- Incremental Cycle Detection and Topological Ordering -

Martin Costa

Supervised by
Dr. Sayan Bhattacharya

Year 3

Discrete Mathematics

University of Warwick

March 2021

Abstract

Due to the fundamental nature of cycle detection and topological ordering in directed graphs, there has been a long and important line of work in dynamic algorithms focusing on the incremental variants of these problems, spanning many decades. In this project, we present 4 new technical results and frameworks for the design and analysis of incremental cycle detection and topological ordering algorithms. Our first result is a general framework for the design of non-local divide and conquer algorithms for incremental topological ordering based on our definition of *semi-topological partitions*. Almost 10 years ago, it was proven that no *local algorithm* for incremental topological ordering is capable of obtaining near-linear amortized update time under *adversarial edge insertions*; we show that this proof breaks down in the *random order model*. Motivated by this observation, we design a framework for the analysis of a simple local algorithm, obtaining partial results on open problems, and extend this framework to yield a reduction from an open problem in this active research area to a conjecture arising naturally in this setting. These frameworks can be extended directly but may also be of independent interest to the reader.

Keywords: Graph Algorithms, Dynamic, Online, Incremental, Cycle Detection, Topological Ordering, Random Order Model.

Acknowledgements

I would like to thank my supervisor, Dr. Sayan Bhattacharya, for all of his support and guidance throughout the last year. Without his help, a project of this quality would not have been possible. I am greatly indebted to Sayan for everything he was taught me and for enabling me to start doing research in theoretical computer science.

Contents

Chapter 1

Introduction

In this chapter, we shall start off by introducing the problems of *incremental cycle detection* and *incremental topological ordering*, followed by useful notation that will be used throughout this report. We then give formal descriptions of the problems, including relevant definitions, along with a survey of the most important and relevant results on the topic. Finally, we give an overview of what will be covered in the different chapters of this report.

1.1 The Problems

This section contains a brief overview of the problems as well as important notation that will be used throughout the report.

1.1.1 Incremental Cycle Detection

Cycle detection in directed graphs is one of the classic problems studied in algorithmic graph theory. The problem is simple, given some directed graph G, return whether or not G is acyclic. The solution is straightforward and can be computed in $\mathcal{O}(n+m)$ time, where n is the number of *nodes* in G and m is the number of *edges* in G.

In this project we will be considering a dynamic version of this problem, where we are given a graph that initially has no edges (and hence no cycles) which is updated over time by a sequence of edge insertions. After each edge is inserted, we must return whether or not the updated graph is acyclic. This problem is referred to as *incremental cycle detection*.

1.1.2 Incremental Topological Ordering

We can can come up with a dynamic version of the *topological ordering* problem in a similar way, where we are given a graph that initially has no edges (and hence admits a topological ordering) which is updated over time by a sequence of edge insertions. After each edge is inserted, we must return a topological ordering of the updated graph, and a warning if the updated graph does not admit a topological ordering. This problem is referred to as *incremental topological ordering*.

The fact that a graph G admits a topological ordering of it's nodes if and only if G is a *directed acyclic graph* (DAG) is also one of the standard results in algorithmic graph theory. In fact, the most well known algorithm for cycle detection in directed graphs relies on this equivalence, and determines the presence of a cycle in G by checking whether or not G admits

a topological ordering. The problem of cycle detection can be reduced to the problem of topological ordering, as long as we ensure that when no topological ordering exists we return the appropriate warning instead of returning an ordering which is not a topological ordering.

It should then be no surprise that the problems of incremental cycle detection and incremental topological ordering are also deeply connected, and that incremental cycle detection can also be reduced to incremental topological ordering, with the same caveat as in the static case. Because of this, every well known algorithm for incremental cycle detection relies on an algorithm for incremental topological ordering.

1.1.3 Notation

Let $G = (V, E)$ be a directed graph with $n = |V|$ and $m = |E|$. Given some countable set S, denote by $\mathscr{S}_S$ the space of all sequences over S where each element of S appears exactly once. Given some $\mathcal{S} \in \mathscr{S}_S$ we denote the i^{th} element in the sequence by $\mathcal{S}_i$, i.e. $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, ...)$. The sequences in $\mathscr{S}_S$ are finite if and only if S is finite. Notice that $\mathscr{S}_E$ is the set of all possible sequences of edge insertions that can be made to construct the graph G, we will call the elements of $\mathscr{S}_E$ the *insertion sequences* of G. Define a correspondence, $\sim$, between *total orderings of* V and elements of $\mathscr{S}_V$ such that given some total ordering of V, $\prec$, and some $(u_1, ..., u_n) \in \mathscr{S}_V$, we have $\prec \sim (u_1, ..., u_n)$ if and only if $u_1 \prec ... \prec u_n$. This correspondence clearly gives us a bijection between the total orderings of V and elements of $\mathscr{S}_V$. For the purposes of this report we shall consider a total ordering and its corresponding sequence to be essentially the same, and shall abuse this notation throughout the report. We call the elements of $\mathscr{S}_V$ the *orderings* of G. Given some $\mathcal{E} \in \mathscr{S}_E$, denote by $G_t^{\mathcal{E}}$ the graph $(V, \{\mathcal{E}_1, ..., \mathcal{E}_t\})$, i.e. the graph $(V, \varnothing)$ after inserting the first t edges in $\mathcal{E}$. When considering some incremental topological ordering algorithm, $\mathcal{A}$, some insertion sequence $\mathcal{E}$ and some ordering $\prec$, we frequently denote by $\prec_t$ the topological ordering computed by $\mathcal{A}$ starting with the initial ordering $\prec$ after the insertion of the first t edges from $\mathcal{E}$.

Given some subgraph H of G and $u \in V(H)$, we denote by $reach_H(u)$ the *descendants* of u contained in H and by $reach_H^{-1}(u)$ the *ancestors* of u contained in H, i.e. $reach_H(u) = \{v \in V(H) \mid \text{there exists a path from } u \text{ to } v \text{ in } H\}$ and $reach_H^{-1}(u) = \{v \in V(H) \mid u \in reach_H(v)\}$. Given some $X \subseteq V(H)$ we let $reach_H(X) = \bigcup_{u \in X} reach_H(u)$ and $reach_H^{-1}(X) = \bigcup_{u \in X} reach_H^{-1}(u)$. Occasionally we will simplify the subscript in this notation to make things easier to read (for example by changing $reach_{G_t}(u)$ to $reach_t(u)$).

1.2 Survey

This section contains a formal introduction to incremental cycle detection and topological ordering and a survey of the more important and relevant results on the topic, as well as a brief overview of some of the literature which was read throughout the course of the project.

1.2.1 Formal Descriptions

Using the notation given in section 1.1.3, the problem of incremental cycle detection is defined formally as follows. Let $G = (V, E)$ be a directed graph, and let $\mathcal{E}$ be some insertion sequence of G. We are given the graph $G_0^{\mathcal{E}} = (V, \varnothing)$ which contains no edges. Subsequently, there are m time-steps. At the t^{th} time step, the edge $\mathcal{E}_t$ is inserted into the graph, giving us the updated graph $G_t^{\mathcal{E}}$, we refer to this as an *update* or an *edge insertion*. After each update we must return whether the updated graph contains a cycle. The time taken to report whether the updated

graph contains a cycle after the edge insertion is called the *update time*. The *total update time* is the sum of the update times for each edge insertion.

The problem of incremental topological ordering can be defined in a similar way. For completeness, we will now give the definition of a topological ordering.

Definition 1.2.1 (Topological Ordering)**.** *Let $G = (V, E)$ be a DAG and $\prec$ be some ordering of G. Then $\prec$ is a **topological ordering** of G if for all $(u, v) \in E$ we have $u \prec v$.*

Incremental topological ordering is defined in almost the same way as incremental cycle detection. The only difference is that after reporting whether or not the updated graph has a cycle, if the graph is acyclic, we also return a topological ordering of the graph. Note that a directed graph admits a topological ordering if and only if it is acyclic. The *update time* and *total update time* are defined in the same way as for incremental cycle detection.

1.2.2 Special Cases

We will now look at some special cases and settings which are worth taking note of.

Offline Setting

We can see that not having access to the whole insertion sequence during the run of the algorithm is what makes incremental cycle detection non-trivial. As briefly described in [BK20], in the *offline setting*, we can perform a binary search to find the smallest value of t for which $G_t^{\mathcal{E}}$ is not acyclic by using a cycle detection algorithm which runs in $\mathcal{O}(m)$ time. This tells us that, in the offline setting, we can solve the problem in $\mathcal{O}(m \log m)$ time. Since this algorithm is asymptotically optimal, it follows that any dynamic algorithm for this problem has total update time $\Omega(m \log m)$. For this reason, we would like to find an algorithm for this problem with total update time $\mathcal{O}(m \operatorname{polylog} m)$, since then the bounds for the online and offline setting would match. However, there are currently no known algorithms for incremental cycle detection with such a total update time. The same arguments apply for incremental topological ordering.

Cycle Detection in Dense Graphs

A very important result in the topic of incremental cycle detection is the fact that the problem is "well understood" for dense graphs (i.e. graphs where $m = \Omega(n^2)$). In the work of Bender et al. [BFG09], they construct an incremental cycle detection algorithm with a total update time of $\mathcal{O}(n^2 \log n)$. This means that, for any dense graph, this algorithm gives near-linear (in the number of edges in the graph) total update time.[1]

Cycle Detection in Sparse Graphs

In contrast to the result given above for dense graphs, there are currently no known algorithms for incremental cycle detection that give us near-linear total update time for sparse graphs. It is currently known that there exists a randomized algorithm with an expected total update time of $\tilde{\mathcal{O}}(m^{4/3})$; this was proven by Bhattacharya et al. in [BK20].[2]

[1] Since polylogarithmic factors grow very slowly ($\operatorname{polylog} n = o(n^{\varepsilon})$ for all $\varepsilon > 0$), we say that f is near-linear if $f(n) = \mathcal{O}(n \operatorname{polylog} n)$

[2] The notation $\tilde{\mathcal{O}}$ is used to suppress poly-logarithmic factors, i.e. $\tilde{\mathcal{O}}(f(n)) = \mathcal{O}(f(n) \operatorname{polylog} f(n))$

1.2.3 Literature Review

As pointed out by [BK20], due to the fundamental nature of cycle detection and topological ordering in directed graphs, there has been a long and important line of work in dynamic algorithms focusing on the incremental variants of these problems, spanning many decades [MNR96, PK06, AFM06, BFG09, AF10, HKM+12, CFKR13, BFGT16, BC18, BK20].

The rest of this section contains a brief overview of some of the relevant content contained in 4 important papers on the topic of incremental cycle detection and topological ordering, [BFG09, HKM+12, BC18, BK20]. These papers collectively contain many of the best performing algorithms for incremental cycle detection.

[BFG09]

In the work of Bender et al., the authors give an algorithm with a total update time of $\mathcal{O}(n^2 \log n)$. This algorithm (like all of the incremental cycle detection algorithms we will discuss in this report) relies on maintaining topological ordering as edges are inserted. In this paper they present a new approach to maintaining the ordering, which assigns each node a label which can be used to determine the ordering instead of placing nodes in an ordered list. These labels can be updated efficiently as edges are inserted, leading to an efficient algorithm that performs very well on dense graphs, obtaining near-linear total update time in the number of edges. There are, however, sparse graphs with $m = \mathcal{O}(n)$ which lead to a total update time of $\Omega(n^2)$ using this algorithm. So this algorithm is not optimal on graphs which are not dense and performs poorly on sparse graphs.

[HKM+12]

In the work of Haeupler et al., the authors present 2 new online algorithms for incremental topological ordering. They present a two-way search algorithm which has total update time $\mathcal{O}(m^{3/2})$ and is tight. They also present an algorithm which performs well on sparse graphs and has total update time $\mathcal{O}(n^{5/2})$, even though this bound may be far from tight. This algorithm is completely different from the algorithm presented in [BFG09], and is not particularly relevant to this project. They identify a class of incremental topological ordering algorithms which they refer to as *local algorithms* (see Definition 2.3.2), and show that *all* local algorithms must have a worst case total update time of at least $\Omega(n\sqrt{m})$. This result is very relevant to this project and we will return to these ideas in later chapters.

[BC18]

In the work of, Bernstein et al., the authors introduce a new technique which improves upon the state of the art for a large spectrum of graph sparsity; They give randomized algorithms for both incremental topological ordering and incremental cycle detection with an expected total update time of $\tilde{\mathcal{O}}(m\sqrt{n})$. Suppose we have a DAG, G, and an edge (u, v) is inserted into G, if we want to check if whether the updated graph is acyclic, we could search the graph starting from v and check whether u is discovered, but this algorithm requires $\Omega(m)$ time in the worst case. At a high level, the algorithm presented by Bernstein et al. uses this idea but prunes the vertices that are visited during the search starting from v, which significantly reduces the time taken to perform the search. They do this by introducing the notion of *S-equivalent vertices*, a relaxed version of vertex equivalence (we say vertices u and v in G are equivalent if they are related, have the same ancestors, and have the same descendants), and argue that while exploring the S-related vertices of v, either there is a cycle with high probability or the search

does not take a long time. They then improve this update time by arguing that, on average, a vertex is not S-equivalent to many different vertices throughout the run of the algorithm, and proceeding to design an algorithm which charges the work it does to the number of such S-equivalent pairs of vertices.

[BK20]

In the work of Bhattacharya et al., the authors give a randomized algorithm with an expected total update time of $\tilde{\mathcal{O}}(m^{4/3})$. They achieve this by combining the algorithm presented in [BC18] and the balanced search framework of [HKM+12]. The algorithm given in this paper gives the best known update time for sparse graphs.

1.3 Overview

This report will be split up into 5 further chapters and a conclusion. The next chapter will lay the foundations for the rest of this report by introducing the ideas and problems that were explored throughout the project. The rest of the report will be dedicated to presenting the various original results and frameworks that came out of this project, including a framework for designing divide and conquer algorithms for incremental topological ordering and a framework leading to progress and partial results in currently open problems. The following gives a brief overview of the content contained in each chapter.

- **Chapter 2.** An introduction to the *recourse* of an algorithm (an important metric on incremental topological ordering algorithms). Definitions of *local algorithms*, the *simple one-way search algorithm* $\mathcal{A}_1$ and the *simple greedy two-way search algorithm* $\mathcal{A}_2$; accompanied by an introduction to the random-order model and the conjectures that $\mathcal{A}_1$ and $\mathcal{A}_2$ have low recourse under random order arrival, as well as experimental evidence to support this.

- **Chapter 3.** A description of our framework for divide and conquer incremental topological ordering algorithms, based on our definition of *semi-topological partitions* and their various properties. The construction of a non-local algorithm using our dynamic *meta-tree* data structure at the core of the framework.

- **Chapter 4.** A proof that $\mathcal{A}_2$ has total recourse $\mathcal{O}(n\sqrt{m})$ under adversarial arrival and the construction of sparse graphs (with corresponding insertion sequences and initial orderings) that yield $\Omega(n\sqrt{m})$ total recourse on all local algorithms under adversarial arrival; followed by a detailed analysis that shows how the latter does not hold under random-order arrival.

- **Chapter 5.** The construction of our framework using the properties of $\mathcal{A}_1$ combined with our definition of *activation sequences* and a function Γ defined on the space of such sequences. A characterization of the recourse of $\mathcal{A}_1$ on trees in terms of Γ yielding a proof that the expected total recourse of trees under adversarial arrival is $\mathcal{O}(n \log n)$ with expectation taken over initial orderings. We also provide concentration bounds to show that this result holds with high probability.

- **Chapter 6.** An extension of the activation sequence framework, directly generalizing results from the previous chapter to give an equivalent characterization of recourse of $\mathcal{A}_1$ on any DAG in terms of Γ. This gives a reduction from an open conjecture to a statement resulting naturally from this framework.

Chapter 2

The Recourse Metric

In this chapter, we shall introduce a metric for analysing the performance of dynamic algorithms for incremental topological ordering which we will be focusing on throughout the rest of this report. We shall refer to this metric as the *recourse* of an algorithm. This metric will allow us to work in a significantly simplified setting, where we do not have to (directly) concern ourselves with update time or any concrete implementations of the algorithms we work with. We will then give descriptions of two simple dynamic algorithms that are currently candidates to solve an open problem in this setting.

2.1 The Definition of Recourse

Let $G = (V, E)$ be a DAG. Suppose we have an algorithm for incremental topological ordering, $\mathcal{A}$, which given an insertion sequence $\mathcal{E} \in \mathscr{S}_E$ maintains an ordering $\prec$ as the edges are inserted. We define a *node movement* to be the operation of changing the position of a single node in the ordering, this can be thought of as removing any node and placing it back in at any position. Assume that the algorithm $\mathcal{A}$ can only affect the ordering by performing such node movements, we formalize the notion of *recourse* as follows.

Definition 2.1.1 (Recourse)**.** *The* ***recourse*** *of $u \in V$ caused by $\mathcal{E}_t$ during the run of $\mathcal{A}$ on G with insertion sequence $\mathcal{E}$ starting with initial ordering $\prec$ is*

$$rec_u(\mathcal{E}, \prec)_t = \begin{cases} 1 & \mathcal{A} \text{ moves } u \text{ while handling the insertion of } \mathcal{E}_t \\ 0 & otherwise \end{cases}$$

- *The* ***recourse*** *of u during the run of $\mathcal{A}$ on G with insertion sequence $\mathcal{E}$ and initial ordering $\prec$ is $rec_u(\mathcal{E}, \prec) = \sum_{t=1}^{m} rec_u(\mathcal{E}, \prec)_t$.*
- *The* ***recourse*** *caused by $\mathcal{E}_t$ during the run of $\mathcal{A}$ on G with insertion sequence $\mathcal{E}$ starting with initial ordering $\prec$ is $rec(\mathcal{E}, \prec)_t = \sum_{u \in V} rec_u(\mathcal{E}, \prec)_t$.*
- *The* ***total recourse*** *caused by the run of $\mathcal{A}$ on G with insertion sequence $\mathcal{E}$ and initial ordering $\prec$ is $rec(\mathcal{E}, \prec) = \sum_{u \in V} rec_u(\mathcal{E}, \prec)$. We will often just refer to this as "the recourse of $\mathcal{A}$" for simplicity when everything is implicit from the context.*

Less formally, the recourse of the algorithm caused by an edge insertion is the amount of nodes the algorithm moves while handling that insertion. Similarly, the recourse of a node is

simply the amount of times that node is moved by the algorithm. Intuitively, we can see that having low recourse during an update would mean that the ordering 'doesn't have to change very much' to obtain a new topological ordering, and having high recourse would tell us the opposite. It should be clear (at least intuitively) that an algorithm that has low recourse would be better than an algorithm with high recourse.

