Incremental Topological Sort and Cycle Detection in $\tilde{O}(m\sqrt{n})$ Expected Total Time

Aaron Bernstein [*] Shiri Chechik[†]

Abstract

In the *incremental cycle detection* problem edges are inserted to a directed graph (initially empty) and the algorithm has to report once a directed cycle is formed in the graph. A closely related problem to the incremental cycle detection is that of the *incremental topological sort* problem, in which edges are inserted to an acyclic graph and the algorithm has to maintain a valid topological sort on the vertices at all times.

Both incremental cycle detection and incremental topological sort have a long history. The state of the art is a recent breakthrough of Bender, Fineman, Gilbert and Tarjan [TALG 2016], with two different algorithms with respective total update times of $\tilde{O}(n^2)$ and $O(m \cdot \min\{m^{1/2}, n^{2/3}\})$. The two algorithms work for both incremental cycle detection and incremental topological sort.

In this paper we introduce a novel technique that allows us to improve upon the state of the art for a wide range of graph sparsity. Our algorithms has a total expected update time of $\tilde{O}(m\sqrt{n})$ for both the incremental cycle detection and the topological sort problems.

1 Introduction

In dynamic graph algorithms our goal is to maintain some key functionality of a given graph while an adversary keeps changing the graph. In other words, the algorithm needs to handle an online sequence of update operations, where each update operation involves an insertion/deletion of an edge of the graph. We say that a dynamic algorithm is decremental if it handles only deletions, incremental if handles only insertions and fully dynamic if it handles both deletions and insertions.

A key objective in dynamic graph algorithms is to minimize the update time, the time it takes the algorithm to adapt to a change in the graph. In the incremental and the decremental setting, it is often convenient to consider the total update time, that is, the aggregate sum of update times over the *entire* sequence of insertions or deletions.

Dynamic graph algorithms have been the subject of an extensive study since the late 70's with many papers cover different aspects and problems of this setting. Many of the very basic dynamic graph algorithms in undirected graphs admit by-now near optimal solutions (see e.g. [10, 11, 12, 23, 24, 14, 9]). However, dealing with the directed case seems more challenging and in many of the fundamental problems for the directed case we are still far from seeing the full picture. In this paper we consider two fundamental dynamic graph problems on directed graphs, the *incremental cycle detection* and the *incremental topological sort* problems. In the incremental cycle detection problem we are given a directed acyclic graph, and edges are added to the graph one at a time; the algorithm then has to report the first time a directed cycle is formed in the graph. A closely related problem to the incremental cycle detection is that of the incremental topological sort problem. Here we are given a directed acyclic graph with edges added one a time, with the additional guarantee that the graph remains acyclic at all times; the algorithm has to maintain a valid topological sort on the vertices at all times.

The problems of detecting a cycle and maintaining topological sort arise naturally in applications for scheduling tasks where some tasks must perform before others. Abstractly, the tasks and constraints are represented by a directed graph, where every node is a task and an edge between two tasks represents a constraint that one task must be performed before the other.

In the static regime, for a given graph G, one can find either a cycle or a topological order in $O(n+m)$ time [1] [17, 20, 22]. Hence, the naive approach in which after every update we simply recompute everything from scratch yields a $O(nm+m^2)$ total update time.

The problems of the incremental cycle detection

[*]Technical University of Berlin, Germany. Supported by the Einstein Grant at TU Berlin. `bernstei@gmail.com`

[†]Tel Aviv University, Israel. Supported by the Israel Science Foundation grant No. 1528/15 and the Blavatnik Fund. `shiri.chechik@gmail.com`

[1]As usual, n (respectively, m) is the number of nodes (resp., edges) in the graph.

and topological order have been extensively studied in the last three decades [3, 19, 21, 15, 18, 2, 1, 16, 8, 4, 5, 6]. Marchetti-Spaccamela *et al.* [19] obtained algorithms for these problems in $O(nm)$ total update time. Katriel and Bodlaender [15] later gave algorithms with improved bounds of $O(\min\{m^{3/2}\log n, m^{3/2} + n^2\log n\})$. Afterward, Liu and Chao [18] improved the bound to $O(m^{3/2} + mn^{1/2}\log n)$, and Kavitha and Mathew [16] gave another algorithm with a total update time bound of $O(m^{3/2} + nm^{1/2}\log n)$. See [8] for further discussion on these problems.

Recently, Haeupler *et al.* [8] presented among other results an elegant algorithm for these problems with $O(m^{3/2})$ total update time. In a breakthrough result Bender, Fineman, Gilbert and Tarjan [5] presented two different algorithms, with total update time of $O(n^2\log n)$ and $O(m\cdot\min\{m^{1/2}, n^{2/3}\})$, respectively. Despite previous attempts, for sparse graphs no better than $O(m^{3/2})$ total update time algorithm was found.

In this paper we present a Las Vegas randomized algorithm that improves upon the state of the art for a large spectrum of graph sparsity – namely, when m is $\omega(n\log(n))$ and $o(n^{3/2})$.

Theorem 1.1 *There exists an incremental algorithm for dynamic cycle detection with expected total update time $O(m\sqrt{n}\log(n))$, where m refers to the number of edges in the final graph.*

Theorem 1.2 *There is an algorithm for incremental topological sort with expected total update time $O(m\sqrt{n}\log(n) + n\sqrt{n}\log^2(n))$.*

1.1 Preliminaries

In the incremental setting, we start with an empty graph, and the adversary inserts directed edges one at a time. Let G always refer to the current version of the graph. The update sequence terminates once G has a cycle; let $G_{\text{FINAL}} = (V, E_{\text{FINAL}})$ refer to the final version of the graph *before the update that led to a cycle*, let let $m = |E_{\text{FINAL}}|$, and let $n = |V|$. We will often speak of the update sequence up to G_{FINAL}, which excludes the very last update if it leads a cycle; thus the graph remains acyclic during the entire update sequence up to G_{FINAL}. Since edges are only inserted, any topological sort for G_{FINAL} is also a topological sort for the current graph G. For the sake of our analysis we fix some topological sort T_{FINAL} of G_{FINAL}; thus, for every $(x,y) \in E_{\text{FINAL}}$ we have $T_{\text{FINAL}}(x) \le T_{\text{FINAL}}(y)$. Given sets S, T, recall that $S \setminus T = \{s \in S | s \notin T\}$ and $S \oplus T = (S \setminus T) \cup (T \setminus S)$.

For any two vertices $u, v \in V$, we say that u is an *ancestor* of v if there is a path from u to v in G. Note that ancestors (and descendants below) are always defined with respect to the current graph G. We let $A(u)$ denote the set of ancestors of u in G. Analogously, if there is a path from u to v then we say that v is a descendant of u, and we let $D(u)$ denote the set of descendants of u in G. We say that u is both an ancestor and a descendant of itself. We say that two vertices u and v are related if one is the ancestor of the other.

We rely on the following existing result about incremental reachability in directed acyclic graphs.

Lemma 1.1 *[13] Given any vertex v, there exists an algorithm that maintains $A(v)$ and $D(v)$ in total time $O(m)$ over the entire sequence of edge insertions.*

Note that the *average* vertex degree in G_{FINAL} is $2m/n$. Our analysis requires that no vertex has degree much larger than this. We will justify this assumption by showing that for every graph $G = (V, E)$ there is a graph $G' = (V', E')$ such that $|E'| = O(|E|)$, $V \subseteq V'$ and $|V'| = O(|V|)$, every vertex $v' \in V'$ has degree at most $O(|E'|/|V'|) = O(|E|/|V|)$, and for any pair of vertices $u, v \in V$, there is $u - v$ path in G' iff there is such a path in V; thus, in particular, G' has a cycle iff G does. This reduction justifies the following assumption; the proof of the reduction is deferred to Section A.

Assumption 1.1 *We assume for the rest of the paper that every vertex in the current graph $G = (V, E)$ has degree $O(|E|/|V|) = O(m/n)$; recall that $n = |V|$ and $m = |E_{\text{FINAL}}|$.*

2 High Level Overview of Techniques

We now give a high level overview of our algorithm for incremental cycle detection; the algorithm for incremental topological sort uses the same basic ideas, but is a good deal more involved. Let us consider the insertion of a new edge (u, v). We want to determine if (u, v) created a cycle. The trivial algorithm would be to do a forward search from v, and see if it reaches vertex u, but this requires $O(m)$ time. We would like to prune the set of vertices that we have to search from v. Let us say that two vertices x and y are equivalent if they are related and $A(x) = A(y)$ and $D(x) = D(y)$; it is easy to see that if two vertices are on a cycle, then they are equivalent. Thus, when we insert edge (u, v), our forward search from v only needs to look at vertices equivalent to v.

