Three-in-a-tree in near linear time

Lai, Kai Yuan; Lu, Hsueh I.; Thorup, Mikkel

Published in:
STOC 2020 - Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing

DOI:
10.1145/3357713.3384235

Publication date:
2020

Document version
Publisher's PDF, also known as Version of record

Document license:
Unspecified

Citation for published version (APA):
Three-in-a-Tree in Near Linear Time∗

Kai-Yuan Lai‡
baalidablo3@gmail.com
National Taiwan University
Taiwan

Hsueh-I Lu‡
hil@csie.ntu.edu.tw
National Taiwan University
Taiwan

Mikkel Thorup§
mikkel2thorup@gmail.com
University of Copenhagen
Denmark

ABSTRACT

The three-in-a-tree problem is to determine if a simple undirected graph contains an induced subgraph which is a tree connecting three given vertices. Based on a beautiful characterization that is proved in more than twenty pages, Chudnovsky and Seymour [Combinatorica 2010] gave the previously only known polynomial-time algorithm, running in $O(m^2 n^3)$ time, to solve the three-in-a-tree problem on an $n$-vertex $m$-edge graph. Their three-in-a-tree algorithm has become a critical subroutine in several state-of-the-art graph recognition and detection algorithms.

In this paper we solve the three-in-a-tree problem in $O(m \log^2 n)$ time, leading to improved algorithms for recognizing perfect graphs and detecting thetas, pyramids, beetsles, and odd and even holes. Our result is based on a new and more constructive characterization than that of Chudnovsky and Seymour. Our new characterization is stronger than the original, and our proof implies a new simpler proof for the original characterization. The improved characterization gains the first factor $n$ in speed. The remaining improvement is based on dynamic graph algorithms.

CCS CONCEPTS
• Mathematics of computing → Graph algorithms.

KEYWORDS
Induced subgraph detection, graph recognition, dynamic graph algorithms, top tree, SPQR-tree, perfect graph, odd hole, even hole

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires specific prior permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires specific prior permission and/or a fee. Request permissions from permissions@acm.org.

STOC ’20, June 22–26, 2020, Chicago, IL, USA © 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6979-4/20/06...$15.00
https://doi.org/10.1145/3357713.3384235

Figure 1: Comparing results for $n$-vertex graphs.

1 INTRODUCTION

The graphs considered in this paper are all assumed to be undirected. Also, it is convenient to think of them as connected. Let $G$ be such a graph with $n$ vertices and $m$ edges. An induced subgraph of $G$ is a subgraph $H$ that contains all edges from $G$ between vertices in $H$. For the three-in-a-tree problem, we are given three specific terminals in $G$, and we want to decide if $G$ has an induced tree $T$, that is, a tree $T$ which is an induced subgraph of $G$, containing these terminals. Chudnovsky and Seymour [18] gave the formerly only known polynomial-time algorithm, running in $O(m n^3)$ time, for the three-in-a-tree problem. In this paper, we reduce the complexity of three-in-a-tree from $O(m n^2)$ to $O(m \log^2 n) = \tilde{O}(m)$ time.

Theorem 1.1. It takes $O(m \log^2 n)$ time to solve the three-in-a-tree problem on an $n$-vertex $m$-edge simple graph.

To prove Theorem 1.1, we first improve the running time to $O(m n)$ using a simpler algorithm with a simpler correctness proof than that of Chudnovsky and Seymour. The remaining improvement is done employing dynamic graph algorithms.

1.1 Significance of Three-in-a-Tree

The three-in-a-tree problem may seem like a toy problem, but it has proven to be of general importance because many difficult graph detection and recognition problems reduce to it. The reductions are often highly non-trivial and one-to-many, solving three-in-a-tree on multiple graph instances with different placements of the three terminals. With our near-linear three-in-a-tree algorithm and some improved reductions, we get the results summarized Figure 1. These results will be explained in more detail in Section 1.2.

Showing some of the connections, our improved three-in-a-tree algorithm leads to an improved algorithm to detect if a graph has an odd hole, that is, an induced cycle of odd length above three. This is via the recent odd-hole algorithm of Chudnovsky, Scott, Seymour, and Spirkl [17]. A highly nontrivial consequence of odd-hole...
algorithm is that we can use it to recognize if a graph $G$ is perfect, that is, if the chromatic number of each induced subgraph $H$ of $G$ equals the clique number of $H$. The celebrated Strong Perfect Graph Theorem states that a graph is perfect if and only if neither the graph nor its complement has an odd hole. An odd-hole algorithm can therefore trivially test if a graph is perfect. The Strong Perfect Graph Theorem, implying the last reduction was a big challenge to mathematics, conjectured by Berge in 1960 [5–7] and proved by Chudnovsky, Robertson, Seymour, and Thomas [16], earned them the 2009 Fulkerson prize. Our improved three-in-a-tree algorithm improves the time to recognize if a graph is perfect from $O(n^5)$ to $O(n^3)$. While this is a modest polynomial improvement, the point is that three-in-a-tree is a central sub-problem on the path to solve many other problems.

The next obvious question is why three-in-a-tree? Couldn’t we have found a more general subproblem to reduce to? The dream would be to get something like disjoint paths and graph minor theory where we detect a constant sized minor or detect if we have disjoint paths connecting of a constant number of terminal pairs (one path connecting each pair) in $O(n^2)$ time. This is using the algorithm of Kawarabayashi, Kobayashi, and Reed [44], improving the original cubic algorithm of Robertson and Seymour [53].

In light of the above grand achievements, it may seem unambitious for Chudnovsky and Seymour to work on three-in-a-tree as a general tool. The difference is that the above disjoint paths and minors are not necessarily induced subgraphs. Working with induced paths, many of the most basic problems become NP-hard. Obviously, we can decide if there is an induced path between two terminals, but Bienstock [8] has proven that it is NP-hard to decide two-in-a-cycle, that is, if two terminals are in an induced cycle. From this we easily get that it is NP-hard to decide three-in-a-path, that is if there is an induced path containing three given terminals. Both of these problems would be trivial if we could solve the induced disjoint path problem for just two terminal pairs. In connection with the even and odd holes and perfect graphs, Bienstock also proved that it is NP-hard to decide if there is an even (respectively, odd) hole containing a given terminal.

In light of these NP-hardness results it appears quite lucky that three-in-a-tree is tractable, and of sufficient generality that it can be used as a base for solving other graph detection and recognition problems nestled between NP-hard problems. In fact, three-in-a-tree has become such a dominant tool in graph detection that authors sometimes explain when they think it cannot be used [19, 58]. Here we explain in more detail how our near-linear three-in-a-tree algorithm for solving detection problems is the algorithm three-in-a-tree of Kawarabayashi, Kobayashi, and Reed [44], improving the original cubic algorithm of Robertson and Seymour [53].

Theorem 1.1 reduces the time to $O(n^3)$. Moreover, we show in Lemma 6.1 that thetas in $G$ can be detected via solving the three-in-a-tree problem on $O(mn^2)$-vertex graphs, leading to an $O(n^6)$-time algorithm as stated in Theorem 1.2.

Theorem 1.2. It takes $O(mn^4 \log^2 n)$ time to detect thetas in an $n$-vertex $m$-edge graph.

1.2 Implications

We are now going to describe the use of our three-in-a-tree algorithm to improve the complexity of several graph detection and recognition problems. The reader less familiar with structural graph theory may find it interesting to see how the route to solve the big problems takes us through several toy-like subproblems, starting from three-in-a-tree. Often we look for some simple configuration implying an easy answer. If the simple configuration is not present, then this tells us something about the structure of the graph that we can try to exploit.

We first define the big problems in context. A hole is an induced simple cycle with four or more vertices. A graph is chordal if and only if it has no hole. Rose, Tarjan, and Leuker [54] gave a linear-time algorithm for recognizing chordal graphs. A hole is odd (respectively, even) if it consists of an odd (respectively, even) number of vertices. $G$ is Berge if and only if $G$ is hole-free. The celebrated Strong Perfect Graph Theorem, which was conjectured by Berge [5–7] and proved by Chudnovsky, Robertson, Seymour, and Thomas [16], states that $G$ is Berge if and only if $G$ is perfect, i.e., the chromatic number of each induced subgraph $H$ of $G$ equals the clique number of $H$.

The big problems considered here are detecting odd and even holes, but related to this we are going to look for “thetas”, “pyramids”, and “beetles”. These are different induced subdivisions where a subdivision of a graph is one where edges are replaced by paths of arbitrary length. A hole is thus an induced subdivision of a length-4 cycle, and a minimal three-in-a-tree is an induced subdivision of a star with 2 or 3 leaves that are all prespecified terminals.

The first problem Chudnovsky and Seymour [18] solved using their three-in-a-tree algorithm was to detect a theta which is a different induced subdivision of $K_{3,3}$ [4]. Chudnovsky and Seymour are interested in thetas because they trivially imply an even hole. They developed the previously only known polynomial-time algorithm, running in $O(n^{11})$ time, for detecting thetas in $G$ via solving the three-in-a-tree problem on $O(n^9)$ subgraphs of $G$. Thus, Theorem 1.1 reduces the time to $O(n^3)$. Moreover, we show in Lemma 6.1 that thetas in $G$ can be detected via solving the three-in-a-tree problem on $O(mn^2)$-vertex graphs, leading to an $O(n^6)$-time algorithm as stated in Theorem 1.2.

Theorem 1.2. It takes $O(mn^4 \log^2 n)$ time to detect thetas in an $n$-vertex $m$-edge graph.