Throughout this report, whenever we say that we have a 'high level description of some algorithm $\mathcal{A}$' or a 'high level algorithm $\mathcal{A}$', this means that we have a sufficiently detailed description of $\mathcal{A}$ such that we can calculate its recourse given any input, but not necessarily its update time. This will be important since the term 'high level algorithm' is generally quite vague, since its really just (roughly) saying that the description of the algorithm in question is in itself somewhat vague and not concrete enough to implement directly. However, for our purposes, it will have said formal meaning.

2.2 Why Recourse?

This notion of recourse is quite a natural metric to consider for dynamic algorithms of this form. We can see that it gives us a lower bound on their update time without having to consider any concrete implementations, since each movement will be performed individually by the algorithm by assumption. Hence, if the worst case total recourse of $\mathcal{A}$ is $\Omega(f(n))$ then the worst case total update time of $\mathcal{A}$ is also $\Omega(f(n))$.

However, the notion of recourse isn't only useful for dynamic algorithms that have this restricted form, where node movements have to be performed one at a time. Informally speaking, we can also see that given any incremental topological ordering algorithm, $\mathcal{A}$, if $\mathcal{A}$ can be modified to create an algorithm $\mathcal{A}^*$, that can only affect the ordering with node movements, such that $\mathcal{A}^*$ doesn't perform worse than $\mathcal{A}$, then $\mathcal{A}^*$ having a large recourse will give us that $\mathcal{A}$ will also have a large update time. While this statement is quite informal and vague, it should give some intuition as to why recourse is a natural metric to consider in an attempt to simplify the problem. Even if it doesn't provide us with a formal reduction, it could give us some very useful insights while allowing us to work in a much simpler setting from a conceptual and technical perspective. Because we only need to worry about high level algorithms when considering recourse, trying to bound the recourse of an algorithm can be a much easier task than trying to bound the update time directly, which will most likely require a more detailed concrete implementation of the algorithm, along with the more detailed and intricate analysis that follows.

As of now, there are currently no known algorithms that have been proven to have a *worst case total recourse* or even an *expected total recourse* of $\tilde{\mathcal{O}}(m)$ on all instances, i.e. algorithms that have worst case or expected polylogarthmic recourse per update/insertion on all graphs, and it is an open problem. In the following chapters we shall be giving some original frameworks and results relating to the recourse of dynamic algorithms.

2.3 The Simple Algorithms

In this section we will introduce two very natural high level algorithms for incremental topological ordering. Before doing so, we will give a relevant definition to set the scene and make things a little easier.

Definition 2.3.1 (Affected Region)**.** *Suppose we are running some incremental topological ordering algorithm, and have a topological ordering of the current graph, $\prec$. If we insert some*

*edge $e = (u, v)$ into the graph, then the **affected region** is the section of the ordering between u and v, including u and v. If a node is in the affected region we say it's an **affected node**.*

Many of the simplest algorithms for incremental topological ordering only rearrange nodes that are contained in the affected region after an insertion. This leads to a very natural class of algorithms that was first identified by Haeupler et al. [HKM+12] which they referred to as *local* algorithms and are formally defined as follows.

Definition 2.3.2 (Local Algorithms)**.** *Suppose we have some incremental topological ordering algorithm, $\mathcal{A}$, which maintains a topological ordering of the graph, $\prec$, as edges are inserted. Suppose we insert an edge $e = (u, v)$ into the graph, then $\mathcal{A}$ is **local** if*

1. *$\mathcal{A}$ does not rearrange any nodes if $\prec$ is still a topological ordering for the updated graph*
2. *$\mathcal{A}$ only rearranges the nodes contained in the affected region of $\prec$. In other words, if $\prec \sim (x_1, ..x_i, y_1, ..., y_j, z_1, ..., z_k)$ with $y_1 = v$ and $y_j = u$, then the topological ordering, $\prec^*$, for the updated graph found by $\mathcal{A}$ satisfies $\prec^* \sim (x_1, ..x_i, y_{\phi_1}, ..., y_{\phi_j}, z_1, ..., z_k)$ where ϕ is a permutation of $\{1, ..., j\}$.*

While local algorithms may be simple and elegant, as we will see in later chapters, they also have inherent limitations which makes it possible to design instances which can not be solved efficiently by *any* local algorithm. The two high level algorithms that we are about to describe, which we shall refer to as the *simple* algorithms, are both local, so they are susceptible to such instances. However, in the next few chapters we shall be discussing variations of the recourse problem in different settings that may allow us to get past this issue, despite the limitations of local algorithms.

2.3.1 A Remark on High Level Algorithms

Because we only care about the recourse of these algorithms, we want descriptions of the algorithms that fully capture the effect they have on the ordering without worrying about their implementations, i.e. the data structures and mechanisms they use to achieve this effect. Because of this, we will be giving high level descriptions (adhering to the definition of a high level algorithm which we gave earlier, not the loose informal term) that will be intuitively clear and formal enough for our purposes but have no implicit or obvious associated concrete implementations.

2.3.2 The One-Way Search Algorithm, $\mathcal{A}_1$

The *simple one-way search algorithm*, $\mathcal{A}_1$, is a local algorithm which is a folklore result in the area. The algorithm can be split into 2 distinct phases: phase 1, a restricted forward-search, and phase 2, a sequence of node movements. Informally, after the insertion of an edge (u, v), in phase 1 the algorithm performs a forward search of the affected region, starting from u, and creates a subsequence of the ordering with all the nodes it finds. In phase 2, the algorithm removes all the nodes in this subsequence from $\prec$ and places the whole subsequence back into $\prec$ just to the right of u.

Algorithm 1: The Simple One-Way Search Algorithm, $\mathcal{A}_1$

Input: DAG $G = (V, E)$, a topological ordering $\prec$ of G, and an edge $e = (u, v) \notin E$ such that $G \cup \{e\}$ is a DAG

Output: a topological ordering $\prec^*$ of $G \cup \{e\}$

```
1  if u ≺ v then
2  |  return ≺;
3  ▷ suppose ≺ ∼ (u_1, ..., u_n);
4  ▷ suppose u = u_j and v = u_i for some 1 ≤ i < j ≤ n;
5  S ← ∅;
6  k ← i;
7  while k ≤ j do
8  |  if u_k ∈ reach_G(v) then
9  |  |  S.push(u_k);
10 |  k ← k + 1;
11 while S ≠ ∅ do
12 |  x ← S.pop();
13 |  move x just to the right of u in ≺;
14 return ≺;
```

2.3.3 The Greedy Two-Way Search Algorithm, $\mathcal{A}_2$

The *simple greedy two-way search algorithm*, $\mathcal{A}_2$, is also a local algorithm, for a more detailed discussion of this algorithm see [HKM+12]. This algorithm can be split into 2 distinct phases: phase 1, simultaneous restricted backwards and forwards searches, and phase 2, a sequence of node movements. Informally, after the insertion of an edge (u, v), in phase 1 the algorithm performs two simultaneous forwards and backwards searches of the affected region, starting from u and v respectively, and creates two subsequences of the ordering with the nodes it finds during each search. The searches continue until they 'meet each other in the middle'. In phase 2, the algorithm moves the nodes it found during the forward search to the right and the nodes it found during the backward search to the left in such a way that it restores the topological ordering.

Algorithm 2: The Simple Greedy Two-Way Search Algorithm, $\mathcal{A}_2$

Input: DAG $G = (V, E)$, a topological ordering $\prec$ of G, and an edge $e = (u, v) \notin E$ such that $G \cup \{e\}$ is a DAG

Output: a topological ordering $\prec^*$ of $G \cup \{e\}$

```
1  if u ≺ v then
2  |  return ≺;
3  ▷ suppose ≺ ∼ (u_1, ..., u_n);
4  ▷ suppose u = u_j and v = u_i for some 1 ≤ i < j ≤ n;
5  S ← ∅, R ← ∅;
6  s ← i, r ← j;
7  while S.peek() ≺ R.peek() do
8  |  if S.peek() ≺ R.peek() then
9  |  |  while u_s ∉ reach_G(v) do
10 |  |  |  s ← s + 1;
11 |  |  S.push(u_s);
12 |  |  s ← s + 1;
13 |  if S.peek() ≺ R.peek() then
14 |  |  while u_r ∉ reach_G^{-1}(u) do
15 |  |  |  r ← r − 1;
16 |  |  R.push(u_r);
17 |  |  r ← r − 1;
18 x ← S.pop();
19 while S ≠ ∅ do
20 |  y ← S.pop();
21 |  move y just to the right of x in ≺;
22 |  x ← y;
23 x ← R.pop();
24 while R ≠ ∅ do
25 |  y ← R.pop();
26 |  move y just to the left of x in ≺;
27 |  x ← y;
28 return ≺;
```

2.3.4 A comparison of $\mathcal{A}_1$ and $\mathcal{A}_2$

The algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ are very similar. The obvious difference is that $\mathcal{A}_1$ only performs one search while $\mathcal{A}_2$ performs two simultaneous searches. We say that $\mathcal{A}_2$ is a *greedy* algorithm because it's strategy is roughly to try and minimize the number of node movements in a single update. Figure 2.1 gives a visual representation of $\mathcal{A}_1$ and $\mathcal{A}_2$ both handling the same edge insertion.

2.4 Open Problems

There are many open problems related to incremental cycle detection and topological ordering, including some related to the notion of recourse which we have just introduced. Most of these are mainly concerned with obtaining polylogarithmic average update time per edge insertion in certain settings, the most important (and also the hardest) of which is the question of whether

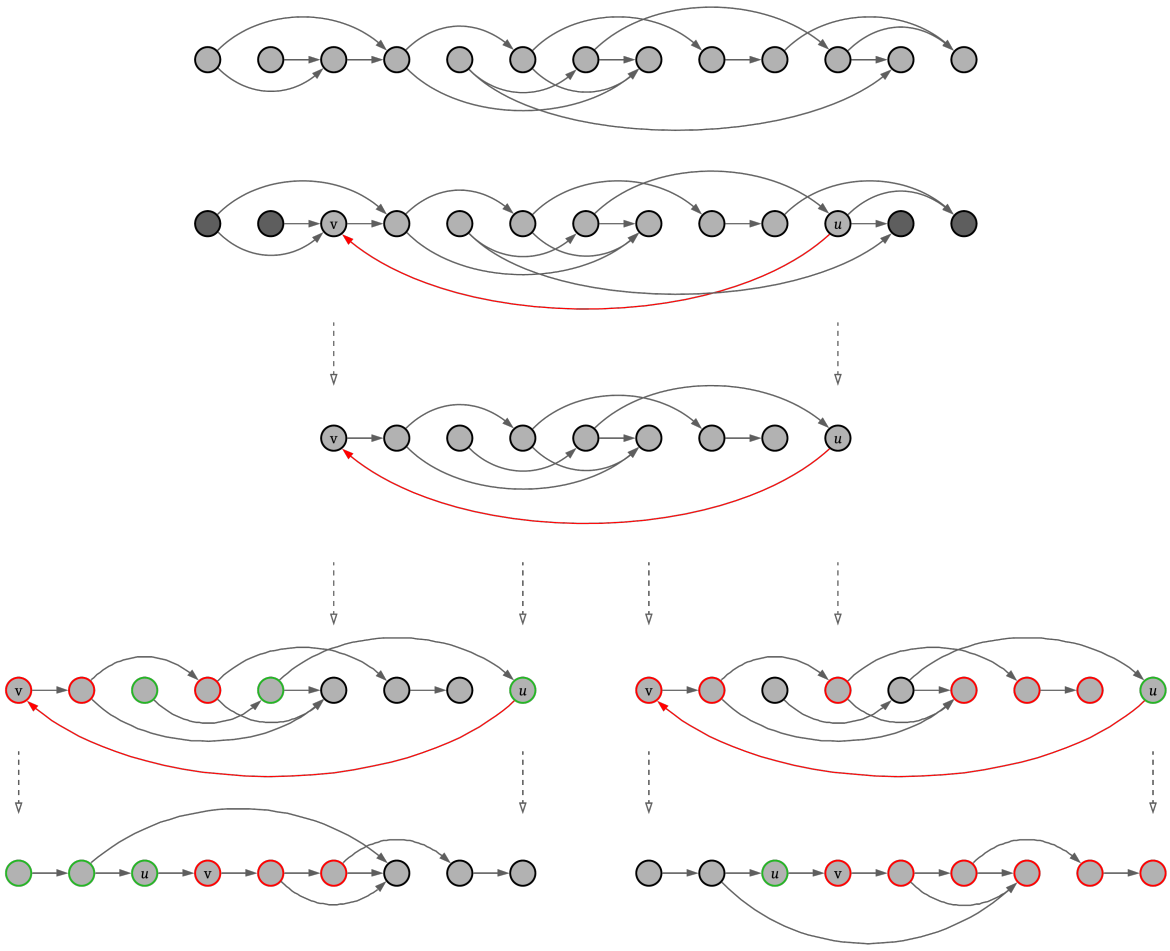

Figure 2.1: A comparison of the simple search algorithms, one-way search $\mathcal{A}_1$ (right), and greedy two-way search $\mathcal{A}_2$ (left), restoring the topological ordering after the insertion of (u, v) by rearranging nodes in the affected region.

there exists an algorithm for incremental cycle detection with a worst case or expected update time of $\tilde{\mathcal{O}}(m)$ on all inputs. While this problem is notoriously hard and far outside the scope of this project, there are other open problems that are more accessible. Before covering these, we will briefly discuss the *random-order model*, and it's relevance to incremental cycle detection and topological ordering.

2.4.1 Random Order Arrival vs. Adversarial Arrival

In the *random-order model* for dynamic algorithms, the input is chosen by an *adversary* but is randomly permuted before being presented to the algorithm. This process of randomizing the order of the input can often weaken the power of the adversary.[1]

This model is very applicable in the context of incremental topological ordering; it's very natural to ask whether randomizing the order of an insertion sequence before presenting it to some algorithm will provide us with improved algorithmic guarantees. We shall refer to this setting as *random-order arrival* and the setting where the adversary has complete control over the insertion sequence as *adversarial arrival*. My supervisor, Sayan Bhattacharya, has conjectured that there exists an incremental topological ordering algorithm that has low expected recourse per insertion for any graph under random-order arrival for any initial ordering. Where the expected recourse (given some inital ordering $\prec$) under random-order arrival is defined as follows.

$$\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[rec(\mathcal{E},\prec)] = \sum_{j\geq 0} j\cdot\mathbb{P}[rec(\mathcal{E},\prec)=j] = \frac{1}{m!}\sum_{\mathcal{E}\in\mathscr{S}_E} rec(\mathcal{E},\prec)$$

More formally, we have the following conjecture.

Conjecture 2.4.1. *Let $G=(V,E)$ be a DAG and $\prec\in\mathscr{S}_V$ any initial ordering of G, then there exists some incremental topological ordering algorithm such that it's recourse satisfies the following*

$$\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[rec(\mathcal{E},\prec)] = \tilde{\mathcal{O}}(m)$$

In fact, Bhattacharya has conjectured that the algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ have low expected recourse *per node* (not just per insertion) for any graph (and initial ordering) under random-order arrival. In contrast, note that insertion sequences leading to high recourse can easily be constructed for both of these algorithms under adversarial arrival (see Proposition 4.2.1). More formally, we have the following conjectures.

Conjecture 2.4.2. *Let $G=(V,E)$ be a DAG and $\prec\in\mathscr{S}_V$ any initial ordering of G, then the recourse of the simple one-way search algorithm, $\mathcal{A}_1$, satisfies the following*

$$\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[rec(\mathcal{E},\prec)] = \tilde{\mathcal{O}}(n)$$

Conjecture 2.4.3. *Let $G=(V,E)$ be a DAG and $\prec\in\mathscr{S}_V$ any initial ordering of G, then the recourse of the simple greedy two-way search algorithm, $\mathcal{A}_2$, satisfies the following*

$$\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[rec(\mathcal{E},\prec)] = \tilde{\mathcal{O}}(n)$$

Since for both of these algorithms we can assume that the input graphs are weakly connected (it can be shown that the effect of either algorithm on some node $u\in V(G)$ only depends on the structure of $G[reach_G(u)\cup reach_G^{-1}(u)]$, so we only have to consider graphs which are weakly

[1] Gupta el al. give a good survey of the random-order model and many examples in their paper [GS20]

connected), we can assume without loss of generality that $m = \Omega(n)$. Hence, Conjecture 2.4.2 and 2.4.3 both individually imply Conjecture 2.4.1.

There is a proposition given by Bernstein et al. in [BC18] which allows us to assume that under adversarial arrival the input graph has max degree at most $\Delta = \mathcal{O}(m/n)$. They do this by constructing a new graph with $\mathcal{O}(m)$ more nodes and max degree Δ that has a very similar structure to the original graph. However, this assumption is not valid under random order arrival, since random order arrival in this new graph will likely not correspond to random order arrival in the original. Hence, we will also be considering the following conjecture, which is a relaxed version of 2.4.1.

Conjecture 2.4.4. *Let $G = (V, E)$ be a DAG with max degree $\Delta = \mathcal{O}(m/n)$ and $\prec \in \mathscr{S}_V$ any initial ordering of G, then there exists some incremental topological ordering algorithm such that it's recourse satisfies the following*

$$\mathbb{E}_{\mathcal{E} \in \mathscr{S}_E}[rec(\mathcal{E}, \prec)] = \tilde{\mathcal{O}}(m)$$

2.4.2 Experimental Results

Before we started thinking about these problems, we implemented the simple algorithms, $\mathcal{A}_1$ and $\mathcal{A}_2$, and tested them on many randomly generated graphs.[2] All the data we collected supports both of these conjectures. Figure 2.2 shows some of the data. We have not included the details of how we generated the DAGs or the insertion sequences within this report since we want to focus on concrete results and not experimental data. However, we did try many different techniques and attained the same results each time.

