-3}{3}-k^{-1/2}\right)(1-o(1))}$ time. We can also state the lower bound in terms of the number of variables n, in which case $U_D(n,m,k)$ requires at least $n^{\frac{2\omega-3}{6}\sqrt{2k}-1-o(\sqrt{k})}$ time. ◀

4.6 Span Lower Bound in the Binary-Forking Model

In the Appendix we prove an $\Omega(\log n)$ span lower bound in the binary forking model. This lower bound only requires that the work of the algorithm is $\Omega(n^c)$ for some constant c, and is thus a weaker condition than the k-Clique Hypothesis.

▶ **Theorem 18.** *The span per update for dynamic boolean formula evaluation is $\Omega(\log n)$ in the binary forking model.*

5 Conclusion and Open Problems

Defining Dynamic Boolean Formula Evaluation and giving a data-structure with sub-linear update time is an important first step in characterizing the complexity of this problem. Our data-structure is simple enough and provides a large enough benefit for small k that we hope it will inspire practical implementations. On the theoretical side, although we give upper and lower bounds that show roughly what the complexity of this problem should be, there is still a significant gap between them and plenty of room to generalize both types of results.

With a more careful analysis both our algorithms and lower bounds should hold for some small super-constant k. Improving this analysis may be of interest, but more importantly finding both algorithms and lower bounds that work for a greater range of k, especially when $k = m^c$ for some constant c.

Closing the gap between our lower and upper bounds is one obvious open question. The conditional lower bound giving $\frac{1}{\sqrt{k}}$ comes directly cliques containing l^2 edges. Thus perhaps one should be trying to use graph problems with sparse structures rather than dense ones. However, one needs to be careful because there is only a single k-clique on k vertices whereas there can be many isomorphic sparse structures. The larger issues comes from the factor of the matrix multiplication constant in our lower bound. We see this disappear with the combinatorial l-clique conjecture which suggests either 1) Dynamic Boolean Formula Evaluation can be solved much faster, but doing so will likely require linear-algebraic techniques or some other fundamentally different approach, 2) we need a slightly more clever reduction. A common solution to strengthening "combinatorial" fine-grained assumptions is to move to a weighted version of the problem. However, the OR of variables is symmetric under permutation and thus does not appear to be something that supports any sort of arithmetic at the clause level. Further, our algorithms fundamentally require this symmetry and will fail when trying to generalize in this direction.

Further, it would be interesting to have conditional lower bounds for the pure Dynamic Boolean Formula problem rather than the counting version which seems much stronger. Both giving lower bounds for more restricted versions of the problem and generalizing the algorithm to handle more general formulas seems of interest. Generalizing the depth of the formula (no longer requiring CNF form) is another obvious direction.

We believe that a $\log(n)$-span lower bound should hold in a stronger model such as CREW PRAM. Both improved lower bounds and whether there are better algorithms in stronger concurrent data-structures models remains an interesting question to explore. Depth 2 Boolean formulas being in AC0 strongly suggests at least this special case should be able to be parallelized more efficiently, perhaps at the cost of work-efficiency.

One obvious problem to consider is a version of Dynamic Boolean Formula Evaluation in which the formula itself can be altered, instead of or in addition to the values of the variables. These problems are equivalent in the DynamicSAT case, but looking for sub-linear update cost we have much less leeway in the efficiency of this reduction. We believe minor alterations to our current algorithm will allow it to handle the insertion and deletion of literals and clauses with the same amortized worst-case cost by simply rebuilding the table whenever variables pass some threshold for their frequency of appearing in clauses. However, de-amortizing this does not seem at all obvious.

References

1 Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 192–203. IEEE Computer Society, 2017. doi:10.1109/FOCS.2017.26.

2 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is valiant's parser. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 98–117. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.16.

3 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 434–443. IEEE, 2014.

4 Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The data locality of work stealing. In *ACM symposium on Parallel algorithms and architectures*, pages 1–12, 2000.