The next problem Chudnovsky and Seymour solved using their three-in-a-tree algorithm was to detect a pyramid which is an induced subgraph consisting of an apex vertex $u$ and a triangle $v_1v_2v_3$ and three paths $P_1$, $P_2$, and $P_3$ such that $P_1$ connects $u$ to $v_1$ and touches $P_j$, $j \neq i$, only in $u$, and such that at most one of $P_1$, $P_2$, and $P_3$ has only one edge. The point in a pyramid is that it must contain an odd hole. An $O(n^6)$-time algorithm for detecting pyramids was already contained in the perfect graph algorithm of Chudnovsky et al. [13, §2], but Chudnovsky and Seymour use their three-in-a-tree to give a more natural “less miraculous” algorithm for pyramid detection, but with a slower running time of $O(n^{10})$. With our faster three-in-a-tree algorithm, their more natural pyramid detection also becomes the faster algorithm with a running time of $O(n^6)$. Moreover, as for thetas, we improve the reductions to three-in-a-tree. We show (see Lemma 6.2) that pyramids in $G$ can be detected via solving the
three-in-a-tree problem on $O(mn)$ $n$-vertex graphs, leading to an $O(mn^3)$-time algorithm as stated in Theorem 1.3.

**Theorem 1.3.** It takes $O(m^3n \log^2 n)$ time to detect pyramids in an $n$-vertex $m$-edge graph.

We now turn to odd holes and perfect graphs. Since a graph is perfect if and only if it and its complement are both odd-hole-free, an odd-hole algorithm implies a perfect graph algorithm, but not vice versa. Cornuénjols, Liu, and Vušković [28] gave a decomposition-based algorithm for recognizing perfect graphs that runs in $O(n^{18})$ time, reduced to $O(n^{15})$ time by Charbit, Habib, Trotignon, and Vušković [12]. The best previously known algorithm, due to Chudnovsky, Cornuénjols, Liu, Seymour, and Vušković [13], runs in $O(n^8)$ time. However, the tractability of detecting odd holes was open for decades [20, 22, 26, 42] until recently. Chudnovsky, Scott, Seymour, and Spirkl [17] announced an $O(n^8)$-time algorithm for detecting odd holes, which also implies a simpler $O(n^8)$-time algorithm for recognizing perfect graphs. An $O(n^8)$-time bottleneck of both of these perfect-graph recognition algorithms was the above-mentioned algorithm for detecting pyramids [13, §2].

By Theorem 1.3, the pyramids can now be detected in $O(mn^3)$-time, but Chudnovsky et al.’s odd-hole algorithm has six more $O(n^8)$-time subroutines [17, §4]. By improving all these bottleneck subroutines, we improve the detection time for odd holes to $O(m^2n^4)$, hence the recognition time for perfect graphs to $O(n^8)$.

**Theorem 1.4.** (1) It takes $O(m^2n^4)$ time to detect odd holes in an $n$-vertex $m$-edge graph, and hence (2) it takes $O(n^8)$ time to recognize an $n$-vertex perfect graph.

Even-hole-free graphs have been extensively studied [1, 23, 24, 29, 30, 37, 45, 55]. Vušković [62] gave a comprehensive survey. Conforti, Cornuénjols, Kapoor, and Vušković [21, 25] gave the first polynomial-time algorithm for detecting even holes, running in $O(n^4)$ time. Chudnovsky, Kawarabayashi, and Seymour [15] reduced the time to $O(n^{11})$. A prism consists of two vertex-disjoint triangles together with three vertex-disjoint paths between the two triangles such that the union of every two of the three paths induces a cycle. Chudnovsky et al. [15] also observed that the time of detecting even holes can be further reduced to $O(n^{15})$ as long as detecting prisms is not too expensive, but this turned out to be NP-hard [51]. However, Chudnovsky and Kapadia [14] and Maffray and Trotignon [51, Algorithm 2] devised $O(n^{35})$-time and $O(n^{26})$-time algorithms for detecting prisms in theta-free and pyramid-free graphs $G$, respectively. Later, da Silva and Vušković [30] improved the time of detecting even holes in $O(n^{19})$. The best formerly known algorithm, due to Chang and Lu [11], runs in $O(n^{11})$ time. One of its two $O(n^{11})$-time bottlenecks [11, Lemma 2.3] detects so-called beetles in $G$ via solving the three-in-a-tree problem on $O(n^7)$ subgraphs of $G$. Theorem 1.1 reduces the time to $O(n^9)$. Moreover, we show in Lemma 6.3 that beetles can be detected via solving the three-in-a-tree problem on $O(m^2) n$-vertex graphs, leading to an $O(n^9)$-time algorithm as stated in Theorem 1.5.

**Theorem 1.5.** It takes $O(m^2n^2 \log^2 n)$ time to detect beetles in an $n$-vertex $m$-edge graph.

Combining our faster beetle-detection algorithm with our $O(n^9)$-time algorithm in §6.4, which is carefully improved from the other $O(n^{11})$-time bottleneck subroutine [11, Lemma 2.4], we reduce the time of detecting even holes to $O(n^9)$ as stated in Theorem 1.6.

**Theorem 1.6.** It takes $O(m^3n^5)$ time to detect even holes in an $n$-vertex $m$-edge graph.

For other implications of Theorem 1.1, Lévêque, Lin, Maffray, and Trotignon gave $O(n^{11})$-time and $O(n^{14})$-time algorithms for certain properties $Π_{R_k}$ and $Π_{B_k}$, respectively [49, Theorems 3.1 and 3.2]. By Theorem 1.1 and the techniques of §6, the time can be reduced by a $Θ(n^{5}/\log^2 n)$ factor. Theorem 1.1 also improves the algorithms of van ’t Hof, Kaminski, and Paulusma [60, Lemmas 4 and 5]. We hope and expect that three-in-a-tree with its new near-optimal efficiency will find many other applications in efficient graph algorithms.

### 1.3 Other Related Work

For the general $k$-in-a-tree problem, we are given $k$ specific terminals in $G$, and we want to decide if $G$ has an induced tree $T$. The $k$-in-a-tree problem is NP-complete [32] when $k$ is not fixed. With our Theorem 1.1, it can be solved in near-linear time for $k \leq 3$, and the tractability is unknown for any fixed $k \geq 4$ [38]. Solving it in polynomial time for constant $k$ would be a huge result. It is, however, not clear that $k$-in-a-tree for $k > 3$ would be as powerful a tool in solving other problems as three-in-a-tree has proven to be.

While $k$-in-a-tree with bounded $k$ is unsolved for general graphs, there has been substantial work devoted to $k$-in-a-tree for special graph classes. Derhy, Picouleau, and Trotignon [33] and Liu and Trotignon [50] studied $k$-in-a-tree on graphs with girth at least $k$ for $k \geq 4$ and general $k \geq 4$, respectively. dos Santos, da Silva, and Szwarcfiter [35] studied the $k$-in-a-tree problem on chordal graphs. Golovach, Paulusma, and van Leeuwen [38] studied the $k$-in-a-tree, $k$-in-a-cycle, and $k$-in-a-path problems on AT-free graphs [48]. Bruhn and Saito [10], Fiala, Kaminski, Lidický, and Paulusma [36], and Golovach, Paulusma, and van Leeuwen [39] studied the $k$-in-a-tree and $k$-in-a-path problems on claw-free graphs.

On the hardness side, recall that three-in-a-tree can also be viewed as three in a subdivided star with two or three terminal leaves. However, detecting such a star with 4 terminal leaves is NP-hard. (This follows from Bienstock’s NP-hardness of 2-in-a-cycle [8], asking if there exists a hole containing two vertices $u$ and $v$, which may be assumed to be nonadjacent: Add two new leaves $u_1$ and $u_2$ adjacent to $u$ and then, for every two neighbors $v_1$ and $v_2$ of $v$, check if the new graph contains an induced subdivision of a star with exactly four terminal leaves $u_1, u_2, v_1, v_2$.) Even without terminals, it is NP-hard to detect induced subdivisions of any graph with minimum degree at least four [3, 49]. Finally, we note that if we allow multigraphs with parallel edges, then even 2-in-a-path becomes NP-hard. This NP-hardness is an easy exercise since the induced path cannot contain both end-points of parallel edges.

We note that it is the subdivisions that make induced graph detection hard for constant sized pattern graphs. Without subdivisions, we can trivially check for any induced $k$-vertex graph in $O(n^k)$ time. Nesetril and Poljak has improved this to roughly $O(n^{k_0}/3^k)$ where $\omega$ is the exponent of matrix multiplication [52]. On the other hand, the ETH hypothesis implies that we cannot detect if a $k$-clique is an induced subgraph in $n^{o(k)}$ time [43]. See [31] for a more general understanding of the hardness of detecting induced graphs.
1.4 Techniques

Chudnovsky and Seymour’s $O(n^2 m)$-time algorithm for the three-in-a-tree problem is based on their beautiful characterization for when a graph with three given terminals are contained in some induced tree [18]. The aim is to either find a three-in-a-tree or a witness that it cannot exist. During the course of the algorithm, they develop the witness to cover more and more of the graph. In each iteration, they take some part that is not covered by the current witness and try to add it in, but then some other part of the witness may pop out. They then need a potential function argument to show progress in each iteration.

What we do is introduce some extra structure to the witness when no three-in-a-tree is found, so that when things are added, nothing pops out. This leads to a simpler more constructive algorithm that is faster by a factor $n$. Our new witness has more properties than that of Chudnovsky and Seymour, so our characterization of no three-in-a-tree is strictly stronger, yet our overall proof is shorter. Essentially the point is that by strengthening the inductive hypothesis, we get a simpler inductive step. The remaining improvement in speed is based on dynamic graph algorithms.