[2] All the code for these implementations can be found at github.com/martin-costa/incremental-cycle-detection

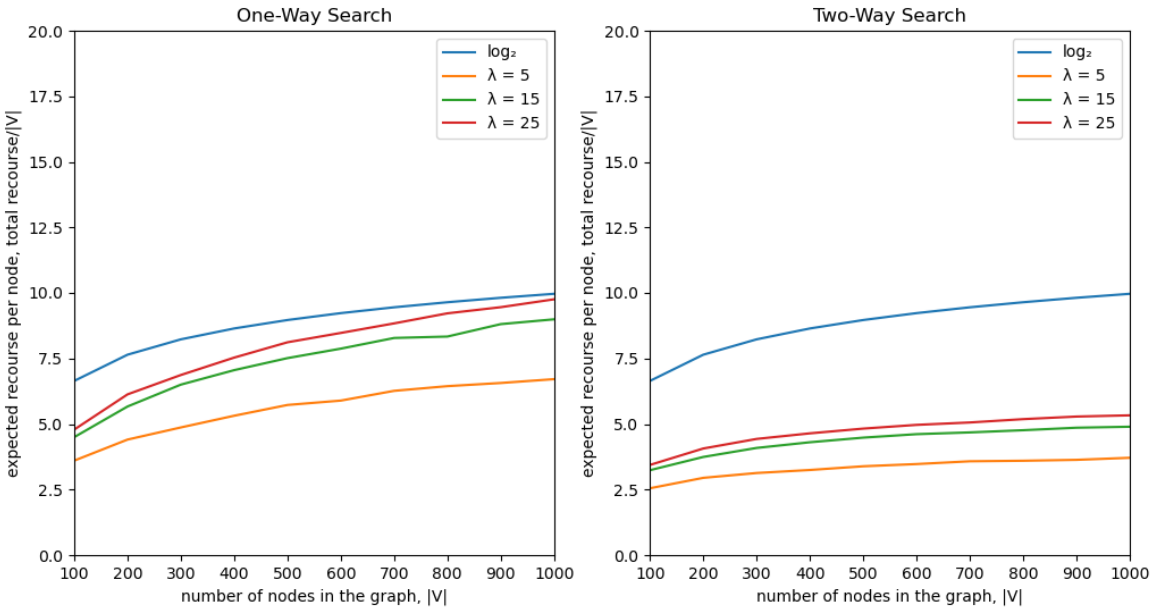

Figure 2.2: The curve $\lambda = k$ shows the average recourse per node (with respect to $\mathcal{A}_1$ on the left and $\mathcal{A}_2$ on the right) over many randomly generated insertion sequences of length exactly λn on randomly generated DAGs with n nodes.

Chapter 3

Divide and Conquer Framework

In this chapter we will be designing a divide and conquer framework for incremental cycle detection and topological ordering, which can be used to design divide and conquer algorithms to tackle these problems. We will use the ideas presented to construct a high level algorithm using the framework, and then see how it's recourse compares to $\mathcal{A}_1$ and $\mathcal{A}_2$ which we introduced in the last chapter.

The reason this is relevant is that currently there are no known *efficient implementations* of the simple search algorithms $\mathcal{A}_1$ or $\mathcal{A}_2$, i.e. implementations that give us a update time close to their recourse. Our hope is that this new framework could yield algorithms with low expected recourse which can be implemented more efficiently. Since this framework also produces non-local algorithms, the algorithms constructed using this framework could potentially be used to tackle more challenging open problems in the area related to adversarial arrival, since local algorithms are not suitable in this setting.

Our framework is constructed around our definition of *semi-topological partitions* which we introduce in the first section of this chapter. At the core of the framework is a dynamic data structure which we call the *meta-tree*; this data structure maintains a system of recursively embedded semi-topological partitions throughout the run of the algorithm and allows us to induce a topological ordering instead of having to maintain one directly. In essence, this framework gives a new way of maintaining a topological ordering in the incremental setting.

3.1 The Core Ideas

We will start off by giving the definitions and propositions that are at the core of this framework. The main idea here is the notion we have defined as *semi-topological partitions*, which will allow us to split the ordering into 3 sections in quite a natural way, allowing us to break down the problem.

3.1.1 Basic Definitions

In order to create a divide and conquer framework for the problem, we need to be able to recursively break down the problem into smaller sub-problems, leading to the following natural definition.

Definition 3.1.1 (Topological Partition)**.** *Let $G = (V, E)$ be a DAG, then (L, R) is a topological partition of G if*

1. *L and R partition V, i.e. $L \cap R = \varnothing$ and $L \cup R = V$*

2. *adding any edge from R to L will create a cycle*

3. *there are no edges from R to L*

It turns out that this notion of topological partitions is too strict to be used directly; it is very easy to construct a DAG $G = (V, E)$ with $m = \Omega(n^2)$ that has no non-trivial topological partitions. For example, suppose $u \in V$ and let (L, R) be a partition of $V \setminus \{u\}$ with $\Theta(|L|) = \Theta(|R|) = \Theta(n)$ and $E = L \times R$. We can see that there can't exist a topological partition of G because u is isolated in G, so adding any edge into the graph which is incident on u would never create a cycle, even though $m = \Omega(n^2)$.

However, even though a topological partition doesn't exist for G, we can see (at least intuitively) that G is *almost* partitioned topologically by (L, R), so we now give a relaxation of this definition that captures this formally.

Definition 3.1.2 (Semi-Topological Partition)**.** *Let $G = (V, E)$ be a DAG, then (L, F, R) is a semi-topological partition of G with freedom $\frac{1}{n}|F|$ if*

1. *L, F and R partition V*

2. *adding any edge from R to L will create a cycle*

3. *there are no edges from F to L, from R to F or from R to L, i.e. $\forall u \in F$, $u \notin reach_G^{-1}(L)$ and $u \notin reach_G(R)$, and $\forall u \in R$, $u \notin reach_G^{-1}(L)$*

We can see that (L, R) is a topological partition (TP) of G if, and only if, $(L, \varnothing, R)$ is a semi-topological partition (STP) of G. More importantly, we can see that this new definition of STPs does not have the same limitations as the definition of TPs. In fact, any non-empty DAG has a non-trivial STP (see Lemma 3.1.5).

3.1.2 Properties of STPs

We will now make some observations about the properties of STPs. In the next section we will discuss how we can use these properties to construct a divide and conquer algorithm using the idea of STPs as a foundation.

Proposition 3.1.3. *Let $G = (V, E)$ be a DAG, given disjoint $L, R \subseteq V$, the following are equivalent*

1. *adding any edge from R to L will create a cycle*

2. *$\forall u \in L$, $v \in R$ we have $v \in reach_G(u)$*

3. *$L \subseteq \bigcap_{u \in R} reach_G^{-1}(u)$*

4. *$R \subseteq \bigcap_{u \in L} reach_G(u)$*

Proof. **1.** $\Leftrightarrow$ **2.** If adding any edge from R to L creates a cycle, then from any node in L we can reach any node in R. If from any node in L we can reach any node in R, then adding any edge (v, u) from R to L will create a cycle, since we can append this edge to the end of a path from u to v.

2. $\Leftrightarrow$ **3.** and **2.** $\Leftrightarrow$ **4.** are obvious. □

Proposition 3.1.4. *Let $G = (V, E)$ be a DAG. Suppose (L, F, R) is an STP of G, then*

1. $\forall u \in L$, $reach_G^{-1}(u) \subseteq L$

2. $\forall u \in R$, $reach_G(u) \subseteq R$

Proof. **1.** Suppose $u \in L$ and $v \in reach_G^{-1}(u)$, if $v \notin L$ then there is a path from $v \in F \cup R$ to u, hence there exists an edge from F to L or from R to L giving a contradiction. It follows that $reach_G^{-1}(u) \subseteq L$. **2.** Same argument as **1.** □

This next lemma will be useful for finding STPs, it will give us a very simple way to construct an STP by looking at any edge in the graph.

Lemma 3.1.5 (Simple STP Construction)**.** *Let $G = (V, E)$ be a DAG, let $u \in V$ and $v \in reach_G(u)$, then*

1. *if $u \neq v$ then $(reach_G^{-1}(u), F, reach_G(v))$ is an STP of G*

2. *if $u = v$ then $(reach_G^{-1}(u), F, reach_G(u) \setminus \{u\})$ and $(reach_G^{-1}(u) \setminus \{u\}, F, reach_G(u))$ are STPs of G*

with $F = V \setminus (reach_G^{-1}(u) \cup reach_G(v))$

Proof. **1.** Let $L = reach_G^{-1}(u)$ and $R = reach_G(v)$. If $L \cap R \neq \varnothing$ then $u \in reach_G(v)$ so G contains a cycle, contradicting the fact that G is a DAG. So L, F and R partition V. If we add any edge (v^*, u^*) from R to L, since we can form paths from $u^* \rightsquigarrow u$, $u \rightsquigarrow v$ and $v \rightsquigarrow v^*$, we can append these paths together along with (v^*, u^*) to obtain a cycle. Let $w \in F$ and suppose $w \in reach_G^{-1}(L) = reach_G^{-1}(u)$, then by the definition of L, $w \in L$, giving a contradiction, so there are no edges from F to L. Similarly, there are no edges from R to F or from R to L since R and L are disjoint. It follows that $(reach_G^{-1}(u), F, reach_G(v))$ is an STP of G. **2.** Same argument as **1.** □

We refer to an STP that has one of the forms presented in Lemma 3.1.5 as a *simple* STP. Given any DAG with at least one edge, we can now construct an STP of the graph by taking any edge e and applying the lemma above to it's endpoints, we call this the *simple STP induced by e*. This lemma will be crucial in the construction of the algorithm which we will see later.

This next lemma formalises the main reason that the notion of STPs allows us to break down the problem. It follows directly from the definition of an STP, so it's quite trivial, but it's an important observation.

Lemma 3.1.6. *Let $G = (V, E)$ be a DAG. Suppose (L, F, R) is an STP of G and that $\prec_L$, $\prec_F$ and $\prec_R$ are topological orderings of $G[L]$, $G[F]$ and $G[R]$ respectively. The ordering $\prec_G$ that we get from combining these topological orderings and setting $L \prec_G F \prec_G R$ is a topological ordering of G. Formally, given $u, v \in G$, if for some $S \in \{L, F, R\}$ we have $u, v \in S$ then $u \prec_G v$ if $u \prec_S v$, else $u \prec_G v$ if $u \in L$ or $v \in R$.*[1]

Proof. Follows directly from the definition of a topological ordering combined with the definition of an STP. □

[1] Given some $S \subseteq V$, the graph $G[S]$ is the *induced subgraph* whose vertex set is S and whose edge set is all edges in E that have both endpoints in S

3.2 Constructing the Framework

In this section we will present a data structure that can be updated incrementally alongside the graph as edge insertions are made. This data structure will be constructed by recursively computing embedded STPs, and will allow us to maintain a topological ordering of the graph by maintaining this system of embedded STPs (and taking advantage of their properties) instead of maintaining a topological ordering directly. We will then show how this structure can be used to design an algorithm for incremental cycle detection and topological ordering.

To make things easier to read, for the rest of this chapter fix a DAG $G = (V, E)$ and an insertion sequence $\mathcal{E} \in \mathscr{S}_E$ of G. As usual, we start with the graph $G_0 = (V, \varnothing)$ and add edges from $\mathcal{E}$ one at a time. Let G_t denote the graph after t edge insertions from $\mathcal{E}$, i.e. the graph $(V, \{\mathcal{E}_1, ..., \mathcal{E}_t\})$, and abbreviate $reach_{G_t}$ and $reach^{-1}_{G_t}$ to $reach_t$ and $reach^{-1}_t$ respectively.

3.2.1 The Meta Tree

As we incrementally perform edge insertions, we also maintain a rooted meta-tree $\mathcal{T}$ which captures information about STPs that we will compute along the way, helping us to classify the edge insertions that are being performed into various types and to (indirectly) maintain a topological ordering of the graph.

Let $\mathcal{T}_t$ denote the state of the meta-tree after the first t insertions. All of the internal meta-nodes x of $\mathcal{T}_t$ have exactly 3 children, the left, middle and right, denoted by x_L, x_F and x_R respectively. The meta-leaves of $\mathcal{T}_t$ all have ordered subsets of $V(G)$ *attached* to them, given a meta-leaf x of $\mathcal{T}_t$, we denote this set by $set_t(x)$ and the ordering of this set by $\prec^x_t$. We also recursively define set_t for internal meta-nodes x by setting $set_t(x) = set_t(x_L) \cup set_t(x_F) \cup set_t(x_R)$ with the ordering of $set_t(x)$ induced from the orderings of the set_t of its children and setting $set_t(x_L) \prec^x_t set_t(x_F) \prec^x_t set_t(x_R)$.

We want to construct and update the meta-tree, $\mathcal{T}$, in such a way that it always satisfies certain properties. These invariants are as follows.

Invariant 3.2.1 (The Meta Tree Properties)**.** *The meta-tree $\mathcal{T}$ maintains the following properties as edges are inserted, $\forall\, t \in \{0, ..., m\}$:*

1. *The subsets of V attached to the meta-leaves of $\mathcal{T}_t$ form a partition of V*
2. *Given any meta-node x of $\mathcal{T}_t$, the ordering $\prec^x_t$ is a topological ordering on $G_t[set_t(x)]$*
3. *Given any internal meta-node x of $\mathcal{T}_t$, $(set_t(x_L), set_t(x_F), set_t(x_R))$ is an STP of $G_t[set_t(x)]$*

The meta-tree $\mathcal{T}_0$ is a single meta-node r with $set_t(r) = V(G)$ and any random ordering of the nodes $\prec^r_t$. We want to find an algorithm that allows us to efficiently compute $\mathcal{T}_t$ from $\mathcal{T}_{t-1}$ and G_t such that $\mathcal{T}_t$ satisfies all the meta properties, giving us $\prec^r_t$ which by properties 2 and 3 of the meta-tree defines a topological ordering on G_t.

Remark 3.2.2. *The second meta tree property actually follows from a relaxation of the same property that only holds for meta-leaves by applying Lemma 3.1.6, so to show that property 2 holds it is sufficient to show it holds for all meta-leaves.*

3.2.2 Handling Edge Insertions

We will now describe, at a high level, how we can update the meta-tree after an insertion has occurred such that its properties are all preserved. We will do this by classifying the edge insertions into different types and handling each type of insertion differently.

Let $\prec_t^{meta}$ be an ordering of the meta-leaves of the meta-tree created by traversing $\mathcal{T}_t$ in pre-order, so for any two meta-leaves $x, y \in V(\mathcal{T}_t)$, $x \prec_t^{meta} y$ if x is explored before y in the pre-order traversal of $\mathcal{T}_t$. Notice that if we combine $\prec_t^{meta}$ with the orderings $\prec_t^x$ for the sets $set_t(x)$ at each meta-leaf x, we get $\prec_t^r$. For some node $u \in V(G)$, let $node_t(u)$ denote the unique meta-leaf x in $\mathcal{T}_t$ that has $u \in set_t(x)$.

Let $\mathcal{E}_t = (u_t, v_t)$ be the edge inserted at step t into G_{t-1} to obtain G_t. Let $p_t = (x_1, ..., x_k)$ be the unique path from $x_1 = node_{t-1}(u_t)$ to $x_k = node_{t-1}(v_t)$ in $\mathcal{T}_{t-1}$. Let $x_j \in p_t$ denote the least common ancestor of x_1 and x_k in $\mathcal{T}_{t-1}$. We now consider the 3 following cases that may occur when an edge is inserted.

1. $node_{t-1}(u_t) \prec_{t-1}^{meta} node_{t-1}(v_t)$
2. $node_{t-1}(u_t) = node_{t-1}(v_t)$
3. $node_{t-1}(v_t) \prec_{t-1}^{meta} node_{t-1}(u_t)$

Here is an outline of what can be done to maintain the properties of the meta-tree in these 3 cases.

Case 1 $node_{t-1}(u_t) \prec_{t-1}^{meta} node_{t-1}(v_t)$**:**

If this is the case, then we don't need to do anything since $x_1 \prec_{t-1}^{meta} x_k$ so $u_t \prec_{t-1}^r v_t$. So we can just let $\mathcal{T}_t = \mathcal{T}_{t-1}$.

Case 2 $node_{t-1}(u_t) = node_{t-1}(v_t)$**:**

If this is the case, then we can apply Lemma 3.1.5 to (u_t, v_t) and break down the set $S = set_{t-1}(x)$ with $x = node_{t-1}(u_t)$ into an STP (L, F, R), with $L = reach_t^{-1}(u_t) \cap S$, $R = reach_t(v_t) \cap S$ and $F = S \setminus (L \cup R)$. To compute $\mathcal{T}_t$, take $\mathcal{T}_{t-1}$ and add 3 children, x_L, x_F and x_R to x with $set_t(x_L) = L$, $set_t(x_F) = F$ and $set_t(x_R) = R$, with the orderings of these sets induced by restricting $\prec_{t-1}^x$ to L, F and R respectively.

Case 3 $node_{t-1}(v_t) \prec_{t-1}^{meta} node_{t-1}(u_t)$**:**

This case is by far the most complex, since we need to be able to remove and add nodes to STPs in such a way that we preserve the invariant properties of the meta-tree. Immediately we can observe that since $node_{t-1}(u_t) \not\prec_{t-1}^{meta} node_{t-1}(v_t)$ we have $u_t \not\prec_{t-1}^r v_t$, hence $\mathcal{T}_t \neq \mathcal{T}_{t-1}$.

In the next section we will show how to recursively add sets of topologically ordered nodes into the sets attached to sub-trees of the meta-tree while maintaining the properties of the meta-tree. For now just assume it's possible. Consider the state of $\mathcal{T}_{t-1}$ around the meta-node x_j defined above. Since we have $node_{t-1}(v_t) \prec_{t-1}^{meta} node_{t-1}(u_t)$, there are 3 possibilities for where x_{j-1} and x_{j+1} can be located. By definition of x_j we have $x_{j-1}, x_{j+1} \in \{x_{jL}, x_{jF}, x_{jR}\}$. Now consider the 3 cases.

3.1 $x_{j-1} = x_{jR}$ and $x_{j+1} = x_{jL}$: we can immediately deduce that a cycle is present in G_t since an edge has been added from the *right* nodes to the *left* nodes of the STP associated with the meta-node x_j

3.2 $x_{j-1} = x_{jR}$ and $x_{j+1} = x_{jF}$: remove $reach_t(v_t) \cap set_{t-1}(x_{jF})$ from $set_{t-1}(x_{jF})$ and add it to $set_{t-1}(x_{jR})$.

3.3 $x_{j-1} = x_{jF}$ and $x_{j+1} = x_{jL}$: remove $reach_t^{-1}(u_t) \cap set_{t-1}(x_{jF})$ from $set_{t-1}(x_{jF})$ and add it to $set_{t-1}(x_{jL})$.

So we can fully restore the properties of the meta-tree (and hence the topological ordering of the graph that is induced by the meta-tree) with a single call to this (currently undefined) algorithm that moves nodes between the STPs in the meta-tree.