5 David Alberts and Monika Rauch Henzinger. Average case analysis of dynamic graph algorithms. In Kenneth L. Clarkson, editor, *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA*, pages 312–321. ACM/SIAM, 1995. URL: http://dl.acm.org/citation.cfm?id=313651.313712.

6 Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. *CoRR*, abs/2010.05846, 2020. arXiv:2010.05846.

7 Marshall Ball, Alon Rosen, Manuel Sabin, and Prashant Nalini Vasudevan. Average-case fine-grained hardness. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 483–496, 2017.

8 Marshall Ball, Alon Rosen, Manuel Sabin, and Prashant Nalini Vasudevan. Proofs of work from worst-case assumptions. In *Annual International Cryptology Conference*, pages 789–819. Springer, 2018.

9 Naama Ben-David, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 145–156, 2016.

10 Guy E Blelloch, Rezaul Alam Chowdhury, Phillip B Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA*, volume 8, pages 501–510. Citeseer, 2008.

11 Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *ACM symposium on Parallelism in algorithms and architectures*, pages 355–366, 2011.

12 Guy E Blelloch, Jeremy T Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 89–102, 2020.

13 Guy E Blelloch, Jeremy T Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 89–102, 2020.

14 Enric Boix-Adserà, Matthew Brennan, and Guy Bresler. The average-case complexity of counting cliques in erdős-rényi hypergraphs. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1256–1280. IEEE, 2019.

15 Enric Boix-Adserà, Matthew Brennan, and Guy Bresler. The average-case complexity of counting cliques in erdős-rényi hypergraphs. In David Zuckerman, editor, *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.

16 S Buss, S Cook, Arvind Gupta, and Vijaya Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM Journal on Computing*, 21(4):755–780, 1992.

17 Samuel R Buss. The boolean formula value problem is in alogtime. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 123–131, 1987.

18 Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2):108–127, 2003.

19 Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. *ACM Transactions on Parallel Computing (TOPC)*, 3(4):1–31, 2017.

20 Søren Dahlgaard. On the hardness of partially dynamic graph problems and connections to diameter. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

21 Mina Dalirrooyfard, Andrea Lincoln, and Virginia Vassilevska Williams. New techniques for proving fine-grained average-case hardness. *arXiv preprint*, 2020. `arXiv:2008.06591`.

22 Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon M. Kleinberg, Christos H. Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. *Theor. Comput. Sci.*, 289(1):69–83, 2002. `doi:10.1016/S0304-3975(01)00174-8`.

23 Rathish Das, Shih-Yu Tsai, Sharmila Duppala, Jayson Lynch, Esther M Arkin, Rezaul Chowdhury, Joseph SB Mitchell, and Steven Skiena. Data races and the discrete resource-time tradeoff problem with resource reuse over paths. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 359–368, 2019.

24 Akshay Degwekar, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Fine-grained cryptography. In *Annual International Cryptology Conference*, pages 533–562. Springer, 2016.

25 Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical systems theory*, 17(1):13–27, 1984.

26 Oded Goldreich. On counting t-cliques mod 2. *Electron. Colloquium Comput. Complex.*, 27:104, 2020. URL: `https://eccc.weizmann.ac.il/report/2020/104`.

27 Oded Goldreich and Guy Rothblum. Counting t-cliques: Worst-case to average-case reductions and direct interactive proof systems. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 77–88. IEEE, 2018.

28 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 symposium on Theory of computing*, pages 21–30, 2015.

29 Monika Henzinger, Andrea Lincoln, Stefan Neumann, and Virginia Vassilevska Williams. Conditional hardness for sensitivity problems. In *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

30 Shuichi Hirahara and Nobutaka Shimizu. Nearly optimal average-case complexity of counting bicliques under seth. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2346–2365. SIAM, 2021.

31 J. JaJa. *An Introduction to Parallel Algorithms*. Addison Wesley, 1997. URL: `https://books.google.com/books?id=9BpYtwAACAAJ`.

32 Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-mac: a run-time assurance tool for java programs. *Electronic Notes in Theoretical Computer Science*, 55(2):218–235, 2001.