Unfortunately efficiently maintaining equivalent vertices is at least as hard as finding a cycle. For this reason we consider a relaxed notion of equivalence. Given any set S, we will say that u and v are S-equivalent if $A(x) \bigcap S = A(y) \bigcap S$ and $D(x) \bigcap S = D(y) \bigcap S$. Now, let us say that we picked S by sampling every vertex in V independently with probability

$O(\log(n)/\sqrt{n})$. A standard Chernoff bound argument will show that with high probability, every vertex x in a directed acyclic graph is S-equivalent to $O(\sqrt{n})$ vertices; conversely, if x is found to be S-equivalent to more than $O(\sqrt{n})$ vertices, then w.h.p the graph contains a cycle through x. We will also show that using $O(m|S|)$ total update time, at any given time given any pair $(x, y) \in V$, we can determine in $O(1)$ time whether x and y are S-equivalent; we do so by maintaining incremental reachability from every vertex in S, which by Lemma 1.1 takes $O(m|S|) = O(m\sqrt{n}\log(n))$ time. This immediately yields a very simple algorithm with amortized update time $O(m/\sqrt{n})$ (total update time $O(m^2/\sqrt{n})$). Namely, when the new edge (u, v) is inserted, v only needs to do a forward search amongst all vertices S-equivalent to v. As indicated above, we can easily test for each vertex that is encountered in the forward search if it is S-equivalent to v. If there are more than $O(\sqrt{n})$ of them, the graph contains a cycle with high probability. Otherwise, since by assumption 1.1 each vertex has degree $O(m/n)$, the forward search requires time $O((m/n)\sqrt{n}) = O(m/\sqrt{n})$.

To improve this update time, we introduce the following definition: x and y are sometime-S-equivalent if x and y are S-equivalent at any point during the update sequence. Now, although at any one time x is S-equivalent to at most $O(\sqrt{n})$ vertices, one can construct an example where some particular vertex x is sometime-S-equivalent to every other vertex in the graph. But the main technical lemma of our paper shows that with high probability, *on average* a vertex is sometime-S-equivalent to only $O(\sqrt{n}\log(n))$ vertices; or rather, if this is not the case, then the graph contains a cycle. This lemma suggest the following algorithm: each vertex v maintains incremental reachability in the set of all vertices u that are sometime-S-equivalent to v; since there are $O(\sqrt{n}\log(n))$ of them, this takes time $O((m/n)(\sqrt{n}\log(n)))$ per vertex, for a total of $O(m\sqrt{n}\log(n))$ as desired. When we insert an edge (u, v) we could then detect a cycle by simply checking if u is a sometime-S-equivalent vertex reachable from v.

Unfortunately, although our main Lemma bounds the number of sometime-S-equivalent pairs, the proof is quite complicated and relies on the topological order T_{FINAL}, which we do not know. For this reason, we are not able to constructively keep track of all the sometime-S-equivalent pairs. We overcome this barrier with a slightly more complicated algorithm, which does not explicitly keep track of all sometime-S-equivalent pairs, but is able to charge the work it does to the number of such pairs.

3 Similarity and Equivalence

Definition 3.1 *We say that vertices u and v are* τ-similar *in the current graph G if they are related $\wedge$ $|A(u) \oplus A(v)| \leq \tau \wedge |D(u) \oplus D(v)| \leq \tau$. We say that u and v are* sometime-τ-similar *if u and v are τ-similar at any point during the entire update sequence up to G_{FINAL}.*

Note that in order to be similar, two vertices u and v have to related. So for example in a graph with no edges, although all vertices have identical (empty) reachability sets, no pair of them τ-similar.

Lemma 3.1 *At any point during the update sequence up to G_{FINAL}, any vertex v is τ-similar to at most $2\tau+2$ vertices in the current graph G.*

Proof. Let S_v be the set of vertices τ-similar to v, and say for contradiction that $|S_v| \geq 2\tau + 3$. Then either $|A(v) \bigcap S_v| \geq \tau + 2$ or $|D(v) \bigcap S_v| \geq \tau + 2$; assume that $|D(v) \bigcap S_v| \geq \tau + 2$ since the proof for the other case is analogous. Let u be the vertex in $D(v) \bigcap S$ with highest $T_{\text{FINAL}}(u)$. But now note that $D(v)$ contains all $\tau + 2$ vertices in $|D(v) \bigcap S_v|$, which $D(u)$ contains only one of them (itself), so $|D(v) \oplus D(u)| \geq \tau + 1$, which contradicts $u \in S_v$.

We now turn to bounding *sometime* τ-similarity. Although Lemma 3.1 shows that at a given time v can be 1-similar to at most 4 vertices, there is a simple example in which a particular vertex v can end up being sometime-1-similar to every vertex in V. Let us say that the other vertices (in addition to v) are $(u_1, u_2, ..., u_{n-1})$. The update sequence will proceed in phases, each of which inserts one or two edges. In the first phase the adversary inserts an edge from u_1 to v. In the second phase it inserts an edge from u_1 to u_2 and u_2 to v. More generally in the kth phase it inserts an edge from u_{k-1} to u_k and from u_k to v. It is easy to see that after phase k, v is sometime-1-similar to u_k.

But in the example above, although v is sometime-1-similar to all n vertices, every other vertex is sometime-1-similar to only 2 vertices. We now argue that we can bound the total number of sometime-τ-similar pairs. This is the main Lemma of our paper, and the proof is quite involved.

Lemma 3.2 *For any positive integer τ, the total number of pairs (u, v) such that u and v are sometime-τ-similar is $O(n\tau\log(n))$*

To prove Lemma 3.2 we first introduce some notations.

Definition 3.2 *For any two nodes u and v, we say that $I(u) < I(v)$ if u appears before v in the final topological order T_{FINAL}. We define the interval $I(u,v)$ to be the set of nodes in T_{FINAL} between u and v, including u and v; $I(u,v)$ is defined to be empty if v appears before u in T_{FINAL}.*

At a high level, in our proof of Lemma 3.2 we maintain for every node v two sets $\hat{A}(v)$ and $\hat{D}(v)$, such that at all times we have $|\hat{A}(v)| \leq 2\tau$ and $|\hat{D}(v)| \leq 2\tau$ and $\hat{A}(v) \subseteq A(v)$ and $\hat{D}(v) \subseteq D(v)$. Some nodes may be added and removed from the sets $\hat{A}(v)$ and $\hat{D}(v)$. We will show that the number of all the insertions to the sets $\hat{A}(v)$ and $\hat{D}(v)$ for all $v \in V$ is $O(n\tau \log n)$. In addition, we will show that there are at most $O(n\tau \log n)$ pairs (u,v) such that u and v are sometime-τ-similar and yet u is not added to $\hat{A}(v)$ or $\hat{D}(v)$. Together these two claims imply the lemma.

We next introduce some additional definitions on the set $\hat{A}(v)$ and $\hat{D}(v)$ for $v \in V$.

Definition 3.3 *Consider two nodes u and v such that u is an ancestor of v. We say that a node v has a* **space free spot** *for u if $|\hat{A}(v)| < 2\tau$. We say that v has an* **asymmetry free spot** *for u if $|\hat{A}(v)| = 2\tau$ and $\hat{A}(v)$ contains a node x such that $I(x) < I(u)$ and $v \notin \hat{D}(x)$. If v has either a space free spot for u or an asymmetry free spot for u then we say that v has a* **free spot** *for u.*

If v has a free spot for u then we say that we add u to the free spot in $\hat{A}(v)$ if we do the following: if v has a space free spot for u then we just add u to $\hat{A}(v)$; if v has an asymmetry free spot for u then we add u to $\hat{A}(v)$ and in order to to maintain that $|\hat{A}(v)| \leq 2\tau$ we remove some vertex x from $\hat{A}(v)$ with the property that $I(x) < I(u)$ and $v \notin \hat{D}(x)$.

Similarly, we say that u has a **space free spot** *for v if $|\hat{D}(u)| < 2\tau$, and u has an* **asymmetry free spot** *for v if $|\hat{D}(u)| = 2\tau$ and $\hat{D}(u)$ contains a node x such that $I(v) < I(x)$ and $u \notin \hat{A}(x)$. If u has either a space free spot for v or an asymmetry free spot for v then we say that u has a* **free spot** *for v.*

If u has a free spot for v then we say that we add v to the free spot in $\hat{D}(u)$ if we do the following: if u has a space free spot for v then we just add v to $\hat{D}(u)$; if u has an asymmetry free spot for v then we add v to $\hat{D}(u)$ and remove some vertex x from $\hat{D}(u)$ with the property that $I(v) < I(x)$ and $u \notin \hat{A}(x)$.

Definition 3.4 *Let $e_A(v)$ (e stands for extreme) be the vertex in $\hat{A}(v)$ farthest away from v with respect to the distance in the final topological sorting T_{FINAL}. Similarly, let $e_D(v)$ be the vertex in $\hat{D}(v)$ farthest away from v. Let $I_A(v) = I(e_A(v), v)$. Similarly, let $I_D(v) = I(v, e_D(v))$.*

Definition 3.5 *When a node u becomes an ancestor of v and $|I(u,v)| \leq |I_A(v)|/2$, we say that u halves v. Similarly, when a node u becomes a descendant of v and $|I(v,u)| \leq |I_D(v)|/2$, we also say that u halves v.*

Defining the sets $\hat{A}(v)$ and $\hat{D}(v)$ Let us consider a specific vertex v. Both $\hat{A}(v)$ and $\hat{D}(v)$ are initially empty. We now define how the sets can change as edges are added to G. We refer to this as the **main set protocol**. The main set protocol will always enforce that $|\hat{A}(v)| \leq 2\tau$ and $|\hat{D}(v)| \leq 2\tau$.