Remark 3.2.3. *At the start of this section we assumed that G was a DAG, this means that case 1 given above is actually impossible, since this can only occur if a cycle is present in G. This assumption is also the reason that we do not have to check for cycles in cases 2, 3.2 and 3.3; it is not hard to extend the algorithm so that it makes this check, allowing us to relax the assumption that G is a DAG, but this formulation is simpler and does a good job of presenting the framework.*

3.3 Computing the Meta Trees

We will now give a high level algorithm that will allow us to recursively move topologically ordered sets of nodes between the sets attached to meta-nodes in the meta-tree where necessary, while also maintaining the properties of the meta-tree. Combining this with the algorithm given in the last section that required the ability to perform this operation, we will have constructed a high level divide and conquer algorithm for incremental cycle detection and topological ordering.

Similar to the simple algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$, we will not be giving any concrete implementation of this algorithm with a well defined update time. The description that we have given of this algorithm is all that is required to study it's combinatorial properties, such as the recourse of this algorithm in different settings, which is our main interest.

3.3.1 Modifying the STPs of the Meta Tree

As we saw in the last section, in some cases, we need to be able to move nodes between the sets attached to the meta-leaves of the meta-tree. Here we give a high level algorithm that will allow us to do this. In particular, we want an algorithm that given a meta-node x in $\mathcal{T}_{t-1}$, allows us to move topologically ordered subsets $A \subseteq \bigcup_{y \prec^{meta}_{t-1} x} set_{t-1}(y)$ and $B \subseteq \bigcup_{x \prec^{meta}_{t-1} y} set_{t-1}(y)$ with orderings $\prec_A$ and $\prec_B$ respectively, into $S = set_{t-1}(x)$, with the sets satisfying the following conditions[2]

1. A, B and S are pairwise disjoint
2. the ordering $\prec^*$ we get from combining $\prec_A$, $\prec_B$ and $\prec^x_{t-1}$ and setting $A \prec^* S \prec^* B$ disagrees with at most one edge in G_t between the sets

We now give an outline for the algorithm ADD$(\mathcal{T}, G_t, x, A, B)$ presented in Algorithm 3 which takes the current meta-tree $\mathcal{T}$ (which may not satisfy the meta-tree properties after the insertion of $\mathcal{E}_t$), the graph G_t, a meta-node x in $\mathcal{T}$, two sets of nodes A and B satisfying the properties above, and moves the nodes in A and B into the meta-leaves in the sub-tree of x. A single call of this algorithm restores all the properties of the meta-tree after a type 3 insertion. More specifically, it can compute $\mathcal{T}_t$ from $\mathcal{T}_{t-1}$ and G_t after a type 3 edge insertion.

Note that up to now we have been considering the meta-tree at specific states, i.e. we have only cared about $\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_m$, and considered how we have to modify the meta-tree $\mathcal{T}$ starting from $\mathcal{T}_{t-1}$ to get to a state where it satisfies the meta-tree properties for the updated graph G_t. We will now have to consider the state of the meta-tree as we modify it to transition between

[2]Here we extend the ordering $\prec^{meta}_t$ to pairs of unrelated meta-nodes in $\mathcal{T}_t$ in the obvious way, where $x \prec^{meta}_t y$ if x is explored before y in the pre-order traversal of $\mathcal{T}_t$

these *nice* states, so the notation will be slightly different. We just generalise the notation from these specific states of the meta-tree to the meta-tree at any state in the obvious way by replacing $\prec_t^x$ with $\prec_{\mathcal{T}}^x$, $set_t(x)$ with $set_{\mathcal{T}}(x)$, and so on.

Algorithm 3: ADD$(\mathcal{T}, G_t, x, A, B)$

▷ **STEP 1: Base case**
If x is not an internal meta-node of $\mathcal{T}$ (i.e. x is a meta-leaf), then let $set_{\mathcal{T}}(x) = A \cup set_{\mathcal{T}}(x) \cup B$, and let $\prec_{\mathcal{T}}^x$ be the ordering we get from combining the other orderings as done above. If there is an edge that disagrees with this ordering, i.e. this is not a topological ordering on $set_{\mathcal{T}}(x)$, then treat this like a type 2 edge insertion.
▷ **STEP 2: Recursive call**
First we create 6 ordered sets that we will use to partition A and B. L_A, F_A and R_A for A and L_B, F_B and R_B for B initially all $\varnothing$. Let L, F and R denote $set_{\mathcal{T}}(x_L)$, $set_{\mathcal{T}}(x_F)$ and $set_{\mathcal{T}}(x_R)$ respectively.
▷ **STEP 3**
Find $reach_t^{-1}(L) \cap A$, remove it from A, and place it in L_A;
Find $reach_t^{-1}(L) \cap B$, remove it from B, and place it in L_B;
Find $reach_t(R) \cap A$, remove it from A, and place it in R_A;
Find $reach_t(R) \cap B$, remove it from B, and place it in R_B;
▷ **STEP 4**
Find $reach_t^{-1}(L_A \cup L_B) \cap F$, remove it from F, and place it in L_B;
Find $reach_t(R_A \cup R_B) \cap F$, remove it from F, and place it in R_A;
▷ **STEP 5**
Let $F_A = A \setminus (L_A \cup R_A)$ and $F_B = B \setminus (L_B \cup R_B)$;
▷ **STEP 6**
recursive call to ADD$(\mathcal{T}, G_t, x_L, L_A, L_B)$;
recursive call to ADD$(\mathcal{T}, G_t, x_F, F_A, F_B)$;
recursive call to ADD$(\mathcal{T}, G_t, x_R, R_A, R_B)$;

3.3.2 Correctness

We will now be giving a brief informal justification of the correctness of the algorithm we have constructed, which we shall refer to as $\mathcal{A}_D$. We have decided not to include the full formal argument since even though the proofs required are all very similar and fairly trivial, they are quite long and not very insightful. The overview of the correctness argument is as follows.

We must show that the properties given in Invariant 3.2.1 hold after each insertion is handled by the methods we have described. This can easily be done by induction. The base case is trivial since $\mathcal{T}_0$ is a single node and any ordering of G_0 is a topological ordering since G_0 is empty. Now assume that $\mathcal{T}_{t-1}$ satisfies Invariant 3.2.1, we want so show that $\mathcal{T}_t$ also satisfies these properties. This is best done case by case for each type of edge insertion. For a type 1 edge insertion, this follows immediately from that fact that $\mathcal{T}_t = \mathcal{T}_{t-1}$ and that the edge $\mathcal{E}_t$ agrees with the topological ordering $\prec_{t-1}^r$ by its definition. For a type 2 edge insertion, the argument is still very simple, properties 1 and 2 hold by the construction of the new meta-leaves combined with Lemma 3.1.6 and property 3 holds by Lemma 3.1.5. For a type 3 edge insertion the argument is more involved, since we must check that Algorithm 3 is correct. We will not give the proof for this, but we will give the following propositions, which are at the core of the proof of correctness for Algorithm 3 and are the motivation behind the way we chose to handle

type 3 edge insertions.

Proposition 3.3.1. *Let $G = (V, E)$ be a DAG. Suppose (L, F, R) is an STP of G with $u \in V$, $D = reach_G(u)$, $A = reach_G^{-1}(u)$ then $(L \setminus D, F \setminus D, R \setminus D)$ is an STP of $G[V \setminus D]$ and $(L \setminus A, F \setminus A, R \setminus A)$ is an STP of $G[V \setminus A]$.*

Proof. Let $P = (L \setminus D, F \setminus D, R \setminus D)$. We need to check the 3 conditions of an STP. **1.** Since L, F and R partition V, we get that $L \setminus D$, $F \setminus D$, and $R \setminus D$ partition $V \setminus D$. **3.** There can't be any edges from $F \setminus D$ to $L \setminus D$, from $R \setminus D$ to $F \setminus D$, or from $R \setminus D$ to $L \setminus D$, or else this would contradict the fact that (L, F, R) is an STP. **2.** Denote $G[V \setminus D]$ by H. Using the properties of an STP, by Proposition 3.1.3, we just need to show that $\forall v \in L \setminus D$, $w \in R \setminus D$ we have $w \in reach_H(v)$. Suppose this is not the case, then some $w \in R \setminus D$ can't be reached from some $v \in L \setminus D$ in H. Since $w \in reach_G(v)$, there exists some path p from v to w in G. Since $p \nsubseteq V \setminus D$ we get that $p \cap D \neq \varnothing$, so let $u^* \in p \cap D$. Since $reach_G(u^*) \subseteq D$, $p \cap D$ is a path from u^* to w contained in D, hence $w \in D$, contradicting the fact that $w \in R \setminus D$. Hence, adding any edge from $R \setminus D$ to $L \setminus D$ creates a cycle. It follows that P is an STP of H. An analogous argument gives the same for the STP $(L \setminus A, F \setminus A, R \setminus A)$ of $G[V \setminus A]$. □

Note that we can modify Proposition 3.3.1 and extend the result to $D = reach_G(X)$ and $A = reach_G^{-1}(X)$ for $X \subseteq V$ without too much work. We can obtain similar results for reducing the freedom of STPs instead of removing nodes from them.

Proposition 3.3.2. *Let $G = (V, E)$ be a DAG and $e = (u, v) \notin E$ such that $G \cup \{e\}$ is a DAG. Let (L, F, R) be an STP of G, then*

1. *if $u \in R$ and $v \in F$ then $(L, F \setminus reach_G(v), R \cup reach_G(v))$ is an STP of $G \cup \{e\}$*
2. *if $u \in F$ and $v \in L$ then $(L \cup reach_G^{-1}(u), F \setminus reach_G^{-1}(u), R)$ is an STP of $G \cup \{e\}$*

Proof. Similar ideas to Proposition 3.3.1. □

3.4 Results

We will now discuss how we can quantify the recourse of this algorithm by implementing $\mathcal{A}_D$ as an algorithm which maintains the meta-tree by only performing mode movements. We will then discuss the implications of this framework as well as experimental results.

3.4.1 The Recourse of $\mathcal{A}_D$

In order to define the recourse created by the insertion of the edge $\mathcal{E}_t$, we must modify the algorithm such that it maintains the ordering induced by the meta-tree by performing node movements. We can get a good lower bound on the recourse of an algorithm which does this quite easily. A type 1 edge insertion will not create any recourse since the meat-tree (and hence the ordering it induces) remains unchanged. A type 2 edge insertion will create recourse at most $|L| + |R|$ recourse since it can rearrange the ordering by moving the nodes in L and to the left and the nodes in R to the right one at a time with a sequence of node movements. The recourse created by a type 3 insertion can be upper bounded by summing up the sizes of the sets A and B in each recursive call of the algorithm ADD$(\mathcal{T}, G_t, x, A, B)$, since we can imagine the algorithm moving all of these nodes individually with node movements each time the method is called. This upper bound on the recourse is quite natural (especially if we want it to be close to the update time of the algorithm, since we count all the recursive calls) and is sufficient to obtain good experimental results.

3.4.2 Implications of the Framework

Since this framework can be used to construct non-local algorithms that perform well experimentally on random graphs, it is possible that it could be used to construct non-local algorithms that perform well under adversarial arrival. $\mathcal{A}_D$ is a very simple example of an algorithm that can be created using this framework, and it does not always perform well under adversarial arrival (it is easy to construct an insertion sequence of length $\mathcal{O}(n)$ that leads to $\Omega(n^2)$ recourse by creating $\Omega(n)$ embedded STPs with very high freedom, and then recursively moving all the nodes in the deepest STP into the next deepest with one insertion).

As stated earlier, there are currently no known algorithms which are believed to have near-linear recourse under random-order arrival and also admit efficient implementations. Since the class of algorithms produced by this framework is very different to other algorithms for incremental topological ordering, it is possible that an algorithm produced by this framework may meet this criteria. Hence, it may be possible to use divide and conquer algorithms for incremental topological ordering to obtain near-linear update times under random order arrival.

3.4.3 Experimental Results

While we were unable to prove that this algorithm has an average expected recourse of $\mathcal{O}(\text{polylog}\, n)$ per insertion, we did implement the algorithm $\mathcal{A}_D$ and test it in the same way as the simple algorithms in section 2.4.2.[3] Figure 3.1 shows some of the data we collected. All the data collected supports that $\mathcal{A}_D$ may give low expected recourse under random order arrival for any DAG. This implies that $\mathcal{A}_D$ is a candidate algorithm that could be used to prove Conjecture 2.4.1.

[3] All the code for these implementations can be found at github.com/martin-costa/incremental-cycle-detection

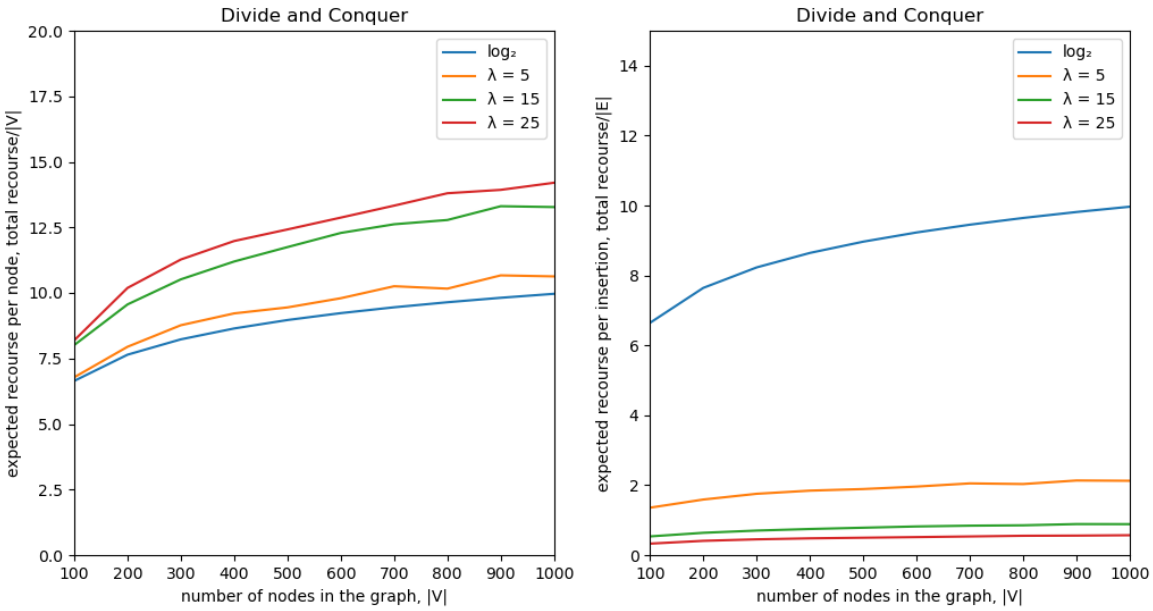

Figure 3.1: The curve $\lambda = k$ shows the average recourse of $\mathcal{A}_D$ per node (left) and per insertion (right) over many randomly generated insertion sequences of length exactly λn on randomly generated DAGs with n nodes.

Chapter 4

Breaking Lower Bounds with Random-Order Arrival

In this chapter we will be expanding on the claims we made at the end of chapter 4. In particular, we shall be covering the lower bound on recourse for *local algorithms* that we briefly mentioned earlier as well as focusing on random order arrival. We will start off by giving an upper bound on the worst case recourse for $\mathcal{A}_2$, and then deducing this bound is tight since $\mathcal{A}_2$ is a local algorithm. We will then show that under random-order arrival, the lower bound on recourse for local algorithms given in [HKM+12] does not hold for $\mathcal{A}_2$. To make things simpler, we will only be focusing on the most important case where the input graphs are sparse, since the problem is well understood for dense graphs.

By showing that the proof of this lower bounds breaks down under random order arrival, it follows that it may be possible to use local algorithms to solve Conjecture 2.4.1, even if local algorithms are not strong enough to solve the main open problem in the area.

4.1 Upper Bound on the Recourse of $\mathcal{A}_2$

We will now give an upper bound on the recourse of the simple greedy 2-way search algorithm, $\mathcal{A}_2$. While proofs of this bound already exist using similar arguments that rely on the same ideas, my proof uses the following potential function. For any graph H, define $\Psi(H) \triangleq \sum_{u \in V(H)} |reach_H(u)| \leq n^2$.

Proposition 4.1.1. *Given any DAG $G = (V, E)$, the recourse of the simple greedy 2-way search algorithm, $\mathcal{A}_2$, satisfies the following.*

$$\max_{\mathcal{E} \in \mathscr{S}_E, \prec \in \mathscr{S}_V} rec(\mathcal{E}, \prec) = \mathcal{O}(n\sqrt{m})$$

Proof. Fix some DAG $G = (V, E)$ and any $\mathcal{E} \in \mathscr{S}_E$, $\prec \in \mathscr{S}_V$. Suppose that during the run of $\mathcal{A}_2$ on input $\mathcal{E}$ staring with initial ordering $\prec$, the edge $\mathcal{E}_t = (u, v)$ is inserted causing the total recourse to increase by $\lambda = rec(\mathcal{E}, \prec)_t$. Let D denote the descendants of v discovered by the forward search done by the algorithm during the insertion, and A denote the ancestors of u discovered by the backwards search done by the algorithm during the insertion. Let $a = |A|$ and $d = |D|$. Before the insertion, at most one node in D could reach at most one node in A, but after the insertion, all of the nodes in D can reach all of the nodes in A. So the descendants

of at least $a-1$ nodes increase by at least $d-1$. Combined with the fact that $|a-d| \leq 1$ and $\lambda \leq a+d$ we get that

$$\left(\frac{\lambda-1}{2}-1\right)^2 \leq (a-1)(d-1) \leq \Psi(G_t^{\mathcal{E}}) - \Psi(G_{t-1}^{\mathcal{E}})$$

Summing over $t \in [m]$ on both sides of the inequality gives

$$\sum_{t=1}^{m}\left(\frac{rec(\mathcal{E},\prec)_t - 1}{2}-1\right)^2 \leq \Psi(G_m^{\mathcal{E}}) - \Psi(G_0^{\mathcal{E}})$$

Using a simple norm bound and noticing that $\Psi(G_0^{\mathcal{E}}) = 0$ we get

$$\sum_{t=1}^{m}\left(\frac{rec(\mathcal{E},\prec)_t - 1}{2}-1\right) \leq \left(m\sum_{t=1}^{m}\left(\frac{rec(\mathcal{E},\prec)_t - 1}{2}-1\right)^2\right)^{\frac{1}{2}} \leq \sqrt{m\Psi(G)} \leq n\sqrt{m}$$

Moving some terms to the right we get $rec(\mathcal{E},\prec) \leq 3m + 2n\sqrt{m} = \mathcal{O}(n\sqrt{m})$. □

4.2 Lower Bound on the Recourse of Local Algorithms

In the work of [HKM+12], they construct a collection of both sparse and dense DAGs $\{H_{k,p} : p,k \in \mathbb{N}, p \leq k\}$ with $n = \Theta(pk)$, $m = \Theta(k(k+p))$, with corresponding insertion sequences and initial orderings such that the recourse of *any* local algorithm (recall definition 2.3.2) is $\Omega(n\sqrt{m})$ on these inputs. We will now define these instances and show that they lead to high recourse. To make things simpler, since the problem is already well understood for dense graphs, we will only be considering the more important cases where the graphs are sparse, so we will only care about $\{H_{k,k}\}$ where $p = k$ (note that the sparse graphs $\{H_{1,p}\}$ where $k = 1$ are just paths of length $\Theta(p)$ and not very interesting).[1]

4.2.1 High Recourse Instances

Fix some $k \in \mathbb{N}$, abbreviate $H_{k,k}$ to H_k and let $H_k = (V,E)$. We can construct the graph H_k with corresponding insertion sequence, $\mathcal{E} \in \mathscr{S}_E$, and initial ordering, $\prec \in \mathscr{S}_V$, such that $rec(\mathcal{E},\prec) = \Omega(n\sqrt{n})$ as follows.