33 Ilan Kremer, Noam Nisan, and Dana Ron. On randomized one-round communication complexity. *Computational Complexity*, 8(1):21–49, 1999.

34 Rio LaVigne, Andrea Lincoln, and Virginia Vassilevska Williams. Public-key cryptography in the fine-grained setting. In *Annual International Cryptology Conference*, pages 605–635. Springer, 2019.

35 Andrea Lincoln and Adam Yedidia. Faster random k-cnf satisfiability. *arXiv preprint*, 2019. `arXiv:1903.10618`.

36 Mihai Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 603–610, 2010.

37 Liam Roditty and Uri Zwick. On dynamic shortest paths problems. In *European Symposium on Algorithms*, pages 580–591. Springer, 2004.

38 Henning Schnoor. The complexity of the boolean formula value problem. Technical report, Technical report, Theoretical Computer Science, University of Hannover, 2005.

39 Uwe Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 410–414, 1999. `doi:10.1109/SFFCS.1999.814612`.

A Parallel Algorithm Figure

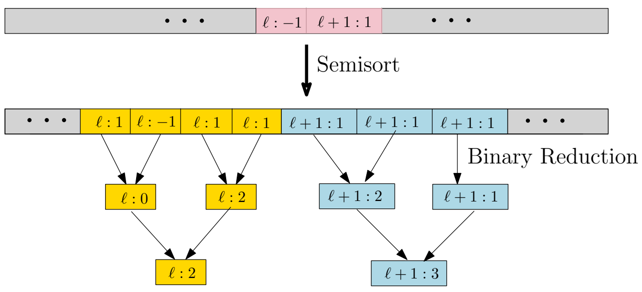

Figure 1 Clause c writes whether count_ℓ and $\text{count}_{\ell+1}$ increment or decrement in their designated locations (in color pink). We then semisort so that all the updates corresponding to count_ℓ come in adjacent locations (in color yellow for count_ℓ). After that we do a binary reduction to get the net update to count_ℓ.

B Lower Bounds Omitted Proofs

B.1 Proof of Linear Space Lower Bound

▶ **Theorem 19.** *Every randomized algorithm for dynamic-formula-evaluation which correctly decides that the formula for each variable flip (with n variables, $O(n)$ clauses, and clause size at least 2) is satisfied or not with probability strictly larger than $1/2$, uses $\Omega(n)$ space in the worst case.*

Proof. We prove this using communication complexity, specifically, reducing the problem to the INDEX problem.

INDEX problem. Alice gets an n-bit string $x \in \{0,1\}^n$ and Bob gets an integer $i \in \{1, n\}$. Alice can communicate to Bob, but Bob cannot send messages to Alice. The goal is to find out x_i, the i-th index of x.

We know the lower bound of the randomized one-way communication complexity of INDEX is $\Omega(n)$ [33].

We now do the following reduction. Given $x \in \{0,1\}^n$, Alice construct a CNF formula F as follows. There are n clauses and $2n$ variables in F. For $1 \leq j \leq n$, clause $C_i = (v_{i,1} \wedge v_{i,2})$. If $x_i = 1$, we assign $v_{i,1} =$ TRUE and $v_{i,2} =$ FALSE. Otherwise, we assign $v_{i,1} =$ FALSE and $v_{i,2} =$ TRUE. Note that in formula F, every clause is initially satisfied and only one variable is TRUE in every clause.

Now given index i, we flip variable $v_{i,1}$. If the formula remains satisfied, then $x_i = 0$ in Alice's string, otherwise, $x_i = 1$. ◀

B.2 Proofs about k-Clique reduction General Reduction

▶ **Lemma 20.** *Consider a graph G, where we have not yet decided what edges exist or don't. There are $|V|$ vertices and $\binom{|V|}{2}$ potential edges. There exists a formula ϕ_ℓ with $n = \binom{|V|}{2}$ variables, width $k = \binom{\ell}{2}$ clauses, and a total of $m = \binom{|V|}{\ell}$ clauses such that when the n variables X are set with a 1 if the edge exists in G and 0 if the edge doesn't exist then the number of false clauses in $\phi_\ell(X)$ corresponds to the number of k cliques in G.*