1.5 Road Map

The rest of the paper is organized as follows. Section 2 is a background section where we review Chudnovsky and Seymour’s characterization for three-in-a-tree, sketch how it is used algorithmically, as well as the bottleneck for a fast implementation. Section 3 presents our new stronger characterization as well as a high level description of the algorithms and proofs leading to our $O(m)$ implementation. Section 4 proves the correctness of our new characterization. Section 5 provides an efficient implementation. Section 6 shows how our improved three-in-a-tree algorithm, in tandem with other new ideas, is used to improve many state-of-the-art graph recognition and detection algorithms. Section 7 concludes the paper. See a full version at https://arxiv.org/abs/1909.07446 for the proofs omitted due to the page limit of STOC 2020.

2 BACKGROUND

Let $|S|$ denote the cardinality of set $S$. Let $R \setminus S$ for sets $R$ and $S$ consist of the elements of $R$ not in $S$. Let $G$ and $H$ be graphs. Let $V(G)$ (respectively, $E(G)$) consist of the vertices (respectively, edges) of $G$. Let $u$ and $v$ be vertices. Let $U$ and $V$ be vertex sets. Let $N_G(u)$ consist of the neighbors of $u$ in $G$. The degree of $u$ in $G$ is $|N_G(u)|$. Let $N_G[u] = N_G(u) \cup \{u\}$. Let $N_G(U)$ be the union of $N_G(u) \setminus U$ over all vertices $u \in U$. Let $N_G(u, H) = N_G(u) \cap V(H)$ and $N_G(U, H) = N_G(U) \cap V(H)$. The subscript $G$ in notation $N_G$ may be omitted. A leaf of $G$ is a degree-one vertex of $G$. Let $\nabla(G)$ denote the graph obtained from $G$ by adding an edge between each pair of leaves of $G$. Let $G[H]$ denote the subgraph of $G$ induced by $V(H)$. Let $G - U = G[V(G) \setminus U]$. Let $G - u = G - \{u\}$. Let $uv$ denote an edge with end-vertices $u$ and $v$. Graphs $H_1$ and $H_2$ are disjoint if $V(H_1) \cap V(H_2) = \emptyset$. Graphs $H_1$ and $H_2$ are adjacent in $G$ if $H_1$ and $H_2$ are disjoint and there is an edge $uv$ of $G$ with $u \in V(H_1)$ and $v \in V(H_2)$. A $UV$-path is either a vertex in $U \cap V$ or a path having one end-vertex in $U$ and the other end-vertex in $V$. A $UV$-rung [18] is a vertex-minimal induced $UV$-path. If $U = \{u\}$, then a $UV$-path is also called a $uV$-path and a $Vu$-path. If $U = \{u\}$ and $V = \{v\}$, then a $UV$-path is also called a $uv$-path. Let $Uv$-rung, $vU$-rung, and $uv$-rung be defined similarly.

For the three-in-a-tree problem, we assume without loss of generality that the three given terminals of the input $n$-vertex $m$-edge simple undirected graph $G$ are exactly the leaves of $G$. A sapling of $G$ is an induced tree containing all three leaves of $G$, so the three-in-a-tree problem is the problem of finding a sapling.

2.1 Chudnovsky & Seymour’s Characterization

Let $H$ be a graph such that each member of $V(H)$ and $E(H)$, called node and arc respectively, is a subset of $X \subseteq V(G)$. $H$ is an $X$-net if the following Conditions N hold (see Figure 2(a)): $N1$: Graph $H$ is connected and graph $\nabla(H)$ is biconnected. $N2$: The arcs of $H$ form a nonempty disjoint partition of the vertex set $X$. $N3$: Graph $H$ has exactly three leaf nodes, each of which consists of a leaf vertex of $G$. $N4$: For any arc $E = UV$ of $H$, each vertex of $X$ in $E$ is on a $UV$-rung of $G[E]$. $N5$: For any arc $E$ and node $V$ of $H$, $E \cap V \neq \emptyset$ if and only if $V$ is an end-node of $E$ in $H$. $N6$: For any vertex $u$ and $v$ in $X$ contained by distinct arcs $E$ and $F$ of $H$, $uv$ is an edge of $G$ if and only if arcs $E$ and $F$ share a common end-node $V$ in $H$ with $\{u, v\} \subseteq V$. An arc $E = UV$ is simple if $G[E]$ is a $UV$-rung. A net is an $X$-net for an $X$. A base net is a net obtained via the next lemma.

Lemma 2.1 (Chudnovsky and Seymour [18]). It takes $O(m)$ time to find a sapling of $G$ or a net of $G$ whose arcs are all simple.

The original definition of Chudnovsky and Seymour only used nets with no parallel arcs, but for our own more efficient construction, we need to use parallel arcs. A triad of $H$ is $\Delta(V_1, V_2, V_3) = (V_1 \cap V_2) \cup (V_2 \cap V_3) \cup (V_3 \cap V_1)$ for nodes $V_1, V_2, V_3$ that induce a triangle in graph $H$. A subset $S$ of $X$ is $H$-local if $S$ is contained by a node, arc, or triad of $H$ [18]. A set $Y \subseteq V(G - X)$ is $H$-local if $N(Y, X)$ is $H$-local. $H$ is local if every $Y \subseteq V(G - X)$ with contained $G[Y]$ is $H$-local. See Figure 2. The following theorem is Chudnovsky and Seymour’s characterization.

Theorem 2.2 (Chudnovsky and Seymour. [18, 3.2]). $G$ is sapling free if and only if $G$ admits a local net with no parallel arcs.

The proof of Theorem 2.2 in [18] takes up more than 20 pages. We will here present a stronger characterization with a shorter proof, which moreover leads to a much faster implementation. Our results throughout the paper do not rely on Theorem 2.2. Moreover, our paper delivers an alternative self-contained proof for Theorem 2.2.
Figure 3: $\nabla(\mathcal{H})$ for a web $\mathcal{H}$. The yellow arcs are in $\nabla(\mathcal{H}) \setminus \mathcal{H}$.

Chudnovsky and Seymour’s proof of Theorem 2.2 is algorithmic maintaining an $X$-net $\mathcal{H}$ with $X \subseteq V(G)$ having no parallel arcs until a sapling of $G$ is found or $\mathcal{H}$ becomes local, implying that $G$ is sapling-free by the if direction of Theorem 2.2. In each iteration, if $\mathcal{H}$ is not local, they find a minimal set $Y \subseteq V(G - X)$ with connected $G[Y]$ such that $Y$ is $\mathcal{H}$-nonlocal. Their proof for the only-if direction of Theorem 2.2 shows that if $G[X \cup Y]$ is sapling-free, then $\mathcal{H}$ can be updated to an $X'$-net with $Y \subseteq X' \subset X \cup Y$. Although $Y$ joins the resulting $X'$-net $\mathcal{H}$, a subset of $X$ may have to be moved out of $\mathcal{H}$ to preserve Conditions N for $\mathcal{H}$. To bound the number of iterations, Chudnovsky and Seymour showed that the potential $|Y| + (n + 1) \cdot |V(\mathcal{H})|$ of $\mathcal{H}$ stays $O(n^2)$ and is increased by each iteration, implying that the total number of iterations is $O(n^2)$. In the next section, we will present a new stronger characterization that using parallel arcs with particular properties avoids the aforementioned in-and-out situation. More precisely, our $X$ will grow in each iteration, reducing the number of iterations to at most $n$.