Constructing the Graph

For all $i \in [k]$ let $S_i = \{x_i, z_i^2, ..., z_i^{k-1}, y_i\}$ be sets of nodes of size k and set $V = \bigcup_{i=1}^{k} S_i$. Starting with $E = \varnothing$, for all $i \in [k]$ add the edges $(x_i, z_i^2), (z_i^2, z_i^3), ..., (z_i^{k-2}, z_i^{k-1}), (z_i^{k-1}, y_i)$ to E. For all $1 \leq i < j \leq k$ add (y_i, x_j) to E. We have $n = k^2$ and $m = \frac{3}{2}k(k-1)$, so $m = \Theta(n)$. We refer to the y_i as *heads* and to the x_i as *tails* and let $X_i = \{x_i, ..., x_k\}$ and $Y_i = \{y_1, ..., y_i\}$.

Constructing $\prec$

For each $i \in [k]$, let $\prec^i \sim (x_i, z_i^2, ..., z_i^{k-1}, y_i)$. Now let $\prec$ we the ordering obtained by combining all of the $\prec^i$ and setting $y_{j+1} \prec x_j$ for all $j \in [k-1]$. We have that

$$\prec \sim (x_k, z_k^2, ..., z_k^{k-1}, y_k, ..., x_1, z_1^2, ..., z_1^{k-1}, y_1)$$

[1] We have introduced the graphs in a slightly different way to [HKM+12] to make them easier to work with, but they are fundamentally the same

Constructing $\mathcal{E}$

We will construct $\mathcal{E}$ by starting with the empty sequence and appending edges one by one. We do this in two distinct phases, and it is best represented by the following algorithm.

Algorithm 4: Insertion Sequence for H_k

```
1  E ← ();
2  ▷ PHASE 1;
3  for i = 1, ..., k do
4  |  ▷ Let ≺^i ∼ (u_1, ..., u_k);
5  |  for j = 1, ..., k − 1 do
6  |  |  E.append((u_j, u_{j+1}));
7  ▷ PHASE 2;
8  for i = k, ..., 1 do
9  |  for j = i − 1, ..., 1 do
10 |  |  E.append((y_j, x_i));
11 return E
```

Lower Bound

We shall now show that these instances leads to high recourse for any local algorithm.

Proposition 4.2.1. *Given any local algorithm $\mathcal{A}$, the run of $\mathcal{A}$ on H_k with insertion sequence $\mathcal{E}$ and initial ordering $\prec$ gives super-polylogarthic average recourse per edge insertion, more specifically, we have that*

$$rec(\mathcal{E}, \prec) = \Omega(n\sqrt{n})$$

Proof. Since in phase 1 we only insert edges (u, v) such that $u \prec v$, after all the edges in phase 1 have been inserted, the initial ordering $\prec$ is still a topological ordering of the graph. Hence, since a local algorithm can only rearrange nodes when an insertion goes against the topological ordering, we do not change the ordering or incur any recourse during any of these insertions.

The effect that each insertion in phase 2 has on the ordering is also completely determined by the fact that we are running local algorithm. During the insertion of the edge (y_j, x_i), we have the affected region is $(x_i, z_i^2, ..., z_i^{k-1}, y_i, x_j, z_j^2, ..., z_j^{k-1}, y_j)$. Since we want to find a topological ordering by only rearranging nodes in the affected region, and we now require $(x_j, z_j^2, ..., z_j^{k-1}, y_j, x_i, z_i^2, ..., z_i^{k-1}, y_i)$ to be a subsequence of the ordering, we can see that the effect the algorithm must have on the ordering is completely determined by the fact it is local. Since we cannot make this change to the ordering without making at least k node movements, we have that the algorithm incurs a recourse of $\Omega(k)$ during each insertion from phase 2. Since there are $\Omega(k^2)$ insertions in phase 2, the total recourse is $\Omega(k^3) = \Omega(n\sqrt{n})$. □

4.2.2 Randomizing High Recourse Instances

Since $\mathcal{A}_2$ is a local algorithm, it follows as an immediate corollary of Proposition 4.2.1 that $\mathcal{A}_2$ does not give $\tilde{\mathcal{O}}(m)$ (and hence $\tilde{\mathcal{O}}(n)$) recourse on all instances. However, it is believed that the expected recourse of $\mathcal{A}_2$ for these instances is $\tilde{\mathcal{O}}(n)$ under random-order arrival (or else they would be counterexamples disproving Conjecture 2.4.3). Informally, the idea here is that these instances are artificially created by an adversary in such a way that they exploit the properties of local algorithms to obtain high recourse, but by randomizing the order in which the edges are inserted, we weaken the ability of these instances to take advantage of these properties, leading to low expected recourse. We were able to come up with an intuitive proof for this,

even though the arguments are relatively involved computationally, they are conceptually quite straight forward. We now prove the following proposition.

Proposition 4.2.2. *Let $\prec$ be any initial ordering of H_k, then the recourse of $\mathcal{A}_2$ satisfies the following*

$$\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[rec(\mathcal{E},\prec)] = \tilde{\mathcal{O}}(n)$$

4.3 Proof of Proposition 4.2.2

Throughout this section we implicitly assume we are running all inputs on the simple greedy 2-way search algorithm, $\mathcal{A}_2$.

Fix some $k \in \mathbb{N}$ and let $G = H_k$. We will now show that in the random order arrival model, given any initial ordering of G, the expected recourse of $\mathcal{A}_2$ on G is $\tilde{\mathcal{O}}(n)$.

Fix some initial ordering $\prec \in \mathscr{S}_V$ and suppose we run $\mathcal{A}_2$ on this graph with some insertion sequence $\mathcal{E} \in \mathscr{S}_E$. We want to classify all the edge insertions into finitely many types and show that we expect $\tilde{\mathcal{O}}(n)$ total recourse from each type. Suppose we insert the edge $\mathcal{E}_t = (u, v)$ into the graph $G^{\mathcal{E}}_{t-1}$, then we classify this edge insertion as follows.

Type 1. If for some $i \in [k]$, $u, v \in S_i$ and $G^{\mathcal{E}}_t[S_i]$ is **not** weakly connected

Type 2. If for some $i \in [k]$, $u, v \in S_i$ and $G^{\mathcal{E}}_t[S_i]$ is weakly connected

Type 3. If u is a head and v is a tail

It should be clear that any edge insertion is exactly one of these 3 types. Denote by $rec^j(\mathcal{E},\prec)$ the total recourse of all type j insertions from the insertion sequence $\mathcal{E}$. So we have that $rec(\mathcal{E},\prec) = \sum_{j=1}^{3} rec^j(\mathcal{E},\prec)$ and hence $\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[rec(\mathcal{E},\prec)] = \sum_{j=1}^{3}\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[rec^j(\mathcal{E},\prec)]$ by linearity of expectation.

The proof of the proposition can split up into the 4 following components, which can all be combined to obtain the result.

I. Show that $rec^1(\mathcal{E},\prec) = \tilde{\mathcal{O}}(n)$ giving us that $\mathbb{E}_{\mathcal{E}}[rec^j(\mathcal{E},\prec)] = \tilde{\mathcal{O}}(n)$.

II. Show that the expected number of type 2 insertions in the first $m - k(\log n)^2$ insertions is $\leq k^{1-\frac{8}{3}\log k} \to 0$ quickly as $k \to \infty$.

III. Show that the expected total recourse of the type 3 insertions which occur before any type 2 insertions is $\tilde{\mathcal{O}}(n)$.

IV. Using the fact that type 3 insertions occur uniformly at random, bound the recourse for the last $k(\log n)^2$ insertions by $\tilde{\mathcal{O}}(n)$.

We can combine these claims to obtain the result we want as follows.

Proof of Proposition 4.2.2. Combining these results, we expect the total recourse from the first $m - k(\log n)^2$ insertions to be $\tilde{\mathcal{O}}(n)$ since: (I) type 1 insertions have an expected total recourse of $\tilde{\mathcal{O}}(n)$ and (II) with high probability there will be no type 2 insertions in this region which tells us that with high probability (III) the type 3 insertions in this region have an expected total recourse of $\tilde{\mathcal{O}}(n)$. Since we also have that (IV) the expected total recourse of the last $k(\log n)^2$ insertions is $\tilde{\mathcal{O}}(n)$, it follows that the expected total recourse is $\tilde{\mathcal{O}}(n)$. □

Throughout the rest of this section we will abbreviate $reach_{G^{\varepsilon}_t}$ to $reach_t$ to make things easier to read.

Step I

We first start off by breaking down type 1 insertions into 2 further types. Suppose we insert an edge like above, and it is classified as a type 1 insertion, then by the definition of a type 1 insertion, we can assume that for some $i \in [k]$, $u, v \in S_i$ and $G_t^{\mathcal{E}}[S_i]$ is not a weakly connected graph. We further classify this edge insertion as follows.

Type 1.1 neither of x_i or y_i is contained in $reach_t^{-1}(u) \cup reach_t(v)$

Type 1.2 one of x_i or y_i is contained in $reach_t^{-1}(u) \cup reach_t(v)$

It should again be clear that all type 1 insertions are exactly one of these 2 types, since if **both** x_i and y_i are contained in this set then $G_t^{\mathcal{E}}[S_i]$ is weakly connected. The weakly connected components of the graph $G_t^{\mathcal{E}}[S_i]$ are simple paths, so we can think of type 1 insertions as appending these paths together. Furthermore, we can think of type 1.1 insertions as appending paths that don't contain either x_i or y_i, and type 1.2 insertions as appending a path that doesn't contain either x_i or y_i with one that does.

Since the only nodes in S_i that may be weakly connected to nodes not contained in S_i are x_i and y_i, we can deduce that the recourse of a type 1.1 insertion is at most twice the length of the smallest path being appended (with the length being how many nodes it contains). Similarly, we can deduce that the recourse of a type 1.2 insertion is at most twice the length of the path being appended that doesn't contain either x_i or y_i.

Hence, within some set S_i, we get that the total recourse of the type 1.1 insertions is $\mathcal{O}(k \log k)$ by maximising this value, and similarly (but more obviously) the total recourse of the type 1.2 insertions is $\mathcal{O}(k)$. Adding up over all the S_i for all $i \in [k]$, we get that the total recourse of the type 1 insertions is $\tilde{\mathcal{O}}(n)$, i.e. $rec^1(\mathcal{E}, \prec) = \tilde{\mathcal{O}}(n)$. So we get $\mathbb{E}_{\mathcal{E}}[rec^1(\mathcal{E}, \prec)] = \tilde{\mathcal{O}}(n)$.

Step II

Given any S_i, what is the expected value of $|S_i \cap reach_t(x_i)|$? Equivalently, how many nodes in S_i do we expect x_i to be able to reach after t insertions? It should be clear that there is exactly one type 2 insertion per each set S_i, and that there has been a type 2 insertion in the set S_i if and only if $|S_i \cap reach_t(x_i)| = k$. Hence we get that the probability that there has been as type 2 insertion in the set S_i is equal to the probability that $|S_i \cap reach_t(x_i)| = k$. We denote the probability that $|S_i \cap reach_t(x_i)| = j$ for some $j \in \{0, ..., k\}$ by $p_j(t)$ (note that by symmetry the probability does not depend on the value of i, i.e. which set it is). Notice that by linearity of expectation it follows that the expected number of type 2 insertions after the first t insertions is $k \cdot p_k(t)$. It can easily be shown via standard combinatorial arguments that

$$p_0(t) = 0, \quad 1 \le j \le k-1 \quad p_j(t) = \binom{m-j}{t-j+1}\binom{m}{t}^{-1}, \quad p_k(t) = \binom{m-k+1}{t-k+1}\binom{m}{t}^{-1}$$

From now on we fix the value $\gamma = m - k(\log n)^2$. We now find the expected value of $k \cdot p_k(\gamma)$.

$$k \cdot p_k(\gamma) = k\binom{m-k+1}{\gamma-k+1}\binom{m}{\gamma}^{-1} = k \prod_{j=\gamma+1}^{m}\left(\frac{j-k+1}{j}\right)$$

$$\le k\left(\frac{m-k+1}{m}\right)^{m-\gamma} = k\left(1-\frac{2}{3k}\right)^{4k(\log k)^2} \le ke^{-\frac{8}{3}(\log k)^2} = k^{1-\frac{8}{3}\log k}$$

Step III

We can use the fact that we do not expect any type 2 insertions to occur within the first γ insertions to help us bound the expected recourse of the type 3 insertions in this region. An edge (u, v) causes a type 3 insertion if and only if u is a head and v is a tail. The recourse of such an insertion is at most the smallest of $|reach_t(v)|$ and $|reach_t^{-1}(u)|$. We expect this value to be less than k since we do not expect any type 2 insertions to have occurred, and hence any tail x_i to be able to reach its corresponding head y_i, and hence any node not in S_i. But how many nodes exactly do we expect it to be able to reach? This will be expected value of $|S_i \cap reach_t(x_i)|$ which is equal to $\sum_{i=0}^{k} k \cdot p_k(t)$. We now show that $\sum_{t=0}^{m}\sum_{i=0}^{k} k \cdot p_k(t) = \mathcal{O}(n \log n)$, hence the type 3 insertions that occur before any type 2 insertions have a total recourse of at most $\tilde{\mathcal{O}}(n)$, since their total recourse is upper bounded by $\mathcal{O}(n \log n)$. Since we expect no type 2 insertions to occur within the first γ insertions, it follows that we expect $\tilde{\mathcal{O}}(n)$ recourse from type 3 insertions in this region. Using standard combinatorial and analytic arguments, we can deduce the following inequality

$$\sum_{i=0}^{k} k \cdot p_k(t) = \left(k\binom{m-k+1}{t-k+1} + \sum_{i=0}^{k-1} i\binom{m-i}{t-i+1}\right)\binom{m}{t}^{-1}$$

$$= k\prod_{j=t+1}^{m}\left(\frac{j-k+1}{j}\right) + (m-t)\sum_{i=1}^{k-1}\frac{i}{m-i+1}\prod_{j=t+1}^{m}\left(\frac{j-i+1}{j}\right)$$

$$= k\prod_{j=1}^{k-1}\left(\frac{t-j+1}{m-j+1}\right) + (m-t)\sum_{i=1}^{k-1}\frac{i}{m-i+1}\prod_{j=1}^{i-1}\left(\frac{t-j+1}{m-j+1}\right)$$

$$\leq k\left(\frac{t}{m}\right)^{k-1} + \frac{m-t}{m-k+1}\sum_{i=1}^{k-1} i\left(\frac{t}{m}\right)^{i-1}$$

$$\leq k\left(1-\frac{m}{m-k+1}\right)\left(\frac{t}{m}\right)^{k-1} + \frac{m^2}{(m-t)(m-k+1)}\left(1-\left(\frac{t}{m}\right)^{k}\right)$$

$$\leq \frac{m^2}{(m-t)(m-k+1)}$$

Since the rational function on the last line is strictly increasing when considered as a function of t, we get that

$$\sum_{t=0}^{m}\sum_{i=0}^{k} k \cdot p_k(t) \leq k + \int_0^{m-1}\frac{m^2}{(m-t)(m-k+1)}dt = k + \frac{m^2 \log m}{m-k+1} = \mathcal{O}(n \log n)$$

Step IV

For any subgraph of H of G with $V(H) = V(G)$, define $\Psi^*(H) \triangleq \sum_{i=1}^{k} |(X_{i+1} \cap reach_H(y_i))|$. Notice that $\Psi^*(G_t^{\mathcal{E}})$ is at least the number of type 3 insertions that have occurred and that $\Psi^*(G) = \frac{1}{2}k(k-1)$.

Combining the fact that the $\frac{1}{2}k(k-1)$ edges that cause type 3 insertions are all predetermined (iff they are from a head to a tail), with the fact that edges are inserted uniformly at random, we get that we expect $\frac{1}{3}\gamma = \frac{1}{2}k(k-1) - \frac{1}{3}k(\log n)^2$ of the first γ insertions to be of

type 3. So we get that we expect $\Psi^*(G) - \Psi^*(G^{\mathcal{E}}_\gamma) \leq \frac{1}{3}k(\log n)^2$.[2] We show that if this is the case, then the recourse of the last $m-\gamma$ insertions is $\tilde{\mathcal{O}}(n)$.

Consider any edge (u, v) of the last $m-\gamma$ edges inserted into the graph, whose insertion causes the total recourse to increase by λ. Let $F, B \subseteq \{S_1, ..., S_k\}$ be the collections of the sets discovered by the algorithm during the forward and backwards searches respectively during the insertion. Set $f = |F|$ and $b = |B|$. Since each of the searches discovers at least $(\lambda-1)/2$ nodes and each S_i contains k nodes, we know that $f, b \geq (\lambda-1)/2k$. Note that F and B are disjoint, since if not, for some i we have $S_i \in F \cap B$, so eventually we will have $y_i \rightsquigarrow u \rightsquigarrow v \rightsquigarrow x_i \rightsquigarrow y_i$ giving us a cycle. Also note that at most one of the heads in sets from B could reach at most one of the tails in the sets from F before this insertion, but after the insertion all of the heads in the sets from B can reach all of the tails in the sets from F. This gives us that

$$\left(\frac{\lambda-1}{2k} - 1\right)^2 \leq (f-1)(b-1) \leq \Psi^*(G^{\mathcal{E}}_t) - \Psi^*(G^{\mathcal{E}}_{t-1})$$

Let $\lambda_{\gamma+1}, ..., \lambda_m$ be the recourse of the last $m-\gamma$ insertions. Applying the equation above we get that

$$\sum_{j=\gamma+1}^{m} \left(\frac{\lambda_j - 1}{2k} - 1\right)^2 \leq \sum_{j=\gamma+1}^{m} \Psi^*(G^{\mathcal{E}}_j) - \Psi^*(G^{\mathcal{E}}_{j-1}) = \Psi^*(G) - \Psi^*(G^{\mathcal{E}}_\gamma) \leq \frac{1}{3}k(\log n)^2$$

Applying a simple norm bound we get

$$\sum_{j=\gamma+1}^{m} \left(\frac{\lambda_j - 1}{2k} - 1\right) \leq \left((m-\gamma+1)\sum_{j=\gamma+1}^{m} \left(\frac{\lambda_j - 1}{2k} - 1\right)^2\right)^{\frac{1}{2}} \leq \frac{1}{\sqrt{3}}k(\log n)^2$$

Moving some terms to the right we get that $\sum_{j=\gamma+1}^{m} \lambda_j = \mathcal{O}(n(\log n)^2)$. Hence the total recourse of the last $k(\log n)^2$ insertions is expected to be $\tilde{\mathcal{O}}(n)$.