1. If the insertion of some edge into G causes some vertex u to become an ancestor of v and u halves v then u is added to $\hat{A}(v)$. If as a result we have $|\hat{A}(v)| > 2\tau$ then $e_A(v)$ is removed from $\hat{A}(v)$.

2. Similarly, if the insertion of some edge into G causes some vertex u to become a descendant of v, and u halves v, then u is added to $\hat{D}(v)$. If as a result we have $|\hat{D}(v)| > 2\tau$ then $e_D(v)$ is removed from $\hat{D}(v)$.

3. Say that some insertion in G causes u and v to become τ-similar and say that u is an ancestor of v. Then if u has a free spot for v AND v has a free spot for u then add u to the free spot in $\hat{A}(v)$ and add v to the free spot in $\hat{D}(u)$; recall the definition of adding to a free spot in Definition 3.3.

Clearly we always have $|\hat{A}(v)| \leq 2\tau$ and $|\hat{D}(v)| \leq 2\tau$. We now show that the sets also satisfy the following invariants

Observation 3.1 *The size of $\hat{A}(v)$ never decreases, and an element is removed from $\hat{A}(v)$ only when $|\hat{A}(v)| = 2\tau$ and a new vertex is inserted into $\hat{A}(v)$. The same holds for $\hat{D}(v)$.*

Observation 3.2 *Let us say that $\hat{A}(v) = 2\tau$ and u is an ancestor of v that is τ-similar to v. Then there are at least τ vertices in $\hat{A}(v)$ that are ancestors of u. This is because if u and v are τ-similar then by definition their ancestor sets can differ by at most τ vertices. Similarly, if $\hat{D}(v) = 2\tau$ and u is a descendant of v that is τ-similar to v then there are at least τ vertices in $\hat{D}(v)$ that are descendants of u.*

Invariant 3.1 *(Improving Invariant) Say that a vertex x is inserted into $\hat{A}(v)$ and as a result y is removed from $\hat{A}(v)$. Then x is closer to v in T_{FINAL} than y is. The same holds for $\hat{D}(v)$.*

Proof. We will prove the invariant for $\hat{A}(v)$; the proof for $\hat{D}(v)$ is analogous. There are four reasons a vertex x can be inserted into $\hat{A}(v)$. **1)** x is added to a space-free spot in $\hat{A}(v)$ (step 3 of the main set protocol), in which case no vertex is removed from $\hat{A}(v)$. **2)** If x is added to an asymmetry free spot in $\hat{A}(v)$ (step 3) then by definition of adding to an asymmetry free spot (Definition 3.3), the vertex y removed from $\hat{A}(v)$ has $I(y) < I(x)$. **3)** If x is added to $\hat{A}(v)$ because it halves v (step 1) then by definition of halving x is closer to v then the vertex $e_A(v)$ that is removed from $\hat{A}(v)$. **4)** x might be added to $\hat{A}(v)$ because v halved $\hat{D}(x)$ and x and v are τ-similar (step 2). But then by Observation 3.2 at least τ of the vertices in $\hat{A}(v)$ are further from v than x is, and this is in particular true of the displaced vertex $e_A(v)$.

We now prove the crucial claims that bound how often the sets $\hat{A}(v)$ and $\hat{D}(v)$ can change.

Claim 3.1 *For any given vertex v, the total number of times that v is halved by some vertex u over the entire update sequence is at most $O(\tau\log(n))$.*

Proof. We will show that $\hat{A}(v)$ is halved at most $O(\tau\log(n))$ times; the proof for $\hat{D}(v)$ is analogous. Let t be the first time in the update sequence when $|\hat{A}(v)| = 2\tau$. Note that up to time t, $|\hat{A}(v)|$ is halved at most 2τ times, because every vertex that halves $\hat{A}(v)$ is inserted into $\hat{A}(v)$ (step 1 of the main set protocol), and by Observation 3.1 there are exactly 2τ insertions into $\hat{A}(v)$ up to time t (since before time t vertices are never removed from $\hat{A}(v)$). We now need to bound the number of times $\hat{A}(v)$ is halved after time t. Define a potential function $\phi(v) = \sum_{w\in\hat{A}(v)} \log(|I(w,v)|)$. Clearly $\phi(v)$ is always non-negative and at time t we have $\phi(v) \leq 2\tau\log(n)$. Moreover, $\phi(v)$ only decreases after time t because by Observation 3.1 we always have $|\hat{A}(v)| = 2\tau$ after time t, and by Invariant 3.1 vertices in $\hat{A}(v)$ only get closer to v over time. We now complete the proof by observing that every time a vertex u halves $\hat{A}(v)$ after time t, $\phi(v)$ decreases by at least one. This is because the insertion of u into $\hat{A}(v)$ causes $e_A(v)$ to be removed from $\hat{A}(v)$, and by definition of halving $|I(u,v)| \leq |I(e_A(v),v)|/2$ so $\log(|I(u,v)|) \leq \log(|I(e_A(v),v)|) - 1$.

Claim 3.2 *The total number of times over the entire update sequence that a vertex is inserted into some $\hat{A}(v)$ or $\hat{D}(v)$ is at most $O(n\tau\log(n))$.*

Proof. Recall that a node u is added to $\hat{A}(v)$ (or analogously $\hat{D}(v)$) because either **1)** u halves v or **2)** v halves u and u and v are τ-similar or **3)** $\hat{A}(v)$ (or $\hat{D}(v)$) has a free spot for u. By Claim 3.1 the total number of times that one vertex halves another is $O(n\tau\log n)$, so the number of insertions due to cases 1) and 2) is $O(n\tau\log n)$.

We are left with bounding the total number of times any vertex u is added into any $\hat{A}(v)$ (or $\hat{D}(v)$) in a free spot. There are two cases of free spots: space-free spot and asymmetry-free spot. The number of insertions to $\hat{A}(v)$ and $\hat{D}(v)$ at space-free spots for a node v is bounded by 2τ because by Observation 3.1 once the size of $\hat{A}(v)$ (or $\hat{D}(v)$) reaches 2τ it will no longer have a space-free spot as its size will always remains 2τ (and up until that point there are no deletions from $\hat{A}(v)$ or $\hat{D}(v)$). Thus the total number of insertions into space free spots is the desired $O(n\tau)$.

We now claim that the total number of insertions into any set $\hat{A}(v)$ (or $\hat{D}(v)$) at an asymmetry free spot is bounded by twice the total number of halving events, which by Claim 3.1 is $O(n\tau\log n)$; this will complete the proof of the claim.

We use a potential function ϕ that upper bounds the total number of asymmetry-free spots at the sets $\hat{A}(v)$ and $\hat{D}(v)$. In particular, let $\phi(v)$ be the number of nodes u such that $u \in \hat{A}(v)$ and $v \notin \hat{D}(u)$ plus the number of nodes u such that $u \in \hat{D}(v)$ and $v \notin \hat{A}(u)$. Let $\Phi = \sum_v \phi(v)$.

We will show that each halving event adds at most 2 to Φ and that each insertion to some set $\hat{A}(v)$ or $\hat{D}(v)$ at an asymmetry-free spot decreases Φ by at least 1. This will imply that the total number of insertions to any set $\hat{A}(v)$ or $\hat{D}(v)$ at an asymmetry free spot is bounded by twice the total number of halves, as desired.

Consider a node u that was added to $\hat{A}(v)$ (resp. to $\hat{D}(v)$) due to a halving event. We need to show that Φ increases by at most 2. To see this note that the only nodes whose potential might have changed by this insertion is either v itself or the node $e_A(v)$ (resp. $e_D(v)$) if it was removed from the set $\hat{A}(v)$ (resp. from $\hat{D}(v)$) as a result of u's insertion. If $v \notin \hat{D}(u)$ then a new asymmetry is created and $\phi(v)$ might have increased by 1. If $e_A(v)$ (resp. $e_D(v)$) was removed from the set $\hat{A}(v)$ (resp. from $\hat{D}(v)$) as a result of u's insertion then note that now it might be that $v \in \hat{D}(e_A(v))$ (resp. $v \in \hat{A}(e_D(v))$) but $e_A(v) \notin \hat{A}(v)$ (resp. $e_D(v) \notin \hat{D}(v)$) so the potential of $e_A(v)$ (resp. $e_D(v)$) might increase by 1. So overall Φ increases by at most 2 as a result of a halve.

We are left to show that Φ decreases by at least one when a vertex u is added to an asymmetry free spot in some $\hat{A}(v)$ or $\hat{D}(v)$ for a vertex v that is τ-similar to u. Assume w.l.o.g. that u is an ancestor of v. Recall that according to Step 3 in our main set protocol we have that u is added to a free spot in $\hat{A}(v)$ AND v is added to a free spot in $\hat{D}(u)$. We are only concerned with the case where at least one of these free spots is an asymmetry free spot so let us say that u is added to $\hat{A}(v)$ in an asymmetry free spot. By definition of adding to an asymmetry free spot (Definition 3.3), adding u to $\hat{A}(v)$ leads to the removal of a node x from $\hat{A}(v)$, where $v \notin \hat{D}(x)$. The potential $\phi(v)$ thus decreases by 1 because after it is removed from $\hat{A}(v)$, x no longer contributes to $\phi(v)$. Also adding u to $\hat{A}(v)$ does not increase $\phi(v)$ because in Step 3 of our main set protocol we also add v to $\hat{D}(u)$, so no asymmetry is created.