3 OUR STRONGER CHARACTERIZATION

A base net of $G$ contains only simple arcs. However, we do need other more complex arcs, but we will see that it suffices that all non-simple arcs are “flexible” in the sense defined below. For vertex sets $S, U_1, V_2$ an $(S, V_1, V_2)$-sprout is an induced subgraph of $G$ in one of the following Types $S$: $S1$: A tree intersecting each of $S, V_1,$ and $V_2$ at exactly one vertex. $S2$: An $SV_1$-rung not intersecting $V_2$ plus a disjoint $SV_2$-rung not intersecting $V_1$. $S3$: A $V_1V_2$-rung not intersecting $S$ plus a disjoint $SV$-rung with $V = V_1 \cup V_2$. Let $S = \{1, \ldots, 7\}$ for the example in Figure 3. Vertex 1 is an $(S, V_1, V_2)$-sprout of Type $S1$. The set $\{2, 19, 12, 11, 13, 14, 15, 16\}$ induces an $(S, V_1, U_2)$-sprout of Type $S1$. The only $(S, U_1, U_2)$-sprout and $(S, W_1, W_2)$-sprout of Type $S1$ contain vertex 1. The set $\{23, 4, 7, 28\}$ induces an $(S, W_1, W_2)$-sprout of Type $S2$. The set $\{19, 2, 13, 14, 15, 16\}$ induces an $(S, U_1, U_2)$-sprout of Type $S3$. An arc $E = UV$ of $H$ is flexible if $G[E]$ contains an $(S, U, V)$-sprout for each nonempty vertex set $S \subseteq E$. For the example in Figure 3, arcs $E_1, E_3, E_4, E_5, E_6$ are simple and arcs $E_1, E_2, E_7$ are flexible. An $X$-net $\mathcal{H}$ is an $X$-web if all arcs of $\mathcal{H}$ are simple or flexible. A web is an $X$-web for some $X$. A base net of $G$ is a web of $G$. Let $\mathcal{H}$ be a net. A split component $\mathcal{G}$ for $\mathcal{H}$ is either an arc $UV$ of $\mathcal{H}$ or a subgraph of $\mathcal{H}$ containing a cutset $\{U, V\}$ of $\nabla(\mathcal{H})$ such that $\mathcal{G}$ is a maximal subgraph of $\nabla(\mathcal{H})$ in which $U$ and $V$ are nonadjacent and do not form a cutset [34]. For both cases, we call $\{U, V\}$ the split pair of $G$ for $\mathcal{H}$. The split components having split pair $\{V_1, V_2\}$ in Figure 3 are (1) the $V_1V_2$-path with an arc $E_1$, (2) the $V_1V_2$-path with arcs $E_3, E_2, E_4$, and (3) the $V_1V_2$-path with arcs $E_5, E_7, E_6$. Thus, even if $\mathcal{H}$ has no parallel arcs, there can be more than one split components sharing a common split pair. One can verify that each split component $G$ of $\mathcal{H}$ contains at most one leaf node of $\mathcal{H}$ and, if $G$ contains a leaf node $V$ of $\mathcal{H}$, then $V$ belongs to the split pair of $G$. A vertex subset $C$ of $G$ is a chunk of $\mathcal{H}$ if $C$ is the union of the arcs of one or more split components for $\mathcal{H}$ that share a common split pair $\{U, V\}$ for $\mathcal{H}$. In this case, we call $\{U, V\}$ the split pair of $C$ for $\mathcal{H}$ and call $C$ a UV-chunk of $\mathcal{H}$. A chunk of $\mathcal{H}$ is maximal if it is not properly contained by any chunk of $\mathcal{H}$. A node of $\mathcal{H}$ is a maximal split node if it belongs to the split pair of a maximal chunk for $\mathcal{H}$. For the net $\mathcal{H}$ of $G$ in Figure 3, $E_1, E_3, E_5 \cup E_2, E_3 \cup E_2 \cup E_4,$ and $E_1 \cup E_3 \cup E_2 \cup E_4$ are all chunks of $\mathcal{H}$. If we consider only the subsets of $V(G)$ that intersect the numbered vertices, then $E_1 \cup E_3 \cup E_7$ is the only maximal chunk and $V_1$ and $V_2$ are the only maximal split nodes. Given an $X$-net $\mathcal{H}$, a subset $S$ of $X$ is $\mathcal{H}$-tamed if every pair of vertices from $S$ is either in the same arc or together in some node of $\mathcal{H}$. A set $Y \subseteq V(G - X)$ is $\mathcal{H}$-tamed if $N(Y, X)$ is $\mathcal{H}$-tamed. $\mathcal{H}$ is taming if every $Y \subseteq V(G - X)$ with connected $G[Y]$ is $\mathcal{H}$-tamed. If $S \subseteq X$ is $\mathcal{H}$-local, then $S$ is $\mathcal{H}$-tamed. The converse does not hold: If $\mathcal{H}$ has simple arcs $E$ and $F$ between nodes $U$ and $V$, $G[E]$ is an edge $uv$ with $u \in U$ and $v \in V$, and $G[F]$ is a vertex $w \in U \cap V$, then $\{u, w, v\}$ is $\mathcal{H}$-tamed and $\mathcal{H}$-nonlocal. However, if $\mathcal{H}$ has no parallel arcs, then each $\mathcal{H}$-tamed subset of $X$ is $\mathcal{H}$-local, as shown in Lemma 3.5(2).

A non-trivial $V_1V_2$-chunk $C$ of $\mathcal{H}$ is one that is not an arc in $\mathcal{H}$. We then define the operation merge($C$) which for a $V_1V_2$-chunk $C$ of $\mathcal{H}$ replaces all arcs of $\mathcal{H}$ intersecting $C$ by an arc $E = V_1V_2$ with $E = C$ and deletes the nodes whose incident arcs are all deleted. We shall prove that this merge operation preserves that $\mathcal{H}$ is a net (see Lemma 3.4). Let $\mathcal{H}^I$ denote the $X$-net obtained from $\mathcal{H}$ by applying merge($C$) on $\mathcal{H}$ for each maximal chunk $C$ of $\mathcal{H}$. We call $\mathcal{H}^I$ the $X$-net that aids $\mathcal{H}$. Such an aiding net has no non-trivial chunks and no parallel arcs. See Figure 4 for examples. The simple graph $\nabla(\mathcal{H}^I)$ is triconnected, $V$ is node of $\mathcal{H}^I$ if and only if $V$ is a maximal split node of $\mathcal{H}$. $E$ is an arc of $\mathcal{H}^I$ if and only if $E$ is a maximal chunk of $\mathcal{H}$ (respectively, $\mathcal{H}^I$). The next theorem is our characterization, which is the basis for our much more efficient near-linear time algorithm.

Theorem 3.1. $G$ is sapling-free if and only if $G$ admits a web $\mathcal{H}$ with a taming aiding net $\mathcal{H}^I$. 

Figure 4: The aiding net of the $\mathcal{H}$ in (a) is the $\mathcal{H}^I$ in (b) with $E^I_1 = E_2 \cup \cdots \cup E_8$ and $E^I_2 = E_9 \cup \cdots \cup E_{15}$. The net $\mathcal{H}$ in (c) aids itself.
Theorem 3.1 is stronger than Chudnovsky and Seymour’s Theorem 2.2 in that our proof of Theorem 3.1 provides as a new shorter proof of Theorem 2.2. For the relation between the two structural theorems, we will prove in Lemma 3.5(2) that every taming net of \(G\) having no parallel arcs is local. Since the aiding net \(H^\dagger\) in Theorem 3.1 has no parallel arcs, \(H^\dagger\) is local as required by Theorem 2.2. The algorithmic advantage of Theorem 3.1 is that \(H^\dagger\) is the aiding net of a web \(H\) which has more structure than an arbitrary net.

To get a self-contained proof of the easy if-direction of Theorem 3.1, we prove more generally that if \(G\) admits a taming net, then \(G\) is sapling-free (Lemma 3.5(1)). This proof holds for any net including nets with parallel arcs like our web \(H\). Proving the only-if direction is the hard part for both structural theorems. Our new proof follows the same general pattern as the old one stated after the statement of Theorem 2.2, but with crucial differences to be detailed later.

We grow an \(X\)-web \(H\) with \(X \subseteq V(G)\) until a sapling of \(G\) is found or \(H^\dagger\) becomes taming, implying \(G\) is sapling-free by the if-direction of Theorem 3.1. In each iteration, if \(H^\dagger\) is not taming, we find a minimal set \(Y \subseteq V(G-X)\) with connected \(G[Y]\) such that \(Y\) is not \(H^\dagger\)-tamed. To prove the only-if direction of Theorem 3.1, we show that if \(G[X \cup Y]\) is sapling-free, then \(H\) can be expanded to an \(X^\dagger\)-web with \(X^\dagger = X \cup Y\).

Comparing with the proof of Chudnovsky and Seymour that we sketched below Theorem 2.2, we note that in their case, their new \(X^\dagger\)-net would be for some \(Y \subseteq X^\dagger \subseteq X \cup Y\), whereas we get \(X^\dagger = X \cup Y\). This is why we can guarantee termination in \(O(n)\) rounds while they need a more complicated potential function to demonstrate enough progress in \(O(n^2)\) rounds.

Another major difference is that we operate both on a web \(H\) and its aiding net \(H^\dagger\). Recall that the web \(H\) is a net allowing parallel arcs, but with the special structure that all arcs are simple or flexible. This special structure is crucial to our simpler inductive step where we can always add \(Y\) as above to get a new web over \(X^* = X \cup Y\). If we just used \(H\), then we would have too many untamed sets. This is where we use the aiding net \(H^\dagger\) which generally has fewer untamed sets. It is only for the minimally \(H^\dagger\)-untamed sets \(Y \subseteq V(G-X)\) that we can guarantee progress as above. Thus we need the interplay between the well-structured fine grained web \(H\) and its more coarse grained aiding net \(H^\dagger\) to get our shorter more constructive proof of Theorem 3.1. On its own, our more constructive characterization buys us a factor \(n\) in speed. This has to be combined with efficient data structures to get down to near-linear time.

3.1 Two Lemmas & Our Algorithm for Sapling

Let \(H\) be an \(X\)-net. An \(H\)-wild set is a minimally \(H\)-untamed \(Y \subseteq V(G-X)\) such that \(G[Y]\) is a path. In Figure 5, \(Y_1 \cup Y_2\) is \(H\)-untamed but not \(H\)-wild, since \(Y_1 \subseteq Y_1 \cup Y_2\) is \(H\)-untamed, \(H\) is not taming if and only if \(G\) admits an \(H\)-wild set. An \(S \subseteq X\) is \(H\)-solid if \(S\) is a node of \(H\) or \(S\) is a subset of an arc \(e = UV\) of \(H\) such that \(G[E] \) contains no \((S, U, V)\)-spout. If \(S\) is a subset of a simple arc of \(H\), then \(S\) is \(H\)-solid if and only if \(G[S]\) is an edge, since a spout has to be an induced subgraph of \(G\). Let \(Y \subseteq V(G-X)\) such that \(G[Y]\) is a path, \(Y\) is \(H\)-solid if (1) \(N(y, X)\) is the union of two \(H\)-solid sets and (2) \(N(y, X)\) is \(\emptyset\) for each internal vertex \(y\), if any, of path \(G[Y]\). A pod of \(Y\) in \(H\) is a \(V_1V_2\)-chunk of \(H\) with the following conditions.