Remark 4.3.1. *It's worth noting that the same analysis performed with $\gamma = m - k\log n$ instead of $\gamma = m - k(\log n)^2$ gives that the expected total recourse is $\mathcal{O}(n \log n)$. However, with this larger value of γ, this analysis yields that the expected number of type 2 insertions in the first γ insertions is $\leq k^{-\frac{1}{3}}$, which converges to 0 much slower than $k^{1-\frac{8}{3}\log k}$ as $k \to \infty$.*

[2] Technically, since γ may not be an integer, $G^{\mathcal{E}}_\gamma$ may not be defined. But we can assume without loss of generality that $\gamma \in \mathbb{N}$ or replace it with $\lfloor\gamma\rfloor$.

Chapter 5

Recourse Bounds for Trees under Adversarial Arrival

While working on the conjectures that the simple algorithms are expected to have low total recourse under random order arrival, we discovered many interesting properties of the one-way search algorithm $\mathcal{A}_1$. We were able to use the properties that we found to make some progress on this problem, including a reduction from Conjecture 2.4.4 to a new setting where we can consider a fixed ordering (more on this in Chapter 6) and a proof of the fact that by randomizing the initial ordering before running $\mathcal{A}_1$ we obtain low expected total recourse under *adversarial* arrival on trees.[1]

This chapter will be devoted to constructing the framework which we will then use to prove these statements, and using it to prove Conjecture 2.4.1 for trees. More formally, we will be proving the following theorem.

Theorem 5.0.1. *Let $\mathcal{T} = (V, E)$ be a tree and $\mathcal{E}$ be any insertion sequence of $\mathcal{T}$, then the recourse of $\mathcal{A}_1$ satisfies the following*

$$\mathbb{E}_{\prec \in \mathscr{S}_V}[rec(\mathcal{E}, \prec)] = \mathcal{O}(n \log n)$$

It should be clear that a weaker version Conjecture 2.4.1 which only holds for trees follows immediately from this theorem, since the result for adversarial arrival implies the results for random order arrival. However, the result holding for adversarial arrival is much stronger than just random order arrival. We achieve this result by introducing our notion of *activation sequences* and a metric Γ on such sequences that allows us to characterize the recourse of $\mathcal{A}_1$ in terms of this metric. It turns out that for trees this yields a very natural characterization of recourse, which we will later extend to all DAGs to obtain the reduction stated above.

Remark 5.0.2. *The definition of a tree is something which can be extended from undirected graphs to directed graphs in various ways. In this project, we take the following definition. Given some directed graph $G = (V, E)$, we say that G is a tree if the undirected graph obtained by forgetting the directions of the edges in G is a tree.*

[1] Alternatively, we can interpret this result as saying that we can construct a *randomized algorithm* which has low expected total recourse on all trees

5.1 Properties of $\mathcal{A}_1$

We will start off by going over some of the properties of the one-way search algorithm $\mathcal{A}_1$. For the rest of this section fix a DAG $G = (V, E)$. We first give the definitions of *critical* and *right critical* edges which will be useful throughout the rest of the chapter.

Definition 5.1.1. *At some point during the run of $\mathcal{A}_1$ on G with some insertion sequence and initial ordering, we say an edge (u, v) is **critical** to w if its insertion would increase the amount of nodes that can reach w. Similarly, we say that an edge (u, v) is **right critical** to w if its insertion would cause the recourse of w to increase. We say u is (right) critical to w if there exists an edge $(u, v) \in E$ that is (right) critical to w.*

This definition has the following intuitive properties.

Proposition 5.1.2. *During the run of $\mathcal{A}_1$ on G, the insertion of an edge (u, v) can only increase the recourse of w if (u, v) is critical to w, i.e. all right critical edges (and nodes) to w are also critical to w.*

Proof. Suppose that the edge $e = (u, v)$ is not critical to w. Then the insertion of e does not increase the amount of ancestors of w, which tells us that either v can not reach w or u can already reach w. If v can not reach w, then clearly the insertion of e cannot cause w to move (and hence incur any recourse during its insertion). If u can already reach w, then u must be behind w in the current (topological) ordering, and hence the insertion of e cannot cause w to move. Either way, the insertion of the edge e cannot cause w to incur any recourse and hence is not right critical to w. □

Proposition 5.1.3. *Given any $u, v \in V$, if u is **not** an ancestor of v in G, then u can never be critical to v during the run of $\mathcal{A}_1$ on G.*

Proof. If u is critical to v, then there exists some edge $e = (u, w) \in E$ such that e is critical to v. This means that w is an ancestor of v and u is not an ancestor of w in the current graph. This tells us that w is an ancestor of v in G and clearly u is an ancestor of w in G since $(u, w) \in E$. It follows that u is an ancestor of v in G. □

Combining these two properties we can get the following useful lemma. Let $x \in V$ and fix an insertion sequence $\mathcal{E}$ and an initial ordering $\prec$ for G, and use these to induce an insertion sequence $\mathcal{E}^*$ and an initial ordering $\prec^*$ for $H = G[reach_G^{-1}(x)]$, such that $\mathcal{E}^*$ and $\prec^*$ are subsequences of $\mathcal{E}$ and $\prec$ respectively.

Lemma 5.1.4. *Given some $u \in V(H)$, the recourse of u during the run of $\mathcal{A}_1$ on G equals the recourse of u during the run of $\mathcal{A}_1$ on H with respect to the insertion sequences and initial orderings defined above.*

Proof. Let $E^* = E(G) \setminus E(H)$. We want to show that the relative ordering of the nodes in $V(H)$ cannot be affected by the insertion of edges in E^* and only depends on the order of insertions of the edges in $E(H)$.

By combining Propositions 5.1.2 and 5.1.3, we can see that the insertion of an edge (u, v) can only cause the recourse of some node w to increase if v can reach w. Suppose that the insertion of some $e = (u, v) \in E^*$ causes the recourse of $w \in V(H)$ to increase. Then we get that $u, v \in V(H)$, so $e \in E(H)$, giving us a contradiction.

Suppose we insert some $e = (u, v) \in E(H)$ into the graph. The algorithm decides which nodes to move by performing a one-way search starting from v, and moves them in such a way that the relative ordering of all nodes that are moved is preserved. Since the presence of any

edges from E^* in the graph cannot change the nodes in $V(H)$ that the search is able to find, we can deduce that their presence in the graph will not change the effect of the algorithm on the relative ordering of the nodes in $V(H)$, and that this will depend purely on the current relative ordering of the nodes in $V(H)$ and which edges from $E(H)$ had already been inserted before the insertion occurred. □

Intuitively, Lemma 5.1.4 is saying that the effect of $\mathcal{A}_1$ on some node u depends *only* on the structure of the ancestors of u. Because of this, if we are only looking at the behaviour of one specific node in the graph, we are allowed to ignore every node that isn't one of it's ancestors.

Our strategy in this chapter will be to upper bound the expected recourse of each node in the graph individually, then use linearly of expectation to add up these values and obtain an upper bound on the expected total recourse of the graph. Since we only care about the recourse of one node, using Lemma 5.1.4, we can make the following assumption.

Proposition 5.1.5. *We can assume without loss of generality that there exists some root node r such that $G = G[reach_G^{-1}(r)]$.*

Proof. Since we are only concerned with bounding the expected recourse of some node r, we can assume that the only nodes in the graph are ancestors of r, since removing a node $u \notin reach_G^{-1}(r)$ from G will not change the recourse of r during the run of $\mathcal{A}_1$ on G. □

Suppose we show that we expect the recourse of r to be low for any insertion sequence, then we expect the recourse of any node x to be low, and Theorem 5.0.1 will follow.

$$\mathbb{E}_{\prec \in \mathscr{S}_V}[rec_r(\mathcal{E}, \prec)] = \mathcal{O}(\log n) \implies \mathbb{E}_{\prec \in \mathscr{S}_V}[rec(\mathcal{E}, \prec)] = \mathcal{O}(n \log n)$$

Of course, we will need to apply concentration bounds to make this result useful.

5.2 Activation Sequences

Fix a DAG $G = (V, E)$ with root r. Throughout this section we implicitly assume we are running all inputs on $\mathcal{A}_1$.

We will start off by introducing the notion of *activation sequences* and a function Γ which will be at the core of our proof of Theorem 5.0.1.

Definition 5.2.1 (Node Activation)**.** *Given some $u \in V \setminus \{r\}$ and an insertion sequence $\mathcal{E}$ of G, we say that u is **activated** at time t, if $u \in reach_t^{-1}(r)$ and $u \notin reach_{t-1}^{-1}(r)$. We take the convention that r is activated at time 0.*[2]

Definition 5.2.2 (Activation Sequence)**.** *Let $\mathcal{E}$ be any insertion sequence of G. The **activation sequence** of $\mathcal{E}$, denoted by $\alpha = act(\mathcal{E})$, is a sequence of length m of potentially empty sets that form a partition of V, where α_t is the set of nodes activated at time t.*

Here, the set $\alpha_t \subseteq V$ is the set of nodes that are able to reach r for the first time after the insertion of $\mathcal{E}_t$. We can see that $act(\mathcal{E})$ depends only on the graph and the insertion sequence, not the initial ordering. We now make a simple but useful observation about activation sequences.

Proposition 5.2.3. *Let $\mathcal{E}$ be an insertion sequence of G with activation sequence α. If $(x, y) \in E$ is inserted before y is activated, then x is not activated after y.*

[2] As usual, we abbreviate $reach_{G_t^{\mathcal{E}}}$ to $reach_t$.

Proof. Suppose that $(x, y) \in E$ is inserted before y is activated. When y is activated, x will be able to reach y which can reach r, so x can reach r. So it follows that if x is not already active when y is activated, then they will be activated simultaneously. □

We now define the following useful function on activation sequences.

Definition 5.2.4. *Let $\Gamma : act(\mathscr{S}_E) \times \mathscr{S}_V \longrightarrow \mathbb{N}$ be a map such that given some insertion sequence $\mathcal{E}$ of G with activation sequence α and an ordering $\prec$ of V, $\Gamma(act(\mathcal{E}), \prec)$ is the length of the sequence of nodes $(u_j)_{j=1}^k$ defined recursively by*

$$u_0 = r, \quad u_i \in \alpha_{\phi_i}, \forall u \in \alpha_{\phi_i}, u \preceq u_i$$

$$\phi_0 = 0, \quad \phi_{i+1} = min\{j \mid \exists u \in \alpha_j, u_i \prec u, \phi_i < j\}$$

We can see that u_i is defined if and only if $\phi_i < \infty$ (using $min\varnothing = \infty$) and that $(\phi_j)_{j=1}^k$ is a strictly increasing sequence. It should be clear that $\Gamma \in [n]$ for all inputs.

Initially this function Γ may seem slightly obscure, unlike the definition of the activation sequence which is more natural and straightforward. Intuitively, this function is counting how many times the rightmost activated node changes with respect to the fixed ordering $\prec$ as we make insertions. An illustration of this is given in figure 5.1.

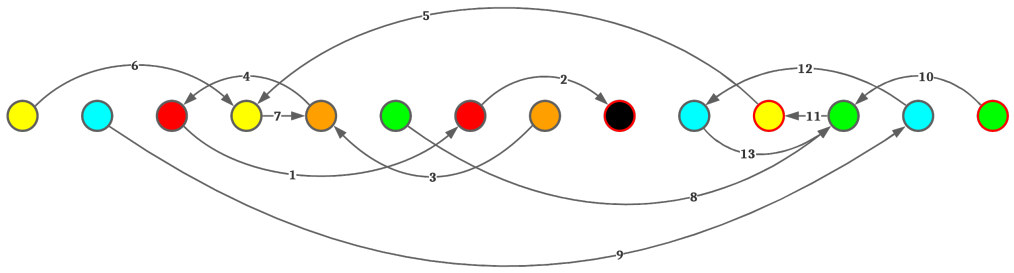

Figure 5.1: This figure represents an activation sequence α with the different coloured nodes; the black, red, orange, yellow, lime and cyan nodes represent $\alpha_{\phi_0}, ..., \alpha_{\phi_5}$ respectively. The nodes outlined in red are the nodes contained in the sequence $(u_j)_{j=0}^k$, the function Γ counts exactly these nodes, excluding the root node, which is the black node in this diagram.

We will now prove a lemma giving a tight bound on the expected value of Γ over random initial orderings.

Lemma 5.2.5. *Let $\mathcal{E}$ be any insertion sequence of G, then*

$$\mathbb{E}_{\prec \in \mathscr{S}_V}[\Gamma(act(\mathcal{E}), \prec)] = \mathcal{O}(\log n)$$

Of course, we must also show that this equality holds with high probability, which will require us to apply concentration bounds. In order to do this we will apply a standard Chernoff bound argument using the following theorem.

Theorem 5.2.6 (Chernoff-Hoeffding Bound)**.** *Let $X = \sum_{i=1}^n X_i$ where $\{X_i\}_{i \in [n]}$ are independent $\{0, 1\}$-random variables. Then for $0 < \varepsilon < 1$ we have*

$$\mathbb{P}[X < (1 - \varepsilon)\mathbb{E}[X]] \leq \exp\Big(-\frac{\varepsilon^2}{2}\mathbb{E}[X]\Big)$$

Proof of Lemma 5.2.5. To make arguments simpler we assume that all nodes are activated at different times. Since this can only increase the values of Γ, the upper bounds derived from this assumption will still hold. Let the sequence $(u_j)_{j=0}^{k}$ be defined as in 5.2.4. Note that is sequence is not defined in this context since we have not fixed an initial ordering, but we can use this sequence do define the following random variable. Let n_t be a random variable equal to the number of nodes u such that $u_t \prec u$ if u_t is defined and 0 if not. Let X_t be the indicator random variable for the event that $n_{t+1} \leq n_t/2$. Now suppose we generate an initial ordering uniformly at random and we have obtained $u_1, ..., u_t$ and $n_t > 0$. Then u_{t+1} is equally likely to be anywhere at any position in the ordering to the right of u_t (note that this requires the assumption we made at the start of the proof) and hence we get that $\mathbb{E}[n_{t+1}] \leq \mathbb{E}[n_t]/2$, giving us $\mathbb{E}[X_t] \geq \frac{1}{2}$. Let $c \geq 10$ be some constant. If we show that $n_{c\cdot\log_2(n)} \leq 1$ with high probability (w.h.p), then we have $n_{c\cdot\log_2(n)+1} = 0$ which implies $\Gamma \leq c \cdot \log_2(n) + 1$ w.h.p and we are done. We prove this with the following claim, which follows from applying Chernoff bounds.

claim 5.2.7. *Let $Y_t = X_1 + ... + X_t$, then $\mathbb{P}[Y_{c\cdot\log_2(n)} \geq \log_2(n)] \geq 1 - n^{-\frac{c}{4}(1-\frac{2}{c})^2}$.*

Proof. Applying linearity of expectation we can see that $\mathbb{E}[Y_{c\cdot\log_2(n)}] \geq \frac{c}{2}\cdot\log_2(n)$. Let $\varepsilon = 1-\frac{2}{c}$, then we can apply Chernoff bounds to obtain

$$\mathbb{P}[Y_{c\cdot\log_2(n)} < \log_2(n)] = \mathbb{P}[Y_{c\cdot\log_2(n)} < (1-\varepsilon)\mathbb{E}[Y_{c\cdot\log_2(n)}]] \leq \exp\Big(-\frac{\varepsilon^2}{2}\mathbb{E}[Y_{c\cdot\log_2(n)}]\Big)$$

$$\leq \exp\Big(-\frac{\varepsilon^2}{2}\cdot\frac{c}{2}\cdot\log_2(n)\Big) \leq n^{-\frac{c}{4}(1-\frac{2}{c})^2}$$

Hence, we get $\mathbb{P}[Y_{c\cdot\log_2(n)} \geq \log_2(n)] \geq 1 - n^{-\frac{c}{4}(1-\frac{2}{c})^2}$. □

Now fix $c = 20$ and we get that

$$n_{20\cdot\log_2(n)} \leq n \cdot 2^{-Y_{20\cdot\log_2(n)}} \leq n \cdot 2^{-\log_2(n)} = 1$$

holds with probability at least $1 - \frac{1}{n^4}$ so it follows that $\Gamma \leq 20\log_2(n) + 1$ with the same probability. It follows that the expected value of Γ is $\mathcal{O}(\log n)$.

□

5.3 Activation Sequences for Trees

Fix a tree $\mathcal{T} = (V, E)$ with root r. Throughout this section we implicitly assume we are running all inputs on $\mathcal{A}_1$.

The activation sequences of rooted trees take a very particular form because all non-root nodes have an out-degree of exactly 1. This allows us to deduce the following lemma, making it easy to show that the expected total recourse of $\mathcal{A}_1$ on trees under adversarial insertion is low.

Lemma 5.3.1. *Let $\mathcal{E}$ be any insertion sequence of $\mathcal{T}$ and $\prec$ any initial ordering, then*

$$rec_r(\mathcal{E}, \prec) = \Gamma(act(\mathcal{E}), \prec)$$

Before proving this we make the following simple observation.