Proof. In this reduction we will use variables to represent possible edges in the graph and we will use clauses to detect cliques. Let us index our variables X with two numbers i, j such that $X[i,j]$ is a variable if $i < j$ and $i, j \in [|V|]$. We will treat $X[i,j]$ as the variable corresponding to if the edge (i,j) exists in G. Now, for all $i_1 < \ldots < i_\ell$ where $i_1, \ldots, i_\ell \in [|V|]$ add the following clause to ϕ_ℓ:

$$\left(\bar{X}[i_1,i_2] \vee \bar{X}[i_1,i_3] \vee \ldots \vee \bar{X}[i_1,i_\ell] \vee \bar{X}[i_2,i_3] \vee \ldots \vee \bar{X}[i_{\ell-1},i_\ell]\right).$$

Note that if the nodes $i_1, \ldots, i_\ell$ in G are a ℓ-clique given the setting of edges implied by X, then this clause is false if any of the edges among vertices 1 to l are missing.

Each clause has size $k = \binom{\ell}{2}$. There are a total of $m = \binom{|V|}{\ell}$ clauses. We need one variable per potential edge in the graph, so there are $n = \binom{|V|}{2}$ variables. Given a setting of X we have a corresponding graph G, and the number of false clauses in ϕ_ℓ is exactly equal to the number of ℓ-cliques in G. ◀

▶ **Theorem 21.** *Assume ℓ is a constant. If the combinatorial ℓ-clique hypothesis is true then any combinatorial algorithm $\mathcal{A}$ for the Dynamic Counting Boolean Evaluation problem, $U(n,m,k)$, with polynomial preprocessing time $P(n,m,k) = poly(n) \cdot poly(m) \cdot poly(k)$ requires $m^{(1-k^{-1/2})(1-o(1))}$ time. We can also state the lower bound in terms of the number of variables n, in which case $U(n,m,k)$ requires at least $n^{(1-o(1))(\sqrt{2k}-2)/2}$ time.*

Proof. We are given an instance G of the ℓ-clique problem with $|V|$ vertices and $|E| = O(|V|^2)$ edges. By the combinatorial ℓ-clique hypothesis this problem requires $|V|^{\ell(1-o(1))}$ time.

First note that, with Lemma 10, we want $c = 1$ from our combinatorial k-clique hypothesis. Next assume that $P(n, m, k) = (n \cdot m \cdot k)^f$ for some constant f. Now set $d = \frac{(1-\epsilon)\ell}{(\ell+2)f}$ for some constant $\epsilon > 0$. Recall, with these settings, we can say that solving all $|V|^{\ell(1-d)}$ clique problems requires $|V|^{\ell(1-o(1))}$ time. This can be stated as a time for each of the problems of size $|V|^d$, they each require $\left(|V|^d\right)^\ell$ time.

Now we will use Lemma 9 to produce our formula ϕ on our problems of size $|V'| = |V|^d$. Note that $n = \binom{|V'|}{2} = \Theta(|V'|^2)$ and $k = \binom{\ell}{2} = \ell(\ell+1)/2$ and $m = \binom{|V'|}{\ell} = \Theta(|V'|^\ell)$. Finally, observe $P(n, m, k) = O\left((|V|^{d(\ell+2)})^f\right)$ which can be written as $P(n, m, k) = O\left((|V|^{\ell(1-\epsilon)})\right)$. Thus all of the updates we do to solve our clique problem must take $|V|^{\ell(1-o(1))}$ time.