Figure 5: An \(X\)-web \(H\), where \(X\) consists of the vertices other than \(y_1, y_2, y_3, y_4\). Vertices \(y_1, \ldots, y_4\) are all \(H\)-tamed and \(H^\dagger\)-tamed. \(y_1\) and \(y_2\) are \(H\)-wild and \(H^\dagger\)-nonwild. \(y_3\) and \(y_4\) are both \(H\)-tamed and \(H^\dagger\)-wild. \(y_5\) is \(H\)-unpadded. \(y_1\) and \(y_2\) are \(H\)-sticky and \(y_3\) and \(y_4\) are \(H\)-nonsticky.

3.2 Reductions to Lemmas 3.2 and 3.3

This subsection reduces Theorems 1.1, 2.2, and 3.1 to Lemmas 3.2 and 3.3. We need a relationship between simple paths in \(H\) and induced paths in \(G\). For any simple \(UV\)-path \(P\) of \(H\) (i.e., \(U\) and \(V\) are the end-nodes of \(P\) in \(H\)), we define a \(P\)-rung of \(G\) as a UV-rung of \(G\) where all edges are contained in the arcs of \(P\). Such a \(P\)-rung always exists by Conditions N4 and N6 of \(H\) as long as \(U \neq V\). For the degenerate case \(U = V\), let \(P\)-rung be defined as the empty vertex set. For any distinct nodes \(U_1\) and \(U_2\) of \(H\) intersecting a \(V_1V_2\)-chunk \(C\) of \(H\), there are disjoint \(UV\)-rungs \(P_1\) and \(P_2\) of \(H\) with \(U = \{U_1, U_2\}\) and \(V = \{V_1, V_2\}\) by Condition N1 of \(H\). Since \(P_1\) and \(P_2\) are disjoint, any \(P_1\)-rung and \(P_2\)-rung of \(G\) are disjoint and nonadjacent by Conditions N2 and N6 of \(H\). Consider the \(V_1V_2\)-chunk \(C = E_1 \cup \cdots \cup E_3\) in Figure 3. Let \(V = \{V_1, V_2\}\). Let \(P_1\) be the path of \(H\) with arc \(E_1\). Let \(P_2\) be the path of \(H\) with arc \(E_2\). Let \(P_3\) be the path of \(H\) with arcs \(E_4\) and \(E_5\). Let \(P_4\) be the degenerate path of \(H\) consisting of a single node \(V_1\). If \(U = \{U_1, U_2\}\), then
If \( Y \) is \( H \)-wild for an \( X \)-net \( H \), then let \((Y, H, G)\) denote the minimum number of \( H \)-tamed subsets of \( X \) whose union is \( N_G(Y, X) \). A net is simple if all of its arcs are simple. If \( H \) is a simple self-aiding \( X \)-net of \( G \), then \( G[X] \) is isomorphic to the line graph of a subdivision of \( H \).

**Lemma 4.4.** If \( Y \) is an \( H \)-wild set for a simple self-aiding \( X \)-net \( H \) of \( G \) with \((Y, H, G) = 2 \) such that \( N_G(Y, X) \) contains a triad of \( H \), then \( G[X \cup Y] \) contains a sapling.

**Lemma 4.5.** Let \( Y \) be an \( H \)-wild set for a simple self-aiding \( X \)-net \( H \) of graph \( G \) with \((Y, H, G) \geq 3 \). If \( G[X \cup Y] \) is sapling-free, then \( Y \) is \( H \)-podded for \( G \).

**Proof of Lemma 4.2.** Assume for contradiction that \( G[X \cup Y] \) is sapling-free. A vertex set \( X \subseteq D \) is an inducing set of \( E_H \) if \( G[E_0 \cap D] \) for each arc \( E_0 = U_0V_0 \) is an \( U_0V_0 \)-run of \( G[E_0] \). For any inducing set \( D \) of \( E_H \), let \( H(\bar{D}) \) denote the simple self-aiding \( D \)-net of graph \( H(D) = G[Y \cup D] \) obtained from \( H \) by replacing each arc \( E_0 \) of \( H(\bar{D}) \) with the arc \( E = E_0 \cap D \) and replacing each node \( V_0 \) of \( H \) with the node \( V = V_0 \cap D \). Let \( N = N_G(Y, X) \). By \((Y, H, G) \geq 3 \), then Lemma 4.3 implies \( N \not\subseteq S_1 \cup S_2 \) for any node or arc \( S_i \) of \( H \). For each \( i \in \{1, 2\} \), let \( S_i \) contain a triangle \( \Delta \) and \( \Delta \not\subseteq N \). By Lemma 3.1, \( \Delta \) is not contained by any arc of \( H \). By Lemma 3.5, \( \Delta \) is contained by a node or arc of \( H \). Hence, \( \Delta \) is a \( H \)-podded for \( G \). Therefore, \( \Delta \) is a \( H \)-podded for \( G \).
(a) If \( V_2 \) is incident to exactly one arc \( F = VV_2 \) in the arc set for \( C \), \( N \cap V_2 \subseteq F \), and \( F \) is simple, then \( N \) intersects \( F \setminus V \) by the minimality of \( C \). Let \( v_2 \) be the end-vertex of the \( NV_2 \)-rung \( P \) of \( G[F] \) in \( N \). Let \( v \) be the other end vertex of \( v_2 \) in \( P \). Call \( \text{SUBDIVIDE}(v, v_2) \) to create a node \( V_2 = \{v, v_2\} \). Delete \( V(P) \) from \( C \) to preserve that \( C \) is a \( V_1V_2 \)-chunk that is a minimal pod of \( F \) in \( H \).

(b) Update \( H \) by \( \text{merge}(C) \). Let \( E = V_1V_2 \) be the arc of \( H \) with \( E = C \).

(c) Add \( Y \) to arc \( E \) and add each end-vertex \( y \) of path \( G[Y] \) to the nodes \( V_1 \) with \( V_1 \subseteq C \cup N(y) \).

The resulting \( H \) of Step B1 is an \( X \cup Y \)-web, since all steps preserve Conditions \( N \) and all new arcs are simple. It remains to prove that the resulting \( H \) of Step B2 is also an \( X \cup Y \)-web.

This completes the proof of our characterization in Theorem 3.1 as well as Chudnovsky and Seymour’s characterization in Theorem 2.2. Subroutine B can be implemented to run in \( O(m) \) time, so Steps A2(c) and A2(d) take \( O(m) \) time. Steps A1, A2(a), and A2(b) take \( O(m) \) time. Since the set of vertices of \( G \) in \( H \) is enlarged by Step A2(d) and not affected elsewhere, Step A2 halts in \( O(n) \) iterations. Thus, Algorithm A can be implemented to run in \( O(\alpha(n)m) \) time. To finish proving Theorem 1.1, it remains to implement Algorithm A to run in \( O(m \log^2 n) \) time in §3 via dynamic graph algorithms and other data structures.

5 PROVING LEMMA 3.3

Let \( G \) be represented by a static adjacency list. We use a dynamic adjacency list to represent an incremental biconnected multigraph \( H^* \) with \( V(\tilde{H}^*) = V(H) \) that is a supergraph of \( V(H) \). An arc or node of \( H^* \) is dummy if it is an empty vertex set of \( G \). Thus, the arcs of \( V(H) \) between the leaves of \( H \) are dummy in \( H^* \). Other dummy nodes and arcs are created only via operation \( \text{MERGE} \). The \( X \)-web \( H \) maintained by Algorithm A is exactly \( H^* \) excluding its dummy arcs and nodes. See Figure 7(a) for an example of \( H^* \). Each node and arc of \( H \) and \( H^* \) is associated with a distinguished color that is a positive integer such that two vertices share a common arc color (resp., node color) for \( H \) and \( H^* \) if and only if they are contained by a common arc (resp., node) of \( H \) and \( H^* \). For each vertex \( v \) of \( G \), we maintain a set of at most six colors indicating the arc, maximal chunk, nodes, and maximal split nodes of \( H \) that contain \( v \), which are called the \( H \)-arc, \( H^* \)-arc, \( H \)-node, and \( H^* \)-node colors of vertex \( v \). For each color \( c \), we store its corresponding arc or node for \( H \) or \( H^* \) and maintain the number of the vertices having the color \( c \) without keeping an explicit list of these vertices. For each node \( v \) and each incident arc \( E \) of \( v \) in \( H \), we maintain the cardinality of the vertex set \( E \cap V \). Thus, it takes \( O(1) \) time to (1) update and query the colors of a vertex and (2) add a vertex to an arc or node of \( H \). For each arc of \( H^* \), mark whether it is dummy, simple, or flexible. For each simple arc \( E = V_1V_2 \) of \( H^* \), use a doubly linked list to store the \( V_1V_2 \)-rung \( G[E] \). For vertex \( v \) and vertex set \( Y \) of \( G \), let \( d(v) = |N(v)| \) and \( d(Y) = \sum_{y \in Y} d(y) \) throughout the section. Based on Lemma 5.1, to be proved in §5.4, Steps A2(a) and A2(b) are implemented in §5.1 to run in overall \( O(m \log^2 n) \) time. Step A2(c) is implemented in §5.2 to run in overall \( O(m) \) time. Step A2(d), i.e., Subroutine B is implemented in §5.3 to run in overall \( O(m \log n \cdot \alpha(n, n)) \) time, where \( \alpha(n, n) \) is the inverse Ackermann function.