Proof of Lemma 3.2

Let us say that a pair (u, v) is a *bad pair* if u and v become τ-similar and yet at this time u is not present in or added to $\hat{A}(v)$ or $\hat{D}(v)$ and similarly v is not present in or added to $\hat{A}(u)$ or $\hat{D}(u)$. We will show that there are at most $O(n\tau \log(n))$ such bad pairs (u, v). Combined with Claim 3.2 this proves the lemma.

We use a charging argument. For every set $\hat{A}(v)$ (or $\hat{D}(v)$) and every pair of nodes (u, x) such that u and x are in $\hat{A}(v)$ at the same time, v adds a credit of $1/\tau$ to the pair (u, x).

We now show that the total number of credits given is $O(n\tau \log(n))$. To see this, note that v only gives new credit when a new vertex u is added to $\hat{A}(v)$ or $\hat{D}(v)$. In particular when some vertex u is added to $\hat{A}(v)$ (the proof is analogous for $\hat{D}(v)$), v gives $1/\tau$ credits to pair (u, x) for every x that is in $\hat{A}(v)$ at the time that u is added to $\hat{A}(v)$. But since we always have $|\hat{A}(v)| \leq 2\tau$, v only gives 2 credits when a vertex u is inserted into $\hat{A}(v)$ or $\hat{D}(v)$. Combined with Claim 3.2 this yields the desired $O(n\tau \log(n))$ bound on the total number of credit given over the entire sequence of updates.

Now consider a bad pair (u, v). Note that by Step 3 of our main set protocol, the only way that u and v could fail to be added to each others sets if at least one has no free spot for the other. Assume w.l.o.g that u is an ancestor of v and v has no free spot for u. Since u and v are τ-similar, Observation 3.2 implies that are at least τ nodes x in $\hat{A}(v)$ such that x is ancestor of u, and in particular, $I(x) < I(u)$. Now note that since v has no free spot for u, v in particular has no asymmetry free spot for u, so for all nodes $x' \in \hat{A}(v)$ such that $I(x') < I(u)$ we have $v \in \hat{D}(x')$. In particular, for the τ nodes $x \in \hat{A}(v) \bigcap A(u)$ mentioned above we have $v \in \hat{D}(x)$. We also claim that for each $x \in \hat{A}(v) \bigcap A(u)$ we have $u \in \hat{D}(x)$. To see this, note that since (u, v) is a bad pair u does not halve v, as otherwise by Step 1 of the main set protocol u would have been inserted into $\hat{A}(v)$. Thus by definition of halving we must have that u is closer to $e_A(v)$ than to v in T_{FINAL}, so since $v \in \hat{D}(x)$, we must have that $|I(x, u)| \leq |I(e_A(v), u)| < |I(u, v)| \leq I(u, e_D(x))$. It follows that u halves x so $u \in \hat{D}(x)$, as desired. Thus, u and v are in $\hat{D}(x)$ at the same time and therefore x gives a credit of $1/\tau$ to the pair (u, v). Recall that there are τ such nodes $x \in \hat{A}(v) \bigcap A(u)$ and each gives a credit of $1/\tau$ to the pair (u, v). Therefore, 1 credit overall is given for the pair (u, v) and there is enough credit to pay for this bad pair. Combined with the $O(n\tau \log(n))$ bound on the total number of credit, this completes the proof of the lemma. □

Lemma 3.2 makes the notion of τ-similarity useful for our analysis, but it is not very algorithmically useful because the sets $A(v)$ and $D(v)$ are hard to maintain efficiently. Our algorithm instead focuses on equivalence with respect to a set S, defined below. We later show that if S is sampled uniformly at random with probability $O(\log(n)/\tau)$, then S-equivalence is closely related to τ-similarity.

Definition 3.6 *Given a graph G, any set of vertices $S \subseteq V$, and any vertex v, let $A_S(v) = A(v) \bigcap S$ and $D_S(v) = D(v) \bigcap S$. We say that two vertices u and v are S-equivalent if u and v are related $\wedge$ $A_S(u) = A_S(v)$ $\wedge$ $D_S(u) = D_S(v)$. We say that two vertices u and v are sometime-S-equivalent if u and v are S-equivalent at any point during the update sequence up to G_{FINAL}.*

Note that as with similarity, two vertices must be related to be equivalent.

Lemma 3.3 *In any graph G and any $S \subset V$, if u and v are related $\wedge$ $|A_S(u)| = |A_S(v)|$ $\wedge$ $|D_S(u)| = |D_S(v)|$ then u and v are S-equivalent.*

Proof. Let us say w.l.o.g that $u \in A(v)$. Then it is easy to see that $A(u) \subseteq A(v)$ and so $A_S(u) \subseteq A_S(v)$; thus $|A_S(u)| = |A_S(v)|$ implies $A_S(u) = A_S(v)$. The proof that $D_S(u) = D_S(v)$ is analogous.

Lemma 3.4 *Given any set $S \subseteq V$, it is possible to maintain the following sets in total update time $O(m|S|)$ over the entire update sequence:*

1. *The sets $A(s)$ and $D(s)$ for every $s \in S$*
2. *The sets $A_S(v)$ and $D_S(v)$ for every $v \in V$*
3. *A data structure that given any* related *pair (u, v) answers in $O(1)$ time whether u and v are S-equivalent.*

Proof. The first point follows trivially from Lemma 1.1. For the second point, whenever some vertex v is added to $A(s)$ for some $s \in S$, we add s to $D_S(v)$; similarly when v is added to $D(s)$ we add s to $A_S(v)$. Finally, for the third point, the data structure simply maintains $|A_S(v)|$ and $|D_S(v)|$ for every vertex v. Given any related pair (u, v), the data structure answers that u and v are S-equivalent if $|A_S(u)| = |A_S(v)|$ and $|D_S(u)| = |D_S(v)|$; otherwise it answers no. Correctness follows directly from Lemma 3.3.

Lemma 3.5 *For any positive integer τ, Let $S_\tau \subseteq V$ be obtained by independently sampling each vertex in V with probability $11\log(n)/\tau$. Then with high probability, at any time during the update sequence, and for all pairs $u, v \in V$, if u and v are S_τ-equivalent then u and v are τ-similar.*

Proof. Note that because we assumed a non-adaptive adversary, the vertices of S_τ are picked completely independently with probability $11\log(n)/\tau$ *from the perspective of every version of the graph during the entire update sequence.* Now let us focus on a specific pair u, v during a specific point in the update sequence. Let us say that u and v are NOT τ-similar. Then either $|A(u) \oplus A(v)| \geq \tau$ or $|D(u) \oplus D(v)| \geq \tau$; let us say, w.l.o.g, that $|A(u) \oplus A(v)| \geq \tau$. Then an easy application of the Chernoff bound shows that with probability at least $1 - 1/n^5$, $A(u) \oplus A(v)$ contains a vertex in S_τ and so u and v are NOT S_τ-equivalent: thus, if u and v *are* S_τ-equivalent then with probability at least $1 - 1/n^5$ they are τ-similar. Thus by the union bound this is true for all pairs (u, v) (at most n^2) in all versions of the graph (again at most n^2) with probability $1 - n^4/n^5 = 1 - 1/n$.

Corollary 3.1 *With high probability, the total number of sometime-S_τ-equivalent pairs is $O(n\tau\log(n))$.*

4 Incremental Cycle Detection

For the rest of this section, we let $S^* \subseteq V$ be a fixed set obtained by sampling every $v \in V$ with probability $11\log(n)/\sqrt{n}$. Note that with high probability $|S^*| = O(\sqrt{n}\log(n))$. Since these claims are true with high probability, we can assume for the rest of this section that $|S^*| = O(\sqrt{n}\log(n))$ and that 3.5 holds.

Observation 4.1 *Let us say that a graph G contains a cycle C. Then any pair of vertices u and v in C have the same ancestors and descendants so they are certainly S^*-equivalent.*

As discussed in the high level overview (Section 2), if we could efficiently maintain for each vertex v the set of all of vertices sometime-S^*-equivalent to v, this would yield a very simple algorithm for the problem with the desired $O(m\sqrt{n}\log(n))$ total update time; each vertex v would simply use Lemma 1.1 to maintain incremental reachability to and from its sometime-S^*-equivalent vertices. Unfortunately, even though we can check in $O(1)$ time if any particular pair u, v is S^*-equivalent, we do not know how to explicitly maintain all such pairs efficiently. We thus need to use a slightly more complicated algorithm.