Note that with at most $|E'| = O(|V|^{2d})$ variable updates we can cause the number of false clauses in ϕ_ℓ to be equal to the number of ℓ-cliques in a new instance. So $|V|^{2d}$ updates should take at least $\left(|V|^d\right)^\ell$ time. Further, $U(n, m, k)$ must be at least $|V|^{d(\ell-2)(1-o(1))} = |V'|^{(\ell-2)(1-o(1))}$. Now, lets re-state this in terms of n, m, k. First lets add our value of m into this equation:

$$|V'|^{(\ell-2)(1-o(1))} = m^{(1-o(1))(\ell-2)/\ell} = m^{(1-2/\ell)(1-o(1))}.$$

We can also write in terms of n:

$$|V'|^{(\ell-2)(1-o(1))} = n^{(1-o(1))(\ell-2)/2}.$$

Now note that $2\sqrt{k} > \ell > \sqrt{2k}$. So we can re-write our above equations as

$$|V'|^{(\ell-2)(1-o(1))} = m^{(1-1/\sqrt{k})(1-o(1))},$$

and

$$|V'|^{(\ell-2)(1-o(1))} = n^{(1-o(1))(\sqrt{2k}-2)/2}.$$ ◀

▶ **Theorem 22.** *Assume ℓ is a constant. If the ℓ-clique hypothesis is true then any algorithm $\mathcal{A}$ for the Dynamic Counting Boolean Evaluation problem, $U(n, m, k)$, with polynomial preprocessing time $P(n, m, k) = O(m)$ requires $m^{\left(\frac{2\omega-3}{3}-k^{-1/2}\right)(1-o(1))}$ time. We can also state the lower bound in terms of the number of variables n, in which case $U(n, m, k)$ requires at least $n^{\frac{2\omega-3}{6}\sqrt{2k}-1-o(\sqrt{k})}$ time.*

Using in $\omega = 2.37286$ (from the best current upper bound [6]) these lower bounds per update are: $m^{\left(0.5819-k^{-1/2}\right)(1-o(1))}$ and $n^{0.5819\sqrt{2k}-1-o(\sqrt{k})}$.

Using $\omega = 2$ (the smallest value ω could be) the lower bounds per update are: $m^{\left(1/3-k^{-1/2}\right)(1-o(1))}$ and $n^{\sqrt{2k}/3-1-o(\sqrt{k})}$.

Proof. We are given an instance G of the ℓ-clique problem with $|V|$ vertices and $|E| = O(|V|^2)$ edges. By the ℓ-clique hypothesis we have that this problem requires $|V|^{(1-o(1))\omega\ell/3}$ time.

First note that with Lemma 10, we want $c = \omega/3$ from our l-clique hypothesis. Next we assumed that $P(n, m, k) = O(m)$. Now set $d = \frac{(1-\epsilon)\omega}{3}$ for some constant $\epsilon > 0$ to be set later. Recall, with these settings we can say that solving all $|V|^{\ell(1-d)}$ clique problems requires $|V|^{(1-o(1))\omega\ell/3}$ time. For convenience call $\delta = \epsilon\omega/3$. This can be stated as a time for each of the problems of size $|V|^d$, they each require time

$$R(|V|) = \left(|V|^d\right)^{\left(\frac{c-1+d}{d}\right)\ell(1-o(1))} = \left(|V|^d\right)^{\left(\frac{2\omega-3-\delta}{3-\delta}\right)\ell(1-o(1))}.$$

We will use the reduction from Lemma 9. Let $|V'| = |V|^d$ Now lets note that $n = \binom{|V'|}{2} = \Theta(|V'|^2)$ and $k = \binom{\ell}{2} = \ell(\ell+1)/2$ and $m = \binom{|V'|}{\ell} = \Theta(|V'|^{\ell})$.

Now note that as before if $P(n, m, \ell) = O(m)$ then it is also $O(|V|^{d\ell})$ which can be restated as $|V|^{(1-\epsilon)\omega\ell/3}$. This is small enough that the updates must take $|V|^{(1-o(1))\omega\ell/3}$ time.