5.1 Steps A2(a) and A2(b)

Vertex colors change only in Step A2(d), but the overall number of changes of the \( H^* \)-arc and \( H^* \)-node colors affects the analysis of our implementation of Steps A2(a) and A2(b). Therefore, this subsection analyzes the time for the change of \( H^* \)-arc and \( H^* \)-node colors. The time for the change of \( H \)-arc and \( H \)-node colors will be analyzed for Step A2(d) in §5.3. A vertex of \( G \) stays uncolored until it is added into \( X \). Each vertex of \( X \) has exactly one \( H^* \)-arc color and at most two \( H^* \)-node colors. Each node \( V \) of \( H^* \) stays a node of \( H^* \) and each vertex in \( V \) stays in \( V \) for the rest the algorithm. Thus, the
Three-in-a-Tree in Near Linear Time

STOC ’20, June 22–26, 2020, Chicago, IL, USA

\[ H^I \] -node colors of each vertex are updated \( O(1) \) times throughout the algorithm, implying that the overall time for updating \( H^I \)-node colors of all vertices is \( O(n) \). Although the \( H^I \)-arc color of a vertex may change many times, the overall time for updating the \( H^I \)-node colors of all vertices can be bounded by \( O(n \log n) \). Observe that \( H \) is updated by Subroutine B only via (1) subdividing a simple arc of \( H \), (2) merging an \( H \)-padded \( Y \) into a minimal pod of \( Y \) in \( H \), and (3) creating an arc \( E = Y \) for an \( H \)-solid \( Y \). If the simple graph \( H^I \) does not change, then each of these updates takes \( O(d(Y)) \) time. If the simple graph \( H^I \) changes, then \( Y \) is \( H \)-solid. For instance, let \( H \) be as in Figure 4(a), implying that \( H^I \) is as in Figure 4(b). If an \( H \)-solid \( Y \) joins \( H \) as the arc \( E_{16} \) in Figure 4(c), then all nodes and arcs of \( H \) become nodes and arcs of \( H^I \). However, once two vertices of \( X \) have distinct \( H^I \)-arc colors, they can no longer share a common arc color for \( H^I \) for the rest of the algorithm. Thus, one can bound the overall number of changes of \( H^I \)-arc colors of all vertices by \( O(n \log n) \): If \( E \) is an arc of the original \( H^I \) and \( E_1, \ldots, E_k \) are the arcs of the updated \( H^I \) with \( E_1 \cup \cdots \cup E_k \subseteq E \) and \( |E_{1}| \leq \cdots \leq |E_{k}| \), then let the vertices in \( E_k \) keep their original \( H^I \)-arc color and assign a new \( H^I \)-arc color to the vertices in each \( E_i \) with \( i \in \{1, \ldots, k-1\} \). Since the cardinality of the arc of \( H^I \) containing a specific vertex of \( X \) is halved each time its \( H^I \)-arc color changes, its \( H^I \)-arc colors have distinct \( H^I \)-node colors corresponding to a subset of the two end-vertices of \( a \), then there is an \( O(m \log n) \)-time obtainable data structure supporting the following queries and updates: (1) Move a vertex \( v \) of \( G - X \) to \( X \) in amortized \( O(d(v) \cdot \log^2 n) \) time. (2) Update the colors of a vertex \( v \in X \) in amortized \( O(d(v) \cdot \log n) \) time. (3) Determine if there is a set \( Y \subseteq V(G - X) \) with \( G \cdot Y \) such that two vertices of \( N(Y, X) \) share no color, and, for the positive case, report a minimal such \( Y \) in amortized \( O(d(Y) \cdot \log^2 n) \) time.

5.2 Step A2(c)

Let \( S \) be the \( O(d(Y)) \)-time obtainable set consisting of the nodes \( V \) of \( H \) with \( V \subseteq N(Y, X) \) and the simple arcs \( E \) of \( H \) with \( G[E \cap N(Y, X)] \) being an edge. \( Y \) is \( H \)-solid if and only if \(|S| = 2, N(y, X) = \emptyset \) for each internal node \( y \) of path \( G \cdot Y \), and \( N(Y, X) \) is contained by the union of the nodes or arcs in \( S \). Therefore, it takes \( O(d(Y)) \) time to determine whether \( Y \) is \( H \)-solid. Lemma 4.1(1) implies that \( Y \) is \( H \)-padded if and only if both of the following conditions hold: (a) \( N(Y, X) \) is contained by the union of an arc \( E \) of \( H^I \) and its end-nodes \( V_1 \) and \( V_2 \) in \( H^I \) and (b) \( E \) is a pod of \( Y \) in \( H^I \). Both conditions can be checked in \( O(d(Y)) \) time via the \( H^I \)-arc and \( H^I \)-node colors of each vertex in \( N(Y, X) \) and \( |V_2 \setminus E| \) and \( |V_1 \setminus E| \). Hence, it takes \( O(d(Y)) \) time to determine whether \( Y \) is \( H \)-padded. Since the \( H^I \)-wild sets \( Y \) in all iterations of the algorithm are pairwise disjoint, it takes overall \( O(m) \) time for Step A2(c) to determine whether \( Y \) is \( H \)-sticky throughout the algorithm.

5.3 Step A2(d), i.e., Subroutine B

This subsection implements Subroutine B so that the overall time of Step A2(d) throughout Algorithm A is \( O(m \log n \cdot o(n, n)) \). We may delete nodes and arcs from \( H \) via \( \text{merge}(C) \) for a minimal pod \( C \) of \( Y \) in \( H \), but they stay as dummy nodes and arcs in \( H^I \) in order to make the multigraph \( H^I \) incremental. One can verify that \( H^I \) aids \( H^I \), even though \( H^I \) is not an \( X \)-net due to its dummy arcs and nodes. Although Step B1(b) may change \( H^I \), the overall time for updating the \( H^I \)-colors has been accounted for in §5.1. This subsection only analyzes the time for changing \( H \)-arc and \( H^I \)-node colors and \( H^I \)-node colors of \( E \) with \( |E \cdot V_1| \) and \( |E \cdot V_2| \) for each arc \( E = V_1V_2 \) of \( H \).

The SPQR-tree \( T \) of the incremental multigraph \( H^I \) is an \( O(n) \)-time obtainable \( O(n) \)-space tree structure representing the triconnected components of \( H^I \) [34, 40]. Each member of \( V(T) \), which we call a knot, is a graph homeomorphic to a subgraph of \( H^I \) [34, Lemma 3] such that the knots induce a disjoint partition of the arcs of \( H^I \). Specifically, there is a supergraph \( G \) of \( H^I \) with \( V(G) = V(H^I) \), where each arc of \( G \) is called virtual [59], and there are four types of knots of \( T \): (1) S-knot: a simple cycle on three or more nodes. (2) P-knot: three or more parallel arcs. (3) Q-knot: two parallel arcs, exactly one of which is virtual. (4) R-knot: a triconnected simple graph that is not a cycle. The Q-knots are the leaves of \( T \) and each arc of \( H^I \) is contained by a Q-knot. No two S-knots (respectively, P-knots) are adjacent in \( T \). Each virtual arc is contained by exactly two adjacent knots. Since \( H \) has \( O(n) \) arcs by Condition N2, \( T \) has \( O(n) \) knots. If \( U \) and \( V \) are nonleaf nodes of \( H \) such that \( UV \) is a virtual arc, then \( \{U, V\} \) is a split pair of \( H \). If distinct nodes \( U \) and \( V \) admit three internally disjoint \( U \)-paths in \( H^I \), then \( U \) and \( V \) are contained by a common P-knot or R-knot of \( T \) [34]. By Condition N1 of \( H \), there are 3 internally disjoint paths in \( V(H) \) between each pair of leaves of \( H^I \), implying an R-knot of \( T \) containing the leaves of \( H \). Let \( T \) be rooted at this unique R-knot. Figure 7(b) is the \( T \) for the \( H^I \) in Figure 7(a). Let \( K \) be a nonroot knot of \( T \). The poles [40] of \( K \) are the end-nodes of the unique virtual arc contained by \( K \) and its parent knot in \( T \). For the four nonroot knots \( K \) in Figure 8, \( V_1 \) and \( V_4 \) (respectively, \( V_2 \)) are the poles of the knots in (a) and (d) (respectively, (b) and (c)). Let \( C(K) \) of \( K \) be the set of poles of \( H \) in the descendant Q-knots of \( K \) in \( T \). Let \( C(K) \) of \( K \) consist of the vertices of \( G \) contained by the arcs of \( C(K) \). If \( U \) and \( V \) are the poles of a nonroot knot \( K \) of \( T \), then \( C(K) \) is a UV-chunk and \( C(K) \) is the arc set for \( C(K) \). A nonempty vertex set \( C \) is a maximal chunk of \( H \) if and only if \( C = C(K) \) holds for a child knot \( K \) of the root of \( T \). For instance, the \( X \)-net \( H \) in Figure 7(a) has six maximal chunks. One of them is \( C(K) \) for the child R-knot.
Let $\alpha(n, n)$ time: (1) Add a node $V$ to subdivide an arc $V_1V_2$ of $\hat{H}$ into arcs $V_1 = VV_1$ and $V_2 = VV_2$. (2) Add an arc $UV$ between nodes $U$ and $V$ of $H$.

We first show that, given a vertex set $S$ contained by a simple arc $E = V_1V_2$ such that $G[S]$ is an edge, Operation $\text{subdivide}(S)$ in Steps B1(a) and B2(a) can be implemented to run in amortized $O(\log n)$ time: Let each $P_j$ with $j \in [1, 2]$ be the $V_j$-s-run of $G[E]$. Let $j$ be an index in $[1, 2]$ with $|V(P_j)| \leq |V(P_{j-1})|$. Using the doubly linked list for the $V_1V_2$-run $G[E]$, it takes $O(|V(P_j)|)$ time to (1) create a new node $V = S$ with a new $\hat{H}$-node color assigned to both vertices in $S$, (2) create a new simple arc $E_1 = VV_1$ consisting of the vertices of $P_j$, (3) assign a new $\hat{H}$-arc color for each vertex in $E_j$, (4) let arc $E_{2-j}$ take over the $\hat{H}$-arc color of $E$, and (5) obtain the doubly linked lists of $G[E_1]$ and $G[E_2]$ from that of $G[E]$. Each time a vertex $x$ is recolored this way, the cardinality of the simple arc of $H$ containing $x$ is halved. Therefore, the overall time for Operation $\text{subdivide}(S)$ in Steps B1(a) and B2(a) is $O(n \log n)$.