The Algorithm The algorithm maintains all the information of Lemma 3.4 in time $O(m|S^*|) = O(m\sqrt{n}\log(n))$. It also maintains, for every vertex v, a set $\mathcal{A}(v) \subseteq A(v)$; a vertex u will only be added to $\mathcal{A}(v)$ if u and v are S^*-equivalent at that time, which will keep the sets $\mathcal{A}(v)$ small on average. See Figure 1 below for a pseudocode description for processing the insertion of edge (u, v). Note that our algorithm uses an internal list of vertices denoted TO-EXPLORE, and that TO-EXPLORE.pop-arbitrary-element removes an arbitrary vertex w from TO-EXPLORE. Also when the pseudocode says "return cycle", the algorithm runs a standard (non-dynamic) cycle-detection algorithm $O(m)$ time.

We also need to describe the data structure for $\mathcal{A}(v)$. The algorithm only performs insertions and lookups into each $\mathcal{A}(v)$, both of which can be done in expected constant time with hashing. If we allow $O(n^2)$ update and preprocessing time, then we assign an integer from 0 to $n-1$ to every vertex and initialize each $\mathcal{A}(v)$ as an array of size n; insertions and lookups can then be done in $O(1)$ worst-case time.

Invariant 4.1 *For every vertex v, $\mathcal{A}(v) \subseteq A(v)$.*

Running Time Analysis Recall that there is only one edge insertion that creates a cycle; once a cycle is found, the algorithm ceases. We thus first bound the time to process the edge (u, v) that creates a cycle. The execution of the while loop takes $O(m)$ time because step 5 can be executed at most once for each vertex w (because after the first time $u \in \mathcal{A}(w)$), so each edge (w, z) is never looked at more than once. The time to actually compute the cycle is then also $O(m)$. We now have to analyze the total update time of all the other edges; that is, the total update time of the sequence up to G_{FINAL}. Consider the insertion of some edge (u, v). Recalling that all vertex degrees are $O(m/n)$ (Assumption 1.1), it is not hard to check that processing this insertion requires time $O(1)$ + $[O(m/n)$ times the number of times step 5 was executed$]$. But note that every time step 5 is executed, some vertex u that was not previously in $\mathcal{A}(w)$ is added to $\mathcal{A}(w)$, for some pair (u, w) that is S^*-equivalent. By Corollary 3.1 step 5 is thus executed a total of $O(n^{1.5}\log(n))$ times over the entire update sequence (with high probability), leading to total update time $O((m/n)n^{1.5}\log(n)) = O(m\sqrt{n}\log(n))$.

Figure 1: Algorithm for inserting an edge (u, v) to G.

```
Cycle Detection: Insert $(u,v)$ in $G$
Initialize TO-EXPLORE= $\{v\}$
While TO-EXPLORE$\neq \emptyset$
  $w$ = TO-EXPLORE.pop-arbitrary-element
  1. if $w = u$
       return cycle                    \\ Have found a path $v \rightsquigarrow u$
  2. If $w \in \mathcal{A}(u)$                    \\ Recall that $\mathcal{A}(u) \subseteq A(u)$.
       return cycle                    \\ $u \rightsquigarrow w \rightsquigarrow u$.
  3. Else If $u \in \mathcal{A}(w)$
       do nothing                      \\ There was already a path $u \rightsquigarrow w$ before the insertion.
  4. Else If $u$ and $w$ NOT $S^*$-Equivalent      \\ Lemma 3.4: can check this in $O(1)$ time.
       do nothing                      \\ By observation 4.1 no cycle with $w$
  5. Else                              \\ $u$ and $w$ are $S^*$-Equivalent
       Add $u$ to $\mathcal{A}(w)$                   \\ Note: $u$ was not in $\mathcal{A}(w)$ before
       For each $(w,z) \in E$
            Add $z$ to TO-EXPLORE
```

Correctness Firstly, note that when we insert (u, v), we start with only v in TO-EXPLORE, and we only explore the outgoing edges of vertices in TO-EXPLORE, so if w is explored then $w \in D(v)$ *before* the insertion of (u, v) and $w \in D(u)$ after the insertion. Thus, adding u to $\mathcal{A}(w)$ in step 5 maintains Invariant 4.1.

We now show that when the algorithm performs "return cycle", there is indeed a cycle in the graph. In step 1, if u was popped from TO-EXPLORE then by the above paragraph $u \in D(v)$, which combined with the inserted edge (u, v) leads to a cycle $u \to v \rightsquigarrow u$. In step 2, if we explore a vertex w then $w \in D(u)$, so if we also have $w \in \mathcal{A}(u) \subseteq A(u)$ then we have a cycle through u and w.

We now show that if (u, v) is the *first* edge to create a cycle in the graph, then the algorithm runs "return cycle". To see this, let the cycle be $u = x_0, v = x_1, x_2, x_3, ..., x_k, u = x_{k+1}$. Let x_i be the vertex of highest index such that x_i is added to TO-EXPLORE. If $i = k+1$ then we are done because in this case $u = x_{k+1}$ is added to TO-EXPLORE so step 1 returns a cycle. We now consider the case that $i < k+1$. Since x_{i+1} was not added to TO-EXPLORE, the while loop for x_i must have terminated in steps 1,2,3, or 4. If it terminated in steps 1 or 2, then it returned a cycle and we are done. By observation 4.1, it could not have terminated with step 4. Thus, the algorithm terminated with step 3, which means that $u \in \mathcal{A}(x_i) \subseteq A(x_i)$ *before* the insertion of (u, v). But note that the path $x_i, x_{i+1}, ..., x_{k+1} = u$ also existed before the insertion of (u, v), so there was a cycle containing u and x_i before the insertion, which contradicts (u, v) being the first edge to create a cycle.

5 Dynamic Topological Ordering

In this Section we prove Theorem 1.2

We assume for incremental topological sort that the graph remains acyclic at all times; otherwise, we can always use Theorem 1.1 to detect a cycle when it develops. To maintain the vertex ordering, we use a data structure known as an ordered list. In particular, this data structure supports the following operations:

1. Insert(X, Y): given a pointer to X, insert element Y immediately after element X in the ordering.
2. Insert-Before(X, Y): given a pointer to X, insert element Y immediately before element X.
3. Delete(X): given a pointer to X, delete element X from the ordering.
4. Order(X, Y): return whether X or Y comes first in the ordering.

Such a data structure can be implemented in deterministic $O(1)$ time per operation (see [7, 25]). We create an ordered list L which will at all times contain the vertices $v \in V$ as well as $O(n \log^2(n))$ dummy elements defined below. We say that $x \prec y$ if x comes before y in L. Because L maintains a topological ordering, we will always have that if $u \in A(v)$ (with $u \neq v$) then $u \prec v$. We will maintain pointers from each vertex $v \in V$ to its position in L, and vice versa.

High Level Overview The topological sort algorithm is a good deal more technical than the cycle detection one, but the main idea is the same. We start by sampling a set S of $\Theta(\sqrt{n}\log(n))$ vertices. Now, the algorithm always maintains an approximate topological sort by bucketing the vertices according to $|A(v) \cap S|$ and $|D(v) \cap S|$. To establish the approximate position of a vertex v, consider the value pair $(|A(v) \cap S|, -|D(v) \cap S|)$: intuitively, the higher $|A(v) \cap S|$, the more ancestors v has, so the later it should come in the ordering; as a secondary factor, higher $|D(v) \cap S|$ will indicate an *earlier* position in the ordering, since it corresponds to having many descendants. Corresponding to this intuition, we will show a simple proof that given any two vertices v and w, if $(|A(v) \cap S|, -|D(v) \cap S|)$ comes before $(|A(w) \cap S|, -|D(w) \cap S|)$ in a lexicographical ordering, then there cannot be a path from w to v, so we can safely put v before w in our topological ordering.

Our algorithm will create a bucket for each possible value of $(|A(v) \cap S|, -|D(v) \cap S|)$, and will always maintain the invariant that vertices in smaller-valued buckets (again ordered lexicographically) come earlier in the topological sort. It is easy to see that each vertex can change buckets at most $2|S| = \Theta(\sqrt{n}\log(n))$ time, for a total of $O(n^{1.5}\log(n))$ inter-bucket moves in total. Dealing with the incident edges of these moving vertices will lead to a total update time of $O(mn^{1.5}\log(n))$.

Bucketing thus offers a very simple solution for ordering vertices that are already quite different. The hard part lies in maintaining a correct topological ordering within each bucket. Here, we rely on our earlier bounds on the number of similar vertices. In particular, it is not hard to see that if v and w are related and in the same bucket, then by Lemma 3.3 they are S-equivalent. Thus, every time we are forced to rearrange two vertices vertices v and w within the same bucket, we are able to charge those changes to the creation of a new S-equivalent pair. Lemma 3.2 tells us that the total number of such pairs over the entire update sequence is only $O(n^{1.5}\log(n))$, so we have a total of $O(n^{1.5}\log(n))$ intra-bucket moves, and a corresponding total update time of $O(m\sqrt{n}\log(n))$ to deal with the incident edges.

Formal Description As in Cycle Detection, we let S^* be sampled at random from V with probability $11\log(n)/\sqrt{n}$. Observe that according to the Chernoff bound, we have that with high probability $|S^*| \leq 12\log(n)\sqrt{n}$; if this is not the case, we simply resample S^*. We now partition the vertices into buckets, where the bucket of each vertex can change over time.