We require $O(|V|^{2d})$ updates per instance. And each instance requires $R(|V|)$ time. So each update requires $R(|V|)|V|^{-2d}$ time which we can expand to

$$\left(|V|^d\right)^{\left(\frac{2\omega-3-\delta}{3-\delta}\right)\ell(1-o(1))-2}$$

time. Now note that $2\sqrt{k} > \ell > \sqrt{2k}$. If we re-state this with m we get:

$$m^{\left(\frac{2\omega-3-\delta}{3-\delta}\right)(1-o(1))-1/\sqrt{k}}.$$

If we treat ϵ as a parameter we tune to be arbitrarily small we can re-state as follows:

$$m^{\frac{2\omega-3}{3}-1/\sqrt{k}-\epsilon'},$$

where ϵ' is some other arbitrarily small constant (a function of ϵ implicitly). So, if someone claims they have update time

$$m^{\frac{2\omega-3}{3}-1/\sqrt{k}-\gamma}$$

for a fixed constant γ then we can tune ϵ sufficiently small for that to give a contradiction from the k-clique hypothesis.

Now, if we state in terms of $n = O(|V'|^2)$

$$n^{\left(\frac{2\omega-3-\delta}{6-2\delta}\right)\ell(1-o(1))-1}$$

we can use the same trick with ϵ' to get the simplified version

$$n^{\left(\frac{2\omega-3}{6}-\epsilon'\right)\ell-1} = \Omega\left(n^{\left(\frac{2\omega-3}{6}-\epsilon'\right)\sqrt{2k}-1}\right)$$

which gives a lower bound of

$$n^{\frac{2\omega-3}{6}\sqrt{2k}-1-o(\sqrt{k})}$$

time per update. ◀

▶ **Lemma 23.** *Given a graph G let the procedure F be the process of selecting a uniformly random pair of vertices (u, v) and either deleting edge (u, v) it if it exists or inserting (u, v) if it doesn't exist.*

If we run the procedure F $\Theta\left(\binom{n}{2}\lg^2(n)\right)$ times on any graph G the resulting distribution over graphs has total variation distance at most $n^{2-\Omega(\lg(n))}$ from the distribution over Erdős-Rényi graphs.

Proof. Let $p = \binom{n}{2}^{-1}$, the probability any given edge is selected to be flipped by F. Let $s = \Theta\left(\binom{n}{2}\lg^2(n)\right)$, the number of times we run F.

For any given edge (u, v) it is flipped a Binomially distributed number of times $B(s, p)$. The $Pr\left[B(s, p) \equiv 0 \mod 2\right] = 1/2 + (1-2p)^s/2$, which is the probability that (u, v) stays an edge if it started with one and continues to have no edge if started without one. If this were an Erdős-Rényi graph the probability an edge exists is $1/2$ iid. So on each edge the TVD from the probability of $1/2$ is $(1-2p)^s$. Now note that $(1-2p)^s = (1-2p)^{(2/(2p))\Theta\left(\lg^2(n)\right)} < e^{-\Omega(\lg^2(n))} < n^{-\Omega(\lg(n))}$.

We can now union bound over all edges to get a TVD between running the procedure F $\binom{n}{2}\lg^2(n)$ times and Erdős-Rényi graphs of at most $n^{-\Omega(\lg(n))} = 2^{-\Omega(\lg^2(n))}$. ◀

B.3 Proof of Span Lower Bound in the Binary-Forking Model

▶ **Theorem 24.** *The span per update for dynamic boolean formula evaluation is* $\Omega(\log n)$ *in the binary forking model.*

Proof. We first show the work lower bound for a single update (or, the time required per update), and then using the work lower bound we compute the span lower bound in the binary-forking model. From Theorem 12, we know that the lower bound of work per update is n^c where $c = \Theta(1)$ for constant k.

In the binary forking model [12], it takes $\Omega(\log t)$ time to launch t threads. This is due to spawning t threads in a binary tree fashion where there are t leaves in the binary tree and the height is $\log t$. The height $\log t$ specifies the span of this launching process.

Let p be the number of processors used for the dynamic formula evaluation problem with n variables. The span $T_\infty(n)$ in the binary forking model is lower bounded by n^c/p and $\log p$. Combining them, we get the following:

$$T_\infty(n) = \Omega(n^c/p + \log p)$$

Minimizing $T_\infty(n)$ over values of p, we get $T_\infty(n) = \Omega(\log n)$. ◀