Proposition 5.3.2. *Let $\mathcal{E}$ be an insertion sequence of $\mathcal{T}$ with activation sequence α. If $(x, y) \in E$, then y is not activated after x.*

Proof. Since $\mathcal{T}$ is a tree, there is a unique path from x to r and it includes y. Hence, when x is activated (i.e. when an insertion causes x to be able to reach r from the first time), we know that y must either be activated simultaneously or is already active. □

Proof of Lemma 5.3.1. Consider the set of nodes $\mathcal{U} = \{u_1, ..., u_k\}$ defined in Definition 5.2.4. Given some $u \in \mathcal{U}$, suppose that the insertion of $\mathcal{E}_t$ which causes u to be activated does not cause r to move. It follows that there exits some node $v \in V$ satisfying $u \prec_{t-1} v$ that is active at time $t-1$, since if this were not the case, r would never have moved to the right of u in the ordering. By the construction of $(u_j)_{j=1}^k$, we can see that $v \prec u$, or else u would not be contained in $\mathcal{U}$. Combining this with the fact that all nodes w satisfying $u \prec w$ are activated after u (since again this would contradict $u \in \mathcal{U}$), it follows that at some point some edge (x, y) is inserted before y is activated such that y is activated before x, or else this would contradict the existence of v, but this contradicts Proposition 5.2.3. It follows that the activation of all nodes in $\mathcal{U}$ cause r to move and hence incur a recourse. Since there are $\Gamma(act(\mathcal{E}), \prec)$ nodes in $\mathcal{U}$ which are all activated at different times we get that $rec_r(\mathcal{E}, \prec) \geq \Gamma(act(\mathcal{E}), \prec)$.

We now want to show that r is only moved by the critical insertions which cause the nodes in $\mathcal{U}$ to be activated. Suppose that this is not the case. Then at some point we must insert an edge (x, y) such that y is activated after x, or else each right critical insertion must also activate a node in $\mathcal{U}$, but this contradicts Proposition 5.3.2. It follows that r can only be moved by such insertions and hence that $rec_r(\mathcal{E}, \prec) \leq \Gamma(act(\mathcal{E}), \prec)$. □

While this lemma does not hold for DAGs in general, the first part of the proof follows from Proposition 5.2.3, so we get that $\Gamma(act(\mathcal{E}), \prec)$ is a lower bound on the recourse of the root for any DAG. In Chapter 6 we give a generalization of this lemma which holds for all DAGs.

Now we can combine Lemma 5.2.5 and Lemma 5.3.1 to obtain the following theorem.

Theorem 5.3.3. *Let $\mathcal{E}$ be any insertion sequence of $\mathcal{T}$, then*

$$\mathbb{E}_{\prec \in \mathscr{S}_V}[rec_r(\mathcal{E}, \prec)] = \mathcal{O}(\log n)$$

Proof. Now we can combine Lemma 5.2.5 and Lemma 5.3.1 to obtain the following equality

$$\mathbb{E}_{\prec \in \mathscr{S}_V}[rec_r(\mathcal{E}, \prec)] = \mathbb{E}_{\prec \in \mathscr{S}_V}[\Gamma(act(\mathcal{E}, \prec))] = \mathcal{O}(\log n)$$

We also get that $\mathbb{P}[rec_r(\mathcal{E}, \prec) \leq 20 \log_2(n) + 1] \geq 1 - \frac{1}{n^4}$ from the concentration bounds in Lemma 5.2.5. □

We can now bound the expected total recourse of $\mathcal{A}_1$ with a random initial ordering on trees under adversarial by $\mathcal{O}(n \log n)$ by applying the argument to each node individually.

Corollary 5.3.4. *Let $\mathcal{E}$ be any insertion sequence of $\mathcal{T}$, then*

$$\mathbb{E}_{\prec \in \mathscr{S}_V}[rec(\mathcal{E}, \prec)] = \mathcal{O}(n \log n)$$

Proof. The total expected recourse is the sum of the expected recourse of every node, so by linearity of expectation we get

$$\mathbb{E}_{\prec \in \mathscr{S}_V}[rec(\mathcal{E}, \prec)] = \sum_{u \in V} \mathbb{E}_{\prec \in \mathscr{S}_V}[rec_u(\mathcal{E}, \prec)] = \sum_{u \in V} \mathcal{O}(\log n) = \mathcal{O}(n \log n)$$

In order to make sure this holds with high probability, we can union bound and apply the concentration bounds obtained in the proof of Theorem 5.3.3 to get that

$$\mathbb{P}[rec_u(\mathcal{E},\prec) > 20\log_2(n)+1 \text{ for some } u \in V] \leq \sum_{u\in V} \mathbb{P}[rec_u(\mathcal{E},\prec) > 20\log_2(n)+1] \leq \frac{n}{n^4} = \frac{1}{n^3}$$

since $\mathbb{P}[rec_u(\mathcal{E},\prec) > 20\log_2(n)+1] \leq \frac{1}{n^4}$ follows from the fact that $|reach^{-1}(u)| \leq n$. Hence, we get that $\mathbb{P}[rec_u(\mathcal{E},\prec) \leq 20\log_2(n)+1 \text{ for all } u \in V] \geq 1 - \frac{1}{n^3}$ which implies that

$$\mathbb{P}[rec(\mathcal{E},\prec) \leq 20n\log_2(n)+n] \leq 1 - \frac{1}{n^3}$$

□

Chapter 6

Extending the Activation Sequence Framework

In the previous chapter we saw that Lemma 5.3.1 gave us an equivalence between values of the function Γ and the recourse of $\mathcal{A}_1$ on trees. We also noted that this could be used to obtain a lower bound for the recourse of $\mathcal{A}_1$ on any DAG. One of the main result of this chapter will be an extension of this result to DAGs, allowing us fully characterize the recourse of $\mathcal{A}_1$ on any DAG in terms of values of Γ. Using this result we will give a reduction from Conjecture 2.4.4 to a conjecture analogous to Lemma 5.2.5.

This reduction is interesting because it will allows us to consider a setting where the ordering does not change, even though we are analysing the recourse of an incremental topological ordering algorithm, which by definition requires the ordering to change. Because of this, this reduction will allow us to work in a conceptually simpler setting where we only need to concern ourselves with the structure of the edges being inserted into the graph and not with the structure of the ordering which is fixed. Furthermore, the current state of the art for sparse graphs requires us to start with the empty graph in order to obtain the algorithmic guarantees, so a result that does not require such a strict assumption would be very interesting. This reduction gives us a way to directly tackle this problem.

6.1 Incomplete Insertion and Activation Sequences

Before continuing with the results in this chapter, it will be helpful to introduce some new notation to make things easier to read. Let $G = (V, E)$ be a DAG with root r. In Definition 5.2.4, we define a sequence of nodes, $(u_j)_{j=1}^{k}$, given some insertion sequence $\mathcal{E}$ and an ordering $\prec$, and we let $\Gamma(act(\mathcal{E}), \prec) = k$. If some node $u \in V$ is contained in the sequence $(u_j)_{j=1}^{k}$, we say that it is *counted* by $\Gamma(act(\mathcal{E}), \prec)$. We say this in a more compact way as follows.

Definition 6.1.1. *Given some $\mathcal{E} \in \mathscr{S}_E$, $\prec \in \mathscr{S}_V$, $u \in V$, we say that u is **relevant** in $\prec$ with respect to $\mathcal{E}$ if $\Gamma(act(\mathcal{E}), \prec)$ counts u.*

In the propositions and proofs throughout this chapter, we want to be able to consider the state of the graph and the ordering after some fixed edges have been inserted, so we must extend our notation to be well defined in this setting. We define **incomplete insertion sequences** of G to be proper subsequences of insertion sequences of G. We define **incomplete activation sequences** to be the activation sequences of incomplete insertion sequences. Note that the

length of an incomplete activation sequence is less than m and that its elements many not partition V. We also extend the domain of Γ to incomplete activation sequences.

Let $\mathcal{E}$ and $\mathcal{F}$ be incomplete insertion sequences of G of length k_1 and k_2 respectively such that $\{\mathcal{E}_i\}$ and $\{\mathcal{F}_i\}$ are disjoint. Let $\mathcal{E}\mathcal{F}$ denote the potentially incomplete insertion sequence obtained by appending $\mathcal{F}$ to the end of $\mathcal{E}$. Then we denote the incomplete activation sequence of $\mathcal{F}$ starting after the insertion of $\mathcal{E}$ by $act(\mathcal{E}, \mathcal{F})$. Note that $act(\mathcal{E}, \mathcal{F}) = (act(\mathcal{E}\mathcal{F})_{k_1+1}, ..., act(\mathcal{E}\mathcal{F})_{k_1+k_2})$ and that $\Gamma(act(\mathcal{E}, \mathcal{F}), \prec) = \Gamma(act(\mathcal{E}\mathcal{F}), \prec) - \Gamma(act(\mathcal{E}), \prec)$.

6.2 Extending Lemma 5.3.1 to DAGs

Throughout this whole chapter we implicitly assume we are running all inputs on $\mathcal{A}_1$ and fix a DAG $G = (V, E)$ with root r and initial ordering $\prec \in \mathscr{S}_V$.

We now give a generalization of Lemma 5.3.1 that gives us an equation relating the recourse of any DAG to values of Γ. The difference between this new equation and the equation given in Lemma 5.3.1 is the presence of a term which is always equal to 0 when G is a tree.

Proposition 6.2.1. *Let $x^+ = max(x, 0)$. Given any insertion sequence $\mathcal{E}$ of G we have that*

$$rec_r(\mathcal{E}, \prec) = \sum_{t=1}^{m} \left(\Gamma(act(\mathcal{E}), \prec_t) - \Gamma(act(\mathcal{E}), \prec_{t-1})\right)^+ + \Gamma(act(\mathcal{E}), \prec_0)$$

Here we let $\prec_t$ denote the ordering computed by $\mathcal{A}_1$ after the first t edges of $\mathcal{E}$ have been inserted. Note that $\prec = \prec_0$. The following lemma will be useful and allow us to give a nicer proof of this proposition.

Lemma 6.2.2. *Given any insertion sequence $\mathcal{E}$ of G we have that*

$$\Gamma(act(\mathcal{E}), \prec_t) < \Gamma(act(\mathcal{E}), \prec_{t-1}) \implies \Gamma(act(\mathcal{E}), \prec_t) = \Gamma(act(\mathcal{E}), \prec_{t-1}) - 1$$

Proof. Let $\mathcal{U} = \{u_1, ..., u_k\}$ be the set of nodes relevant in $\prec_{t-1}$ with respect to $\mathcal{E}$. Given any $u_i, u_j \in \mathcal{U}$, we know that u_i and u_j are activated at different times. We also know that given any $u \in \mathcal{U}$ we have that $u \prec_t r$ if and only if u is activated at time t. Now suppose that $\Gamma(act(\mathcal{E}), \prec_t) < \Gamma(act(\mathcal{E}), \prec_{t-1}) - 1$, then there are (at least) two nodes $u_i, u_j \in \mathcal{U}$ that are not relevant in $\prec_t$ with respect to $\mathcal{E}$, this means that both u_i and u_j were activated at time t, but u_i and u_j must be activated at different times, so we get a contradiction. It follows that $\Gamma(act(\mathcal{E}), \prec_t) \geq \Gamma(act(\mathcal{E}), \prec_{t-1}) - 1$, so we have that $\Gamma(act(\mathcal{E}), \prec_t) = \Gamma(act(\mathcal{E}), \prec_{t-1}) - 1$. □

Proof of Proposition 6.2.1. Fix some $\mathcal{E} \in \mathscr{S}_E$. By the construction of Γ, after the insertion of the edge $\mathcal{E}_{t-1}$, we have that $\Gamma(act(\mathcal{E}), \prec_t) < \Gamma(act(\mathcal{E}), \prec_{t-1})$ if and only if r is moved during the insertion of $\mathcal{E}_{t-1}$. By Lemma 6.2.2, this tells us that

$$rec_r(\mathcal{E}, \prec)_t = 1 \iff \Gamma(act(\mathcal{E}), \prec_t) = \Gamma(act(\mathcal{E}), \prec_{t-1}) - 1$$

So we can form the equation

$$rec_r(\mathcal{E}, \prec) = \sum_{t=1}^{m} \left(\Gamma(act(\mathcal{E}), \prec_{t-1}) - \Gamma(act(\mathcal{E}), \prec_t)\right)^+$$

Since the expression on the right hand side counts the total amount that $\Gamma(act(\mathcal{E}), \prec_t)$ *decreases* as we increase t from 1 to m, and we know that $\Gamma(act(\mathcal{E}), \prec_m) = 0$, we can see that this is

equal to the starting value $\Gamma(act(\mathcal{E}), \prec_0)$ plus the total amount that $\Gamma(act(\mathcal{E}), \prec_t)$ *increases* as we increase t from 1 to m. Hence, we get that

$$rec_r(\mathcal{E}, \prec) = \sum_{t=1}^{m} \left(\Gamma(act(\mathcal{E}), \prec_t) - \Gamma(act(\mathcal{E}), \prec_{t-1})\right)^+ + \Gamma(act(\mathcal{E}), \prec_0)$$

□

6.3 Γ Under Random-Order Arrival

We will now introduce a new conjecture about the values taken by Γ under random order arrival with respect to some fixed ordering. We will then show that this conjecture implies Conjecture 2.4.4.

Conjecture 6.3.1. *Let $F \subseteq E$, $\mathcal{E} \in \mathscr{S}_{E\setminus F}$, $\prec \in \mathscr{S}_V$, then we have that*

$$\mathbb{E}_{\mathcal{F}\in\mathscr{S}_F}[\Gamma(act(\mathcal{E}, \mathcal{F}), \prec)] = \text{polylog}\,|F|$$

The statement holding for arbitrary $\mathcal{E}$ is equivalent to the statement holding in expectation over all $\mathcal{E} \in \mathscr{S}_{E\setminus F}$ since for any $\mathcal{E}, \mathcal{E}^\star \in \mathscr{S}_{E\setminus F}$ we have $\Gamma(act(\mathcal{E}, \mathcal{F}), \prec) = \Gamma(act(\mathcal{E}^\star, \mathcal{F}), \prec)$. However, the fact that the statement holds for arbitrary $F \subseteq E$ is actually quite a strong restriction. In fact, Conjecture 6.3.1 is actually equivalent to a strengthened version of Conjecture 2.4.4 in which we start from an arbitrary subgraph H of G and an arbitrary topological ordering of H before inserting the remaining edges uniformly at random instead of starting from the empty graph $(V, \varnothing)$. Even though this restriction is strong, it leads to a very natural conjecture with nice properties leading to an elegant reduction; whereas attempting to relax this condition makes things much messier.

The current state of the art for sparse graphs, by [BK20], requires us to start with the empty graph in order to obtain the algorithmic guarantees. Hence, this reduction gives us a way to directly tackle this problem making it quite interesting.

For the rest of this chapter, any lemma or theorem marked with an asterisk follows from assuming that Conjecture 6.3.1 is true. Technically this means they aren't actually lemmas or theorems, but this is more elegant than labelling them all conjectures.

Lemma* 6.3.2. *Given any $\prec \in \mathscr{S}_V$ we have that*

$$\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[\Gamma(act(\mathcal{E}), \prec)] = \text{polylog}\, n$$

Proof. Follows directly from Conjecture 6.3.1 by setting $F = E$. □

We can see that Lemma* 6.3.2 is analogous to Lemma 5.3.1, the only relevant difference is that here we are fixing the ordering and taking expectation of insertion sequences instead of fixing the insertion sequence and taking expectation over orderings.

We now define some useful functions that will make the notation easier to read.

Definition 6.3.3. *Let $\mathcal{E} \in \mathscr{S}_E$, $t \in [m]$, we define the following functions.*

1. *Let $\chi_t(\mathcal{E}, \prec)$ be the indicator function for the event that $\Gamma(act(\mathcal{E}), \prec_t) > \Gamma(act(\mathcal{E}), \prec_{t-1})$*
2. *Let $\lambda_t(\mathcal{E}, \prec)$ be the indicator function for the event that $\mathcal{E}_t = (u, v)$ where u is relevant in $\prec_{t-1}$ with respect to $\mathcal{E}$*

3. *Let* $\Lambda_t(\mathcal{E}, \prec) = \left(\Gamma(act(\mathcal{E}), \prec_t) - \Gamma(act(\mathcal{E}), \prec_{t-1})\right)^+$

Proposition 6.3.4. *Given any potentially incomplete insertion sequence* $\mathcal{E}$ *of* G *of length* t*, we have that*

$$\Gamma(act(\mathcal{E}), \prec_t) = 0$$

Proof. Suppose that $\Gamma(act(\mathcal{E}), \prec_t) > 0$, then some node u is relevant in $\prec_t$ with respect to $\mathcal{E}$. This implies that $r \prec_t u$ and that for some $i \leq t$ the insertion of $\mathcal{E}_i$ causes u to be activated, but after u is activated we know that it must appear before r in the ordering, in other words, for all $j \geq i$ we have $u \prec_j r$. But this tells us that $u \prec_t r$, giving us a contradiction. □

Proposition 6.3.5. *Let* $\mathcal{E} \in \mathscr{S}_E$*,* $t \in [m]$*, then*

$$\chi_t(\mathcal{E}, \prec) \leq \lambda_t(\mathcal{E}, \prec)$$

Proof. Suppose that $\chi_t(\mathcal{E}, \prec) = 1$, i.e. that $\Gamma(act(\mathcal{E}), \prec_t) > \Gamma(act(\mathcal{E}), \prec_{t-1})$. Then the insertion of $\mathcal{E}_t = (u, v)$ causes some node w which is relevant in $\prec_t$ but not in $\prec_{t-1}$ to incur recourse. Since $w \in reach_t(u)$ and neither are active at time t, we get that u is not activated after w, so since w is relevant in $\prec_t$ we get that u is relevant in $\prec_{t-1}$, i.e. that $\lambda_t(\mathcal{E}, \prec) = 1$. Hence, $\chi_t(\mathcal{E}, \prec) \leq \lambda_t(\mathcal{E}, \prec)$. □

We now give two lemmas* and use them to prove that Conjecture 6.3.1 implies Conjecture 2.4.4. For the rest of the chapter we assume that the maximum degree of any node in G is $\Delta = \mathcal{O}(m/n)$.