Definition 5.1 *Let i and j be integers with $0 \leq i, j \leq 12\log(n)\sqrt{n}$.*
Let bucket $B_{i,j} = \{v \in V \mid |A_{S^*}(v)| = i \wedge |D_{S^*}(v)| = 12\log(n)\sqrt{n} - j\}$. We say that $B_{i,j} > B_{i',j'}$ if $i > i'$ or if $i = i'$ and $j > j'$. We let $B(v)$ be the bucket belonging to vertex v.

Observation 5.1 *There are $O(n\log^2(n))$ buckets in total and each $B(v)$ changes at most $O(\sqrt{n}\log(n))$ times over the entire sequence of insertions.*

Observation 5.2 *If $u \in A(v)$ then $B(u) \leq B(v)$. This follows from the fact that $A(u) \subseteq A(v)$ and $D(u) \supseteq D(v)$.*

Observation 5.3 *If u and v are related and $B(u) = B(v)$ then u and v are S^*-equivalent. This follows directly from Lemma 3.3.*

Our algorithm starts with $G = \emptyset$ and for every bucket $B_{i,j}$ in increasing order it inserts a placeholder element $P_{i,j}$ into L. It then inserts all the vertices of V right after $B_{0,12\log(n)\sqrt{n}}$ in an aribtrary order. Our topological sort T will maintain the following invariant:

Invariant 5.1 Bucket Invariant *If $B_{i,j} < B_{i',j'}$ then $P_{i,j}$ will always come before $P_{i',j'}$ in L. If $B(v) = B_{i,j}$ then v always comes after $P_{i,j}$ in L, but before $P_{i',j'}$, where $P_{i',j'}$ is the earliest placeholder after $P_{i,j}$.*

Note that the Bucket Invariant combined with Observation 5.2 ensures that if $B(u) < B(v)$ then we will always have $T(u) < T(v)$, as desired. The more difficult task will be to ensure that the vertices within a bucket are also in the correct topological order.

Definition 5.2 *Consider the insertion of an edge (u, v). For any vertex w, define $A^{\text{OLD}}(w), D^{\text{OLD}}(w)$ to be the sets $A(w)$ and $D(w)$* before *the insertion of (u, v), and define $A^{\text{NEW}}(w), D^{\text{NEW}}(w)$ to be the sets after the insertion. We define L^{OLD} to be the ordered list L before the insertion, and L^{NEW} to be the L after the insertion. We define $B^{\text{OLD}}(w)$ and $B^{\text{NEW}}(w)$ analogously for all $w \in V$. Define sets $UP = \{w \in V \mid B^{\text{NEW}}(w) > B^{\text{OLD}}(w)\}$ and $DOWN = \{w \in V \mid B^{\text{NEW}}(w) < B^{\text{OLD}}(w)\}$. We say that a vertex w is affected by the insertion of (u, v) if $w \in UP \cup DOWN$. For any bucket $B_{i,j}$, define $UP_{i,j} = \{w \in UP \mid B^{\text{NEW}}(w) = B_{i,j}\}$ and $DOWN_{i,j} = \{w \in DOWN \mid B^{\text{NEW}}(w) = B_{i,j}\}$. We say that bucket $B_{i,j}$ is affected by the insertion if either $UP_{i,j}$ or $DOWN_{i,j}$ is non-empty.*

The Algorithm As in cycle detection, the algorithm uses Lemma 3.4 to maintain $A_S(w)$ and $D_S(w)$ for every vertex w in total time $O(m|S^*|) = O(m\sqrt{n}\log(n))$. This information also clearly allows the algorithm to maintain each $B(w)$ in the same total update time. Now, consider the insertion of an edge (u, v). It is not hard to see that since each vertex w explicitly maintains $B(w)$, we can return $\text{UP}_{i,j}$ and $\text{DOWN}_{i,j}$ for each affected bucket $B_{i,j}$ in total time equal to the number of vertices w affected by the insertion of (u, v). We now present pseudocode for how the algorithm handles the insertion of an edge (u, v): see Figure 2. Note that we assume that L^{OLD} was a valid topological ordering of the graph before (u, v) was inserted: the corectness proof will then show that L^{NEW} is valid as well. TO-EXPLORE and TO-CHANGE can both be implemented as simple lists.

Claim 5.1 *The algorithm preserves the Bucket Invariant (5.1).*

Proof. After the insertion of (u, v), Part 1 (steps 1,2,3) move every element to its correct bucket: i.e., if $B^{\text{NEW}}(w) = B_{i,j}$, and $B_{i',j'}$ is the bucket right after $B_{i,j}$, then w is moved between the placeholders $P_{i,j}$ and $P_{i',j'}$. Then Part 2 (steps 4,5) only moves vertices within $B^{\text{NEW}}(v)$ and so does not violate the invariant.

Claim 5.2 *If for some pair of vertices x, y we have $x \in A^{\text{OLD}}(y)$ (and so $L^{\text{OLD}}(x) \prec L^{\text{OLD}}(y)$) then at the end of steps 1,2,3 we have $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(y)$.*

Proof. Since $x \in A^{\text{OLD}}(y) \subseteq A^{\text{NEW}}(y)$, we have by Observation 5.2 that $B^{\text{OLD}}(x) \leq B^{\text{OLD}}(y)$, and $B^{\text{NEW}}(x) \leq B^{\text{NEW}}(y)$. Now, if $B^{\text{NEW}}(y) > B^{\text{NEW}}(x)$ then since by Claim 5.1 steps 1,2,3 restore the Bucket Invariant, we will have $L(x) \prec L(y)$ as desired. Now, if $B^{\text{NEW}}(y) = B^{\text{NEW}}(x) = B_{i,j}$ then we consider three cases.

Case 1: $B^{\text{OLD}}(y) > B^{\text{OLD}}(x)$. In this case, either $y \in \text{DOWN}_{i,j}$ or $x \in \text{UP}_{i,j}$ (or both). But if $x \in \text{UP}_{i,j}$ then step 2 of the algorithm puts x at the very beginning of the bucket (right after $P_{i,j}$) while if $y \in \text{DOWN}_{i,j}$ then step 3 puts y at the very end of the bucket (right before the next placeholder $P_{i',j'}$), so we still have $L(x) \prec L(y)$.

Case 2: $B^{\text{OLD}}(y) = B^{\text{OLD}}(x)$ and $x, y \notin \text{UP} \cup \text{DOWN}$. In this case, x and y both remain in the same bucket and are simply not affected by steps 1,2,3.

Case 3: $B^{\text{OLD}}(y) = B^{\text{OLD}}(x)$ and $x, y \in \text{UP} \cup \text{DOWN}$. In this case, x and y are either both in $\text{UP}_{i,j}$ or both in $\text{DOWN}_{i,j}$. Say that they are both in $\text{UP}_{i,j}$. Then note that step 2 preserves the L-ordering amongst vertices in $\text{UP}_{i,j}$: it inserts each vertex in $\text{UP}_{i,j}$ right after $P_{i,j}$, but since it inserts them in backwards-sorted order on L, by the time x is inserted right after $P_{i,j}$, y will already have been inserted after $P_{i,j}$, and so x will end up before y in the ordering. An analogous argument applies to the case where both x and y are in $\text{DOWN}_{i,j}$, since step 3 also preserves the L-ordering among vertices in $\text{DOWN}_{i,j}$.

Claim 5.3 *If vertex w is added to* TO-CHANGE *in step 4., then the following holds*

1. $w \in D^{\text{OLD}}(v)$ and $w \in D^{\text{NEW}}(u)$

2. $w \notin D^{\text{OLD}}(u)$

3. u and w are S^*-equivalent after the insertion of (u, v)

Proof. The first part of the claim follows from the fact that TO-CHANGE$\subseteq$ TO-EXPLORE, and we initialize TO-EXPLORE with v and then only follow outgoing edges.

For the second part, say for contradiction that $w \in D^{\text{OLD}}(u)$. Then we must have $L^{\text{OLD}}(u) \prec L^{\text{OLD}}(w)$. But then by Claim 5.2, we still have that $L(u) \prec L(w)$ after steps 1,2,3 are executed for the insertion of (u, v). Step 4 only changes L after the entire set TO-CHANGE has been constructed, and w will not be added to TO-CHANGE because of step 4a, which contradicts the assumption of the lemma.

The third part of the claim follows from the fact that w is only added to TO-CHANGE if $B^{\text{NEW}}(w) = B^{\text{NEW}}(v)$ (step 4b.), and we know that $B^{\text{NEW}}(v) = B^{\text{NEW}}(u)$ (step 4.), so by Observation 5.3 w and u are S^*-equivalent.