Step B1: By the above analysis for $\text{subdivide}$, Step B1(a) runs in amortized $O(\log n)$ time. As for Steps B1(b) and B1(c), a new $\hat{H}$-arc color is created for the new arc of $H$. The $\hat{H}$-arc and $\hat{H}$-node colors of the vertices in $Y$ and the cardinality of each vertex set that is a node, arc, or the intersection of a node and its incident arc can be updated in $O(d(Y))$ time. By Lemma 5.2 and that Subroutine B is executed $O(n)$ times, the overall time for Step B1 is $O(m \log n)$.

Step B2: We first assume that we are given a set $C$ of arcs of $H$ whose union is a minimal pod $C$ of $Y$ in $H$ and show how to implement Steps B2(a), B2(b), and B2(c) to run in overall $O(m \log n)$ time throughout Algorithm A. Let $C = V_1V_2$-chunk of $H$.

Step B2(a): It takes $O(|C|)$ time to determine whether $V_2$ is incident to exactly one arc $F = VV_2$ in $C$ and $F$ is simple. We start from $V$ to traverse the $V_1V_2$-run $G[F]$ to obtain the node $v_2 \in N(Y, F)$ that is closest to $V_2$ in $G[F]$. The required time is linear in the number of traversed edges plus $d(Y)$. Observe that Step B2(a) in any remaining iteration of Algorithm A does not traverse these edges again. Moreover, the sum of $|C|$ over all iterations of Algorithm A is $O(n)$. Thus, the overall time of Step B2(a) including that of calling $\text{subdivide}([v_1, v_2])$ is $O(m \log n)$.

Step B2(b): Let $E_1, \ldots, E_k$ with $|E_1| \leq \cdots \leq |E_k|$ be the arcs of $H$ in $C$. We implement $\text{merge}(C)$ in Step B2(b) to run in amortized $O(\log n)$ time: Create an arc $E = V_1V_2$ in $\hat{H}$ consisting of all vertices in $C$ and mark the original arcs $E_1, \ldots, E_k$ of $\hat{H}$ intersecting $C$ dummy so that $H'$ is incremental as required by Lemma 5.2. The nodes of $H'$ whose incident arcs are all dummy are also marked dummy. The cardinalities of $E, V_1, V_2, E \cap V_1,$ and $E \cap V_2$ can be obtained in $O(k)$ time. Since we do not keep an explicit list of the vertices in $C$, we simply let all vertices in $C$ adopt the $H'$-color of the vertices in $E_k$. Each time a vertex $x$ is recolored this way, the cardinality of the arc of $H$ containing $x$ is doubled. Once a vertex in $X$ loses its $H'$-node colors, it stays without any $H'$-node color for the rest of the algorithm. Combining with Lemma 5.2(2), Step B2(b) takes overall $O(n \log n)$ time throughout Algorithm A.

Step B2(c): The $\hat{H}$-arc and $\hat{H}$-node colors of the vertices of $Y$ and $E[V_1]$ and $E[V_2]$ can be updated in $O(d(Y))$ time.

Lemma 5.3 (Alstrup, Holm, Lichtenberg, and Thorup [2, §3.3]). For any dynamic rooted $n$-knot tree, there is an $O(n)$-time obtainable data structure supporting the following operations and queries on $T$ in amortized $O(\log n)$ time for any given distinct knots $K_1$ and $K_2$ of $T$: (1) If $K_2$ is not a descendant of $K_1$, then make the subtree rooted at $K_1$ a subtree of $K_2$ such that $K_2$ becomes the parent of $K_1$. (2) Obtain the lowest common ancestor of $K_1$ and $K_2$. (3) If $K_2$ is a descendant of $K_1$, then obtain the child knot of $K_1$ that is an ancestor of $K_2$ in $T$.

It remains to show that it takes overall $O(m \log n \cdot \alpha(n, n))$ time to obtain the arc set $C$ of a minimal pod of $C$ of an $H$-podded $Y$ in all iterations of Algorithm A. We additionally construct a data structure for $T$ ensured by Lemma 5.3. By Lemmas 5.2 and 5.3(1), the overall time for updating the data structure reflecting the updates to $T$ throughout Algorithm A is $O(n \log n \cdot \alpha(n, n))$. Let $C' = V_1V_2$ be the arc of $H'$ with $V_1 = V_1 \subseteq N(Y, V_1) \subseteq C$. By Conditions P, $C$ has to contain all arcs $E$ of $H$ with $(1) (E \cap V_1) \cap N(Y, X) \neq \emptyset$ or $(2) (E \cap V_1) \cap N(Y, X) \neq \emptyset$. Let $C_1$ and $C_2$ consist of the arcs of Types (1) and (2), respectively. It takes $O(d(Y))$ time to obtain $C_1$ and the incident arcs of $V_1$ that are not of Type (1) or (2). Then it takes $O(|C_2|)$ time to obtain $C_2$. By Lemma 5.3(2), it takes $O(|C_1| \cup |C_2| \cdot \log n)$ time to obtain the lowest knot $K$ of $T$ with $C_1 \cup C_2 \subseteq C(K)$. Since all arcs in $C_1 \cup C_2$ are merged into a single arc of $H$ via $\text{merge}(C)$ at the end of the current iteration, the overall time for obtaining $K$ throughout Algorithm A is $O(m \log n \cdot \alpha(n, n))$. It remains to show that $C$ can be obtained from $K$ in overall $O(m \log n \cdot \alpha(n, n))$ time throughout Algorithm A.

Case 1: $K$ is an $S$-knot. Let $V_1V_2 \cdots V_TV_1$ with $t \geq 3$ be the cycle of $K$ such that $V_1$ and $V_T$ are the poles of $K$. For each $i \in \{1, \ldots, t-1\}$, let $K_i$ be the child knot of $K$ with poles $V_i$ and $V_{i+1}$, $C_i = C(K_1) \cup \cdots \cup C(K_t)$, and let $C_i$ be the union of the arcs in $C_i$. Let $j$ be the smallest index in $\{2, \ldots, t-1\}$ with $C_i \cup C_j \subseteq C_j$. If $N(Y, X) \setminus (V_j \cup C_{j-1}) = V_j \setminus C_{j-1}$, then $C = C_{j-1}$; otherwise, $C = C_j$. For the example in Figure 8(a), if $N(X, Y) \setminus (V_1 \cup C_1) = V_2 \setminus E_1$, then $E_1$ is a minimal pod of $Y$ in $H$; otherwise, $E_1 \cup E_2$ is a minimal pod of $Y$ in $H$. By Lemma 5.3(3), the time required to obtain the index $j$ and determine whether $C = C_{j-1}$ or $C = C_j$ is dominated by the time of obtaining $K$ plus the time of $\text{merge}(C)$.

Case 2: $K$ is a $P$-knot. $C$ equals the union of $C(K')$ over all child knots $K$ of $K$ in $T$ with $(C_1 \cup C_2) \cap C(K') \neq \emptyset$. For the example in Figure 8(b), $E_1 \cup E_2$ is a minimal pod of $Y$ in $C$. By Lemma 5.3(3), the time needed to obtain $C$ is dominated by that of obtaining $K$.

Case 3: $K$ is a $Q$-knot. As illustrated by Figure 8(c), $C = C(K)$ can be obtained in $O(1)$ time.

Case 4: $K$ is an R-knot. If there is child knot $K'$ of $K$ in $T$ with poles $V_2$ and $V_3$ such that all arcs of $K$ intersecting $C_1 \cup C_2$ are incident to $V_2$ and $N(Y, X) \setminus (V_1 \cup C(K')) = V_2 \setminus C(K')$, then
Three-in-a-Tree in Near Linear Time

\( C = C(K') \); otherwise, \( C = C(K) \). For the example in Figure 8(d), if \( (N(Y, X) \setminus (V_1 \cup E_1)) = V_2 \setminus E_1 \), then \( E_1 \) is a minimal pod of \( Y \) in \( H \); otherwise, \( E_1 \cup \cdots \cup E_5 \) is a minimal pod of \( Y \) in \( H \). By Lemma 5.3(3), the time required to identify all possible vertices \( V_2 \), which can be at most two, is dominated by the time of identifying \( K \). If there are no possible \( V_2 \), then we have \( C = C(K) \). Otherwise, for each of the at most two vertices \( V_2 \), we spend \( O(d(Y)) \) time to determine whether the child knot \( K' \) with poles \( V_1 \) and \( V_2 \) satisfies \( N(Y, X) \setminus (V_1 \cup C(K')) = V_2 \setminus C(K') \). For the positive (respectively, negative) case, we have \( C = C(K') \) (respectively, \( C = C(K) \)).

Therefore, the overall time for obtaining the arc set of a minimal pod of \( Y \) in \( H \) is \( O(m \log n \cdot d(n, n)) \). To complete our proof of Lemma 3.3, it remains to prove Lemma 5.1 in Section 5.4.