Lemma* 6.3.6. *Let* $F \subseteq E$*,* $\mathcal{E} = (\mathcal{E}_1, ..., \mathcal{E}_{t-1}) \in \mathscr{S}_{E \setminus F}$*, then we have that*

$$\mathbb{E}_{\mathcal{F} \in \mathscr{S}_F}[\lambda_t(\mathcal{E}\mathcal{F}, \prec)] \leq \frac{\Delta \operatorname{polylog} n}{m - t + 1}$$

Lemma* 6.3.7. *Let* $F \subseteq E$*,* $\mathcal{E} = (\mathcal{E}_1, ..., \mathcal{E}_{t-1}) \in \mathscr{S}_{E \setminus F}$*, then we have that*

$$\mathbb{E}_{\mathcal{F} \in \mathscr{S}_F}[\Lambda_t(\mathcal{E}\mathcal{F}, \prec) | \lambda_t(\mathcal{E}\mathcal{F}, \prec) = 1] = \operatorname{polylog} n$$

Theorem* 6.3.1. $\mathbb{E}_{\mathcal{E} \in \mathscr{S}_E}[rec(\mathcal{E}, \prec)] = \tilde{\mathcal{O}}(m)$

Proof. By setting $F = E$ we can see that Lemma* 6.3.6 implies that

$$\mathbb{E}_{\mathcal{E} \in \mathscr{S}_E}[\lambda_t(\mathcal{E}, \prec)] \leq \frac{\Delta \operatorname{polylog} n}{m - t + 1}$$

and that Lemma* 6.3.7 implies that

$$\mathbb{E}_{\mathcal{E} \in \mathscr{S}_E}[\Lambda_t(\mathcal{E}, \prec) | \lambda_t(\mathcal{E}, \prec) = 1] = \operatorname{polylog} n$$

Since $\chi_t = 0$ if and only if $\Lambda_t = 0$, and $\chi_t \leq \lambda_t$ by Proposition 6.3.5, we can see that $\mathbb{E}_{\mathcal{E} \in \mathscr{S}_E}[\Lambda_t(\mathcal{E}, \prec) | \lambda_t(\mathcal{E}, \prec) = 0] = 0$. Applying the law of total expectation we get that

$$\mathbb{E}_{\mathcal{E} \in \mathscr{S}_E}[\Lambda_t(\mathcal{E}, \prec)] = \mathbb{E}_{\mathcal{E}}[\Lambda_t(\mathcal{E}, \prec) | \lambda_t(\mathcal{E}, \prec) = 1] \cdot \mathbb{P}[\lambda_t(\mathcal{E}, \prec) = 1]$$

$$\leq \mathbb{E}_{\mathcal{E}}[\Lambda_t(\mathcal{E}, \prec) | \lambda_t(\mathcal{E}, \prec) = 1] \cdot \frac{\Delta \operatorname{polylog} n}{m - t + 1} = \frac{\Delta \operatorname{polylog} n}{m - t + 1}$$

We can now take expectations over insertion sequences on both sides of the equation in Proposition 6.2.1 to obtain

$$\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[rec_r(\mathcal{E},\prec)] = \sum_{t=1}^{m}\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[\Lambda_t(\mathcal{E},\prec)] + \mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[\Gamma(act(\mathcal{E}),\prec)]$$

$$\leq \sum_{t=1}^{m}\frac{\Delta\,\mathrm{polylog}\,n}{m-t+1} + \mathrm{polylog}\,n = \Delta\,\mathrm{polylog}\,n$$

So it follows that

$$\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[rec(\mathcal{E},\prec)] = \sum_{x\in V}\mathbb{E}_{\mathcal{E}\in\mathscr{S}_E}[rec_x(\mathcal{E},\prec)] = \sum_{x\in V}\Delta\,\mathrm{polylog}\,n = \tilde{\mathcal{O}}(m)$$

□

6.3.1 Proof of Lemma* 6.3.6

Proof. Let $V = \{x_1, ..., x_n\}$. By Proposition 6.3.4 and Conjecture 6.3.1 we can see that

$$\mathbb{E}_{\mathcal{F}\in\mathscr{S}_F}[\Gamma(act(\mathcal{E}\mathcal{F}),\prec_{t-1})] = \mathbb{E}_{\mathcal{F}\in\mathscr{S}_F}[\Gamma(act(\mathcal{E}\mathcal{F}),\prec_{t-1}) - \Gamma(act(\mathcal{E}),\prec_{t-1})]$$

$$= \mathbb{E}_{\mathcal{F}\in\mathscr{S}_F}[\Gamma(act(\mathcal{E},\mathcal{F}),\prec_{t-1})] = \mathrm{polylog}\,n$$

Suppose the rest of the edges are inserted uniformly at random. Let $p(x_i)$ be the probability that x_i is relevant in $\prec_{t-1}$. We can see that

$$p(x_1) + ... + p(x_n) = \mathbb{E}_{\mathcal{F}\in\mathscr{S}_F}[\Gamma(act(\mathcal{E}\mathcal{F}),\prec_{t-1})] = \mathrm{polylog}\,n$$

Since $\mathcal{E}_t = (u, v)$ is equally likely to be any of the edges in F and each node has at most Δ outgoing edges, we get that

$$\mathbb{E}_{\mathcal{F}}[\lambda_t(\mathcal{E}\mathcal{F},\prec)] = p(x_1)\cdot\mathbb{P}[u = x_1] + ... + p(x_n)\cdot\mathbb{P}[u = x_n]$$

$$\leq p(x_1)\cdot\frac{\Delta}{m-t+1} + ... + p(x_n)\cdot\frac{\Delta}{m-t+1} = \frac{\Delta\,\mathrm{polylog}\,n}{m-t+1}$$

□

6.3.2 Proof of Lemma* 6.3.7

Outline

Suppose we have inserted the incomplete insertion sequence $\mathcal{E} = (\mathcal{E}_1, ..., \mathcal{E}_{t-1})$ into the graph and let $F = E \setminus \{\mathcal{E}_1, ..., \mathcal{E}_{t-1}\}$. In this context, given some $\mathcal{F} \in \mathscr{S}_F$ let $\overline{\mathcal{F}}$ denote $\mathcal{E}\mathcal{F}$ and assume that $t \leq m$. Let $S \subseteq \mathscr{S}_F$ be the set of incomplete insertion sequences $\mathcal{F}$ such that $\overline{\mathcal{F}}_t = (u, v)$ where u is relevant in $\prec_{t-1}$, in other words, such that $\lambda_t(\overline{\mathcal{F}},\prec_0) = 1$. We assume that $\Lambda_t > 0$. Let $f \in F$, then let $S_f \subseteq S$ be the set of all $\mathcal{F} \in S$ such that $\overline{\mathcal{F}}_t = f$. Note that S_f may be empty and that $\{S_f\}_{f\in F}$ is a partition of S. The idea of this proof is to define an equivalence relation, $\simeq$, on S_f (and hence on S) such that, given any $f \in F$, for any $\mathcal{F}^\star \in S_f$ we have

$$\mathbb{E}_{\mathcal{F}\in[\mathcal{F}^\star]_\simeq}[\Lambda_t(\overline{\mathcal{F}},\prec)] = \mathrm{polylog}\,n$$

Notice crucially that this expression requires $\prec_t$ to be defined, which is why we start by restricting to S_f. Once we have established this, we can construct a partition $P_1, ..., P_k$ of S such that for all $i \in [k]$ we have $\mathbb{E}_{\mathcal{F}\in P_i}[\Lambda_t(\overline{\mathcal{F}},\prec)] = \mathrm{polylog}\,n$ and it will follow that

$$\mathbb{E}_{\mathcal{F}\in\mathscr{S}_F}[\Lambda_t(\mathcal{E}\mathcal{F},\prec)|\lambda_t(\mathcal{E}\mathcal{F},\prec) = 1] = \mathbb{E}_{\mathcal{F}\in S}[\Lambda_t(\overline{\mathcal{F}},\prec)] = \mathrm{polylog}\,n$$

Proof

Proof. We start off by defining $\simeq$ and proving that it defines an equivalence relation. Fix some $f \in F$ and let $\prec_t \sim (y_1, ..., y_{l_1}, x_1, ..., x_k, y_{l_1+1}, ..., y_{l_2})$ where the startpoint of f is x_1 and $x_2, ..., x_k$ are the nodes that incur a recourse during the insertion of f. Note that this is the state of the ordering *after* the insertion if f.

Given some $\mathcal{F} \in S_f$, we define $\tau_1(\mathcal{F})$ to be the number $i \in [m]$ such that $\overline{\mathcal{F}}_i$ is the first insertion that causes a node in $\{x_1\}_{j=1}^{k}$ to be activated. Similarly, we define $\tau_2(\mathcal{F})$ to be the number $i \in [m]$ such that $\overline{\mathcal{F}}_i$ is the first insertion that causes a node in $\{y_j\}_{j=l_1+1}^{l_2}$ to be activated. Since x_1 is relevant in $\prec_{t-1}$, and if some x_j is active at time $s \geq t$ then x_1 is too, it follows that $\tau_1(\mathcal{F}) < \tau_2(\mathcal{F})$.

Definition 6.3.8. *Let $f \in F$ and $\mathcal{F}, \mathcal{F}^\star \in S_f$. We say that $\mathcal{F} \simeq \mathcal{F}^\star$ if the following hold.*

1. *$\tau_1(\mathcal{F}) = \tau_1(\mathcal{F}^\star)$ and $\tau_2(\mathcal{F}) = \tau_2(\mathcal{F}^\star)$*
2. *$\mathcal{F}^\star$ can be obtained by starting with $\mathcal{F}$ and permuting the positions of the edges in $\mathcal{F}$ that appear strictly between the edges $\overline{\mathcal{F}}_i$ and $\overline{\mathcal{F}}_j$ with $i = \tau_1(\mathcal{F})$ and $j = \tau_2(\mathcal{F})$.*

The fact that permuting edges as described doesn't change the value of $\tau_1(\mathcal{F})$ or $\tau_2(\mathcal{F})$ can be used to show that $\simeq$ is an equivalence relation on S_f. This can be proven using the *memoryless* property of activation sequences, in other words, the fact that the activation sequence after some point only depends on the order in which the remaining edges will be inserted and does not depend on the order in which the edges already in the graph were inserted.

Let $\mathcal{F} \in S_f$ and let $i = \tau_1(\mathcal{F})$ and $j = \tau_2(\mathcal{F})$. An important observation is that any node $y \in \{y_i\}_{i=1}^{l_2}$ is relevant in $\prec_{t-1}$ if and only it is relevant in $\prec_t$ with respect to $\overline{\mathcal{F}}$. This tells us that Λ_t is the difference in the number of nodes from $\{x_i\}$ relevant in $\prec_t$ and $\prec_{t-1}$. This difference can only be created by the edges contained strictly between $\overline{\mathcal{F}}_i$ and $\overline{\mathcal{F}}_j$. Let $\overline{\mathcal{F}}^1 = (\overline{\mathcal{F}}_1, ..., \overline{\mathcal{F}}_i)$, $\overline{\mathcal{F}}^2 = (\overline{\mathcal{F}}_{i+1}, ..., \overline{\mathcal{F}}_{j-1})$ and $\overline{\mathcal{F}}^3 = (\overline{\mathcal{F}}_j, ..., \overline{\mathcal{F}}_m)$. Then we have that

$$\begin{aligned}
\Lambda_t(\overline{\mathcal{F}}, \prec) &= \Gamma(act(\overline{\mathcal{F}}), \prec_t) - \Gamma(act(\overline{\mathcal{F}}), \prec_{t-1}) \\
&= \big(\Gamma(act(\overline{\mathcal{F}}^1), \prec_t) + \Gamma(act(\overline{\mathcal{F}}^1, \overline{\mathcal{F}}^2), \prec_t) + \Gamma(act(\overline{\mathcal{F}}^1\overline{\mathcal{F}}^2, \overline{\mathcal{F}}^3), \prec_t)\big) \\
&-\big(\Gamma(act(\overline{\mathcal{F}}^1), \prec_{t-1}) + \Gamma(act(\overline{\mathcal{F}}^1, \overline{\mathcal{F}}^2), \prec_{t-1}) + \Gamma(act(\overline{\mathcal{F}}^1\overline{\mathcal{F}}^2, \overline{\mathcal{F}}^3), \prec_{t-1})\big) \\
&= \Gamma(act(\overline{\mathcal{F}}^1, \overline{\mathcal{F}}^2), \prec_t) - \Gamma(act(\overline{\mathcal{F}}^1, \overline{\mathcal{F}}^2), \prec_{t-1}) \leq \Gamma(act(\overline{\mathcal{F}}^1, \overline{\mathcal{F}}^2), \prec_t)
\end{aligned}$$

Let $\mathcal{F}^\star \in S_f$ and let $P = [\mathcal{F}^\star]_\simeq$. Given any $\mathcal{F} \in P$, $\overline{\mathcal{F}}^1 = (\overline{\mathcal{F}^\star})^1$. So we get that $\mathbb{E}_{\mathcal{F} \in P}[\Gamma(act(\overline{\mathcal{F}}^1, \overline{\mathcal{F}}^2), \prec_t)] = \operatorname{polylog} n$ by Conjecture 6.3.1. This is because $\overline{\mathcal{F}}^1$ is a fixed incomplete insertion sequence for all $\mathcal{F} \in P$, and because the (multi) set of incomplete insertion sequences $\overline{\mathcal{F}}^2$ corresponding to $\mathcal{F} \in P$ is exactly all incomplete insertion sequences over the edges contained in $(\overline{\mathcal{F}^\star})^2$ (with each incomplete insertion sequence appearing the same number of times). It immediately follows that

$$\mathbb{E}_{\mathcal{F} \in P}[\Lambda_t(\overline{\mathcal{F}}, \prec)] \leq \mathbb{E}_{\mathcal{F} \in P}[\Gamma(act(\overline{\mathcal{F}}^1, \overline{\mathcal{F}}^2), \prec_t)] = \operatorname{polylog} n$$

□

Chapter 7

Conclusion

We now evaluate the contributions of this project and discuss possibilities for further work provided by this project.

7.1 Contribution

This project has contributed 4 new technical results and frameworks to the long line of research on the topic of incremental topological ordering and cycle detection.

The first result is a divide and conquer framework for the design of non-local incremental topological ordering algorithms. This framework is constructed from the definition of *semi-topological partitions* along with many propositions and lemmas which we introduce for the first time. This framework yields very different algorithms to what is currently in the literature on this topic and provides a new way of maintaining a topological ordering by using a dynamic *meta-tree* data structure.

Almost 10 years ago, it was proven by Haeupler et al. that no local algorithm for incremental topological ordering is capable of obtaining near-linear total recourse (and hence update time) under adversarial arrival. The second result of this project shows that this proof breaks down under random order arrival, showing that even though local algorithms are not suitable for dealing with the major and notoriously hard open problems in the area related to adversarial arrival, they are potentially candidates to solve other open problems in the area related to random order arrival.

Motivated by our second result, our third result is a framework for the design and analysis of the local *simple one-way search algorithm*. This framework is constructed from the definitions of *activation sequences* and a metric Γ defined on such sequences along with many propositions and lemmas which we introduce for the first time. The framework yields a lower bound on the recourse of this algorithm on any DAG G in terms of Γ, which we prove to be tight when G is a tree. Using this we then prove that there exists a randomized local algorithm that obtains $\mathcal{O}(n \log n)$ total recourse on trees with high probability under *adversarial arrival*, making progress towards solving an open problem in the area and obtaining a strong result for trees.

Our fourth result is a direct extension of the activation sequence framework. We generalize the lower bound on the recourse of the one-way search algorithm to obtain an equivalent characterization of the recourse of this algorithm on any DAG G in terms of the function Γ. Using this characterization, we give a reduction from an open problem to a statement arising naturally from this framework which we believe to be true. This reduction allows us to fix the

ordering of the graph and only concern ourselves with how the structure of the edges changes over time as insertions are made in a random order. The statement arising in the framework is also equivalent to an interesting variant of the open problem where we can start with an arbitrary DAG instead of having to start with an empty graph.

7.2 Future Work

The various contributions of this project can be extended directly in many different ways but may also be of independent interest to the reader.

Our first result, the divide and conquer framework, may be of interest to researchers looking to design new non-local algorithms or algorithms with near-linear recourse under random order arrival which admit efficient implementations, where recourse is close to update time. Using the definitions and propositions we created to construct the framework and formulating new results to build on them, it would be easy to make modifications to the meta-tree data structure and obtain new algorithms.

Our second result, Proposition 4.2.2, will be of interest to researchers attempting to prove that there exists an algorithm with near-linear total recourse (or update time) under random order arrival. This result shows that there is no evidence that local algorithms perform worse than non-local algorithms in this model, and hence their simplicity may make them desirable to study in this setting over non-local algorithms.

Our activation sequence framework is probably the most interesting result we give and has the most potential for future work, with Theorem 5.0.1 being an example of a concrete result obtained from this framework. This framework may be particularly interesting to researchers attempting to prove Conjecture 2.4.2 or Conjecture 2.4.1. The reduction we give from Conjecture 6.3.1 to Conjecture 2.4.4 may be directly useful since proving Conjecture 6.3.1 from our framework would lead to a significant result for sparse graph of bounded degree. The framework also contains many propositions about the properties of the one-way search algorithm that may be of independent interest and indirectly useful as well.

Bibliography

[AF10] Deepak Ajwani and Tobias Friedrich. "Average-case analysis of incremental topological ordering". *Discret. Appl. Math.*, 158(4):240–250, 2010.

[AFM06] Deepak Ajwani, Tobias Friedrich, and Ulrich Meyer. "An $\mathcal{O}(n^{2.75})$ Algorithm for Online Topological Ordering". In *SWAT 2006*, volume 4059 of *Lecture Notes in Computer Science*, pages 53–64. Springer, 2006.

[BC18] Aaron Bernstein and Shiri Chechik. "Incremental Topological Sort and Cycle Detection in Expected $\tilde{\mathcal{O}}(m\sqrt{n})$ Total Time". In *SODA 2018*, pages 21–34, 2018.

[BFG09] Michael A. Bender, Jeremy T. Fineman, and Seth Gilbert. "A new approach to incremental topological ordering". In *SODA 2009*, pages 1108–1115, 2009.

[BFGT16] 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, 2016.

[BK20] Sayan Bhattacharya and Janardhan Kulkarni. "An Improved Algorithm for Incremental Cycle Detection and Topological Ordering in Sparse Graphs". In *SODA 2020*, pages 2509–2521, 2020.

[CFKR13] Edith Cohen, Amos Fiat, Haim Kaplan, and Liam Roditty. "A Labeling Approach to Incremental Cycle Detection". *CoRR*, abs/1310.8381, 2013.

[GS20] Anupam Gupta and Sahil Singla. "Random-Order Models". *CoRR*, abs/2002.12159, 2020.

[HKM+12] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan. "Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance". *ACM Trans. Algorithms*, 8(1):3:1–3:33, 2012.

[MNR96] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. "Maintaining a Topological Order Under Edge Insertions". *Inf. Process. Lett.*, 59(1):53–58, 1996.

[PK06] David J. Pearce and Paul H. J. Kelly. "A dynamic topological sort algorithm for directed acyclic graphs11. *ACM J. Exp. Algorithmics*, 11, 2006.