Running Time Analysis First let us consider the total running time of Part 1 (steps 1,2,3). For step 1, Recall that using Lemma 3.4 we can maintain all the $B(w)$ in total update $O(m\sqrt{n}\log(n))$ over all insertions: since UP and DOWN only consist of vertices for which $B(w)$ changed, we can clearly maintain all the sets UP, DOWN, $\text{UP}_{i,j}$ $\text{DOWN}_{i,j}$ in the same total update time over all insertions. For steps 2 and 3, since our ordered list L can compare any two elements in the list in $O(1)$ time, we can use a comparison sort such as merge sort. The sorting thus requires $O(\log(n))$ time per element in UP $\cup$ DOWN; that is $O(\log(n))$ per vertex w such that $B(w)$ changes as a result of the insertion of (u, v). By Observation 5.1, for any vertex w, $B(w)$ changes at most $O(\sqrt{n}\log(n))$ times over the entire sequence of insertions. Thus, over the entire sequence of insertions, $O(n^{1.5}\log(n))$ elements appear in UP $\cup$ DOWN, leading to a total sorting time of $O(n^{1.5}\log^2(n))$. Similarly, since L implements Insert(X, Y) and Insert-Before(X, Y) in $O(1)$ time, the total time to insert elements after $P_{i,j}$ (step 2) or before $P_{i',j'}$ (step 3) is the total number of elements to appear in UP $\cup$ DOWN over the entire sequence of insertions, which is $O(n^{1.5}\log(n))$.

We now consider Part 2 (steps 4 and 5). Note that other than the initial element v, each element $z \in$ TO-EXPLORE was put there in step 4c, and in particular can be associated with some edge (w, z) where w was added to TO-CHANGE. Thus, steps 4a and 4b can be charged to the work in 4c. In step 4c the algorithm spends $O(m/n)$ time per vertex w added to TO-CHANGE; recall from Assumption 1.1 that every vertex w has degree $O(m/n)$. In step 4d merge sort requires at most $O(\log(n))$ time

Figure 2: Algorithm for inserting an edge (u, v) to G.

```
Topological Ordering: Insert $(u, v)$ in $G$
\\ Part 1: first we move elements that changed buckets
1. Find UP, DOWN, and all non-empty sets $\mathrm{UP}_{i,j}$, $\mathrm{DOWN}_{i,j}$
2. For each non-empty set $\mathrm{UP}_{i,j}$
   Sort elements in $\mathrm{UP}_{i,j}$ in increasing order according to $L$.
• For each $w \in \mathrm{UP}_{i,j}$ in decreasing order:
        Delete $w$ from its current place in the ordering.
        Insert $w$ right after $P_{i,j}$ in $L$          \\ Maintains sorted order in $\mathrm{UP}_{i,j}$
3. For each non-empty set $\mathrm{DOWN}_{i,j}$
   Sort elements in $\mathrm{DOWN}_{i,j}$ in increasing order according to $L$.
• For each $w \in \mathrm{DOWN}_{i,j}$ in increasing order:
        Delete $w$ from its current place in the ordering.
        Let $B_{i',j'}$ be the bucket right after $B_{i,j}$.
        Insert $w$ right before $P_{i',j'}$ in $L$          \\ Maintains sorted order in $\mathrm{DOWN}_{i,j}$.
\\ Part 2: now we fix up the bucket containing $u$ and $v$.
4. If $B^{\mathrm{NEW}}(u) = B^{\mathrm{NEW}}(v)$
   Initialize TO-EXPLORE= $\{v\}$
   Initialize TO-CHANGE= $\emptyset$          \\ All vertices in TO-CHANGE will be moved.
• While TO-EXPLORE $\neq \emptyset$
        $w$ = TO-EXPLORE.pop-arbitrary-element
        4a. If $L(u) \prec L(w)$
            do nothing          \\ $u$ and $w$ already correctly ordered.
        4b. Else If $B^{\mathrm{NEW}}(w) \neq B^{\mathrm{NEW}}(u)$
            do nothing          \\ $u$ and $w$ already correctly ordered by bucket structure
      • 4c. Else
            Add $w$ to TO-CHANGE
          • for every edge $(w, z)$
                Add $z$ to TO-EXPLORE
   4d. Sort the elements in TO-CHANGE in increasing order according to the ordering of $L$.
   4e. For each $w \in$ TO-CHANGE in decreasing order insert $w$ right after $u$ in $L$
          \\ Note: Case 4e preserves the $L$-order of vertices in TO-CHANGE.
5. Else                    \\ $B^{\mathrm{NEW}}(u) \neq B^{\mathrm{NEW}}(v)$
   Do Nothing              \\ $u$ and $v$ already correctly ordered by bucketing structure.
```

per vertex in TO-CHANGE and in step 4e the algorithm spends $O(1)$ for each vertex in TO-CHANGE. Thus, in total the algorithm spends $O(\log(n) + d)$ time for each vertex added to TO-CHANGE. First off, note that if w is added to TO-CHANGE during the insertion of some (u, v), then for all future edge insertions we will have $w \in D(u)$, and so by Claim 5.3 (part 2) w will never again be added to TO-CHANGE for the insertion of some (u, v'). Thus the total number of vertices ever added to some TO-CHANGE over the entire sequence of insertions is exactly the number of pairs (u, w) such that w is added to TO-CHANGE during the course of some insertion (u, v). But note that for any such pair (u, w), we have by Claim 5.3 that u and w are sometime-S^*-equivalent, and so by Corollary 3.1 the total number of such pairs is $O(n^{1.5} \log(n))$ with high probability. Thus the total work done in step 4 is $O((\log(n) + (m/n))n^{1.5} \log(n)) = O(m\sqrt{n} \log(n) + n\sqrt{n} \log^2(n))$.

Correctness Analysis Consider the insertion of some edge (u, v). We assume that L was a valid topological ordering before the insertion, and want to show that it is also a valid topological ordering afterwards. Consider any pair of vertices (x, y). We want to show that if $x \in A^{\text{NEW}}(y)$ then $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(y)$. Note that if $B^{\text{NEW}}(x) \neq B^{\text{NEW}}(y)$ then they will be correctly ordered by the Bucket Invariant, which by Claim 5.1 is preserved by the algorithm. We can thus assume for the rest of the proof $B^{\text{NEW}}(x) = B^{\text{NEW}}(y)$. We now consider two main cases

Case 1: $x \in A^{\text{OLD}}(y)$. In this case we had $L^{\text{OLD}}(x) \prec L^{\text{OLD}}(y)$.

- Case 1a: $B^{\text{NEW}}(u) \neq B^{\text{NEW}}(v)$. In this case step 4 will simply not be executed; on the other hand, Claim 5.2 guarantees that at the end of steps 1,2,3 we have $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(y)$, as desired.
- Case 1b: $B^{\text{NEW}}(x) = B^{\text{NEW}}(y) \neq B^{\text{NEW}}(v)$. In this case neither x nor y will end up in TO-CHANGE (step 4b), so their position in L will be unaffected by step 4; on the other hand, Claim 5.2 guarantees that at the end of steps 1,2,3 we have $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(y)$, as desired.
- Case 1c: $B^{\text{NEW}}(x) = B^{\text{NEW}}(y) = B^{\text{NEW}}(u) = B^{\text{NEW}}(v)$. Note that x ends up in TO-CHANGE if and only if $x \in D^{\text{OLD}}(v)$ and $L^{\text{OLD}}(x) \prec L^{\text{OLD}}(u)$, and that y is added to TO-CHANGE if and only if $y \in D^{\text{OLD}}(v)$ and $L^{\text{OLD}}(y) \prec L^{\text{OLD}}(u)$.
  - Case 1c.1: Neither x nor y end up in TO-CHANGE. In this case by Claim 5.2 we have $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(y)$.
  - Case 1c.2: y is added to TO-CHANGE but x is not. In this case, since y ends up in TO-CHANGE we had $L^{\text{OLD}}(y) \prec L^{\text{OLD}}(u)$, and so by the assumption of Case 1, $L^{\text{OLD}}(x) \prec L^{\text{OLD}}(y) \prec L^{\text{OLD}}(u)$. But then step 4e moves y after u in the ordering, whereas the position of x stays the same (x is not in TO-CHANGE), so $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(u) \prec L^{\text{NEW}}(y)$, as desired.
  - Case 1b.3: x is added to TO-CHANGE but y is not. Note that since x is added to TO-CHANGE we must have $x \in D^{\text{OLD}}(v)$, so since $y \in D^{\text{OLD}}(x)$ we have $y \in D^{\text{OLD}}(v)$. Thus the only possible reason y is not added to TO-CHANGE is that $L^{\text{OLD}}(u) \prec L^{\text{OLD}}(y)$. But since x is in TO-CHANGE it is added directly after u in L, so it is inserted between u and y and we have $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(y)$.
  - Case 1b.4: Both x and y are added to TO-CHANGE. In this case $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(y)$ because step 4e preserves the L^{OLD}-order of verices in TO-CHANGE. To see this, note that the vertices of TO-CHANGE are inserted directly after u in decreasing order of L^{OLD}, so by the time x is inserted into L directly after u, y will have already been inserted after u, so x will end up between u and y in L^{NEW}.