### 5.4 Proving Lemma 5.1

The subsection omits \( H^I \) from the terms \( H^I \)-wild, \( H^I \)-tamed, \( H^I \)-untamed, and \( H^I \)-node and \( H^I \)-arc colors. Recall that each vertex \( x \) of \( X \) is associated with exactly one arc color and at most two node colors from which we know which arc of \( E^I \) contains \( x \) and whether \( x \in E \cap V \) holds for each end-node \( V \) of \( E \). For any nonempty \( S \subseteq X \), we say that an \( R \subseteq S \) represents \( S \) and call \( R \) a representative set of \( S \) if \(|R| \leq 3 \) and, for any \( V \subseteq X \), \( R \cup V \) is tamed if and only if \( S \cup V \) is tamed. If \( S \) is untamed, then each untamed two-vertex subset of \( S \) represents \( S \). If \( R_1 \) represents \( S_1 \), \( R_2 \) represents \( S_2 \), and \( R \) represents \( R_1 \cup R_2 \), then \( R \) represents \( S_1 \cup S_2 \).

**Lemma 5.4.** Any nonempty \( S \subseteq X \) admits a representative set obtainable from the colors of the vertices of \( S \) in \( O(|S|) \) time.

For each \( y \in V(G - X) \), we maintain a balanced binary search tree \( T_y \) on \( N(y, X) \). For each vertex \( x \) of \( T_y \), we maintain a representative set \( R_y(x) \) of the vertices in the subtree of \( T_y \) rooted at \( x \). Thus, \( R_y = R_y(\text{root}(T_y)) \) represents \( N(y, X) \). We also maintain a doubly linked list \( D_1 \) for the vertices \( y \in V(G - X) \) with untamed \( N(y, X) \). When a vertex joins \( N(y, X) \) or a vertex in \( N(y, X) \) changes color, \( R_y \) and \( D_1 \) can be updated in \( O(|S|) \) time by Lemma 5.4. Thus, as long as \( D_1 \neq \emptyset \), \( H^I \) is not taming and an \( H^I \)-wild set consisting of a single vertex can be obtained from \( D_1 \) in \( O(1) \) time, implying Lemma 5.1. The rest of the subsection handles the case \( D_1 = \emptyset \).

**Lemma 5.5 (Holm, de Lichtenberg, Thorup [41]).** A spanning forest of an \( n \)-vertex dynamic graph can be maintained in amortized \( O(n \log n) \) time per edge insertion and deletion such that each update to the graph adds and deletes at most one edge in the spanning forest.

We maintain a spanning forest \( F \) of the decremental graph \( G - X \) by Lemma 5.5. For each maximal connected \( U \subseteq V(F) \), we maintain a balanced binary search tree \( T_U \) on \( U \). For each \( y \in U \), we maintain a representative set \( R_U(y) \) of the vertices in the subtree of \( T_U \) rooted at \( y \). It takes \( O(1) \) time to determine if \( U \) is tamed from \( R_U = R_U(\text{root}(T_U)) \). We also maintain a doubly linked list \( D_2 \) for the untamed maximal connected subsets \( U \) of \( V(F) \). When \( R_U \) for a vertex \( y \in V(G - X) \) changes, \( D_2 \) and \( R_U \) for the maximal connected \( U \subseteq V(F) \) containing \( y \) can be updated in \( O(n \log n) \) time by Lemma 5.4. If deleting an edge of \( F \) decomposes a maximal connected \( U \subseteq V(F) \) into \( U_1 \) and \( U_2 \) with \(|U_1| \geq |U_2| \), it takes \( O(|U_1| \log n) \) time to delete the vertices of \( U_1 \) from \( T_{U_1} \), construct \( T_{U_2} \), and obtain \( R_{U_1} \). The resulting \( T_{U_2} \) and \( R_{U_2} \) become \( T_{U_2} \) and \( R_{U_2} \), respectively, \( D_2 \) can be updated in \( O(1) \) time. Whenever a vertex

\[ y \text{ moves to a new connected component, the number of vertices of the connected component containing } y \text{ is halved. Hence, the } T_U \text{ for all maximal connected sets } U \subseteq V(F) \text{ are changed overall } O(n \log n) \text{ times. Thus, the overall time to maintain } D_2 \text{ and all representative sets } R_U \text{ throughout the algorithm is } O(n \log n) \text{, not affecting the correctness of Lemmas 5.1(1) and 5.1(2) and the first half of Lemma 5.1(3). It remains to prove the second half of Lemma 5.1(3) for the case } D_1 = \emptyset \text{ and } D_2 = \emptyset, \text{ i.e., each } N(y, X) \text{ with } y \in V(G - X) \text{ is tamed and } H^I \text{ is not taming.}

A top tree is defined over a dynamic tree \( T \) and a dynamic set \( d T \) of at most two vertices of \( T \). For any subtree \( C \) of \( T \), \( d C = d T \cap d T \cap C \) consists of the vertices of \( C \) belonging to \( d T \) or adjacent to \( V(T) \setminus V(C) \). A cluster [2] of \( (T, d T) \) is a subcluster of \( T \) with \( |E(C)| \geq 1 \) and \( |d C| \leq 2 \). If \( |d C| = 2 \), then \( \Pi(C) \) be the path of \( T \) between the vertices of \( d C \). If \( E(T) = \emptyset \), then \( (T, d T) \) admits no cluster and the top tree over \( (T, d T) \) is empty. If \( E(T) \geq 1 \), then a top tree \( T' \) over \( (T, d T) \) is a binary tree on clusters of \( (T, d T) \) such that (1) the root of \( T' \) is the maximal cluster \( T \) of \( (T, d T) \), (2) the leaves of \( T' \) are the edges of \( T \), i.e., the minimal clusters of \( (T, d T) \), and (3) the children \( A \) and \( B \) of any cluster \( C \) of \( (T, d T) \) are edge-disjoint clusters of \( (T, d T) \) with \( C = A \cup B \) and \( V(A) \cap V(B) = \emptyset \).

Figure 9: The cases of joining the child clusters \( A \) and \( B \) with \( |d A| \geq |d B| \) into their parent cluster \( C = A \cup B \) on a top tree. Row 1 shows the three cases with \( |d A| = |d B| \). Row 2 shows the two cases with \( |d A| > |d B| \). The vertex in \( A \cap B \) is purple. The vertices in \( d C \) are black. If \( |d C| = 2 \), then the black line is \( \Pi(C) \). If \( |d C| = 2 \), then the red line is \( \Pi(A) \). If \( |d B| = 2 \), then the yellow line is \( \Pi(B) \).

\[ \text{Figure 9: The cases of joining the child clusters } A \text{ and } B \text{ with } |d A| \geq |d B| \text{ into their parent cluster } C = A \cup B \text{ on a top tree. Row 1 shows the three cases with } |d A| = |d B|. Row 2 shows the two cases with } |d A| > |d B|. \text{ The vertex in } A \cap B \text{ is purple. The vertices in } d C \text{ are black. If } |d C| = 2, \text{ then the black line is } \Pi(C). \text{ If } |d C| = 2, \text{ then the red line is } \Pi(A). \text{ If } |d B| = 2, \text{ then the yellow line is } \Pi(B). \]
O(log n)-height top trees such that for any maximal subtree T of F (1) it takes O(1) time to obtain on the top tree T′ for T (a) the cluster C_v for any v ∈ V(T) \ dT, (b) the parent of a nonroot cluster, (c) the children of a non-leaf cluster, and (d) dC for a cluster C and (2) it takes O(log n) time to identify a sequence of O(log n) top-tree operations with which T′ can be modified in O(log n) time with respect to (a) updating dT, (b) deleting an edge of T, or (c) adding an edge between T and another maximal subtree of F.

We use Lemma 5.6 to maintain a top forest F over the spanning forest F of G – X maintained by Lemma 5.5. For each cluster C on each nonempty top tree T of F, we maintain a representative set R_C of N(V(C) \ dC, X). We first show that maintaining the representative sets R_C does not affect the complexity of maintaining F stated in Lemma 5.6 and that of maintaining the colors of the vertices of X stated in Lemmas 5.1(1) and 5.1(2). By Lemma 5.4, the following bottom-up update for a cluster B on a top tree T of F takes O(log n) time: For each cluster C on the BT-path of B from T to B′, if C is an edge of C, then R_C can be obtained from R_{C′} ∪ R_{C}, in O(1) time; if C is not an edge of C, then R_C can be obtained from R_{C′} ∪ R_{C} in O(1) time, where C_1 and C_2 are the children of C on T and e is the vertex in V(C_1) ∩ V(C_2). Thus, the initial R_C for all clusters C of all top trees T of F can be obtained in overall O(m log n) time by performing a bottom-up update for each leaf cluster of each top tree. With respect to each top-tree operation, the representative sets R_C can be updated in O(1) time: For destroy and split, we simply delete R_C together with the root C of T_C. For create and merge, we just perform a bottom-up update for C in O(1) time. Therefore, maintaining the representative sets R_C does not affect the complexity of maintaining F stated in Lemma 5.6. If a vertex v ∈ V(G – X) moves to X or the colors of a vertex v ∈ X change, we update R_C for all O(d(v) log n) clusters C with v ∈ N(V(C) \ dC, X), specifically, for each of the O(d(v)) vertices y ∈ V(G – X) with v ∈ N(y, X), we perform a bottom-up update for C_y in O(log n) time. Thus, maintaining the representative sets R_C does not affect the correctness of Lemmas 5.1(1) and 5.1(2). We omit the proof of Lemma 5.1(3) for the case with D_1 = ∅ and D_2 ≠ ∅.

6 IMPROVED RECOGNITION