Case 2: $x \notin A^{\text{OLD}}(y)$. The only way that the insertion of the edge (u, v) could lead to $x \in A^{\text{NEW}}(y)$ but $x \notin A^{\text{OLD}}(y)$ is if the graph G contains a path $x \rightsquigarrow u \to v \rightsquigarrow y$; that is, if $x \in A^{\text{OLD}}(u) \subseteq A^{\text{NEW}}(u)$ and $y \in D^{\text{OLD}}(v) \subseteq D^{\text{NEW}}(v)$. Now, note that x is not added to TO-CHANGE because otherwise by Claim 5.3 we would have $x \in D^{\text{OLD}}(v)$ and there would be a cycle $u \to v \rightsquigarrow x \rightsquigarrow u$. Thus, since $L^{\text{OLD}}(x) \prec L^{\text{OLD}}(u)$ (because $x \in A^{\text{OLD}}(u)$) we must have by Claim 5.2 that $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(u)$.

- Case 2a: $B^{\text{NEW}}(u) \neq B^{\text{NEW}}(v)$. In this case, since $u \in A^{\text{NEW}}(v)$, Observation 5.2 implies that $B^{\text{NEW}}(u) < B^{\text{NEW}}(v)$. But since $x \in A(u)$ we have $B^{\text{NEW}}(x) \leq B^{\text{NEW}}(u)$ and since $y \in D^{\text{NEW}}(v)$ we have $B^{\text{NEW}}(v) \leq B^{\text{NEW}}(y)$. Thus $B^{\text{NEW}}(x) < B^{\text{NEW}}(y)$, so since the algorithm preserves the Bucket Invariant in L (Claim 5.1) we have $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(y)$, as desired.
- Case 2b: $B^{\text{NEW}}(u) = B^{\text{NEW}}(v)$ and y is not added to TO-CHANGE. In this case we

must have had $L^{\text{OLD}}(u) \prec L^{\text{OLD}}(y)$, and so $L^{\text{NEW}}(u) \prec L^{\text{NEW}}(y)$ (since neither u nor y end up in TO-CHANGE); thus $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(u) \prec L^{\text{NEW}}(y)$ so we are done.

- Case 2c:
  $B^{\text{NEW}}(u) = B^{\text{NEW}}(v)$
  and y is added to TO-CHANGE. In this case step 4e of the algorithm ensures that y is placed after u in L, so $L^{\text{NEW}}(u) \prec L^{\text{NEW}}(y)$ so we again have $L^{\text{NEW}}(x) \prec L^{\text{NEW}}(u) \prec L^{\text{NEW}}(y)$.

Appendix

A Justification of Assumption 1.1

In what follows we show a reduction from the general case (vertices have arbitrary degree) to the bounded-degree case in the assumption.

Let $d = \lceil |E|/|V| \rceil$. In the static setting the assumption can be easily obtained by replacing all the outgoing edges from v by a balanced d-tree (a balanced tree with degree d where v is the root of the tree and all edges of the tree are directed from the root) where each leaf in this tree is "responsible" for d of the outgoing edges of v, that is, each leaf has d outgoing edges to d (different) neighbors of v. A similar process is done for the incoming edges of v for every node $v \in V$. It is not hard to verify that the number of new nodes created is proportional to the number of edges divided by d, that is, the number of new nodes is $O(m/d) = O(n)$. In addition, every two original nodes u and v that have a directed path in G, also have a directed path in the modified graph.

In the dynamic setting we use a very similar reduction. There are two slight technical issues. The first is that since m refers to the number of edges in the final graph, we do not know $d = \lceil m/n \rceil$ in advance. The second is that even if we did know d in advance, for any given vertex v we do not know the final in-degree and out-degree of v in advance, so we cannot create the in d-tree and out d-tree of v in advance. For example, if v has d^5 neighbors then we use a different tree than if it has d^6. We overcome this, by using a slightly different tree that is easier to construct dynamically as the degree gets larger but has similar properties to the balanced d-trees.

Let us for now assume that we know d in advance. The only issue left is thus that we do not know in- and out-degrees in advance. We explain the dynamic process for the outgoing edges for a node $v \in V$. A similar process is done for the incoming edges to v and for every node $v \in V$. For every d outgoing edges added from v for the first d^2 outgoing edges, add a new node v' and connect it by an edge from v and add these outgoing d edges from v' rather than from v. That is, when a first edge from v is added, we add a new node v', connect it from v by an edge and add the new edge from v' rather than from v. All the next $d-1$ edges added from v are added from v' instead. After d edges are added to v we create a new node $v'_2$for the next d edges and so on. After d^2 edges, we can no longer continue in this fashion because the degree of v would get too big. We thus add an additional node v'' and connect it by an edge from v (this is the last outgoing edge we will add from from v). For the node v'' we add d additional nodes and connect edges from v'' to them; each such node will take care of d^2 additional edge insertions from v (in exactly similar way we did for v, i.e., for every d edges add another node and connect and so on). This takes care of the next d^3 edge insertions from v. We now add an additional node v''' with an edge from v'' (the last one from v'') and create new d^2 nodes, connect the first d of these nodes by an edge from v''' and for each such node connect it to additional d nodes where each such node will take care of d edge insertions from v. This will take care of next d^4 edge insertions from v. We continue with this process by adding now new d^3 new nodes and so on. It is not hard to verify that this process satisfy all requirements.

We now consider the case where we do not know $d = \lceil m/n \rceil$ in advance. Let m^* refer to the current number of edges in the graph and let $d^* = \lceil m^*/n \rceil$. (Recall that we assume a model where all the vertices are given in advance. If do not know the number of vertices n either a slightly more complicated doubling argument can be used.) Note that we always have $d^* \leq d$. We now use a standard doubling argument. Every time d^* increases to 2^i for some i we register this change. Let d^*_O be the old value of d^* (O for old), and let d^*_N be the new value; note that $d^*_N = 2d^*_O$. Now, as more edges are inserted, we slowly turn the old d^*_O-tree into a new d^*_N-tree. We do this bottom up. Each internal leaf in the d^*_O tree now has allowed degree d^*_N, so the first bunch of insertions can simply be added to the existing leaves. We then move one layer up. The vertices at height 1 currently have at most d^*_O leaves, but are now allowed d^*_N. The next new bunch of insertions can thus be handled by creating a new internal leaf from the vertices of height 1, until all vertices at height 1 have d^*_N internal leaves. We then move up to height 2 and so on. It is not hard to check that the new tree satisfies all requirements.

Acknowledgement: We are deeply indebted to Haim Kaplan for many discussions on this topic.

References

[1] Deepak Ajwani and Tobias Friedrich. Average-case analysis of incremental topological ordering. *Discrete Applied Mathematics*, 158(4):240 – 250, 2010. 6th Cologne/Twente Workshop on Graphs and Combinatorial Optimization (CTW 2007).

[2] Deepak Ajwani, Tobias Friedrich, and Ulrich Meyer. An o(n2.75) algorithm for incremental topological ordering. *ACM Trans. Algorithms*, 4(4):39:1–39:14, August 2008.

[3] Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 32–42, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[4] Michael A. Bender, Jeremy T. Fineman, and Seth Gilbert. A new approach to incremental topological ordering. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 1108–1115, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[5] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, December 2015.

[6] Edith Cohen, Amos Fiat, Haim Kaplan, and Liam Roditty. A labeling approach to incremental cycle detection. *CoRR*, abs/1310.8381, 2013.

[7] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 365–372, New York, NY, USA, 1987. ACM.

[8] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E. Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algorithms*, 8(1):3:1–3:33, January 2012.

[9] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, FOCS, pages 146–155, 2014.

[10] Monika Rauch Henzinger and Mikkel Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Struct. Algorithms*, 11(4):369–379, 1997.

[11] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[12] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In *Proc. of 23rd ESA*, pages 742–753, 2015.

[13] Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48(2-3):273–281, 1986.

[14] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proc. of 24th SODA*, pages 1131–1142, 2013.

[15] Irit Katriel and Hans L. Bodlaender. Online topological ordering. *ACM Trans. Algorithms*, 2(3):364–379, July 2006.

[16] Telikepalli Kavitha and Rogers Mathew. Faster algorithms for online topological ordering. *CoRR*, abs/0711.0251, 2007.

[17] D.E. Knuth. *The Art of Computer Programming: Fundamental algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley Publishing Company, 1973.

[18] Hsiao-Fei Liu and Kun-Mao Chao. A tight analysis of the katrielbodlaender algorithm for online topological ordering. *Theoretical Computer Science*, 389(1):182 – 189, 2007.

[19] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59(1):53 – 58, 1996.

[20] University of Newcastle upon Tyne. Computing Laboratory, D.E. Knuth, and J.L. . *A structured program to generate all topological sorting arrangements*. Technical report series. University of Newcastle upon Tyne, 1974.

[21] David J. Pearce and Paul H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *J. Exp. Algorithmics*, 11, February 2007.

[22] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM JOURNAL ON COMPUTING*, 1(2), 1972.

[23] Mikkel Thorup. On RAM priority queues. *SIAM J. Comput.*, 30(1):86–109, 2000.

[24] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proc. of 24th SODA*, pages 1757–1769, 2013.

[25] Jack Zito, Heraldo Memelli, Kyle G. Horn, Irene C. Solomon, and Larry D. Wittie. Application of a "staggered walk" algorithm for generating large-scale morphological neuronal networks. *Comp. Int. and Neurosc.*, 2012:876357:1–876357:8, 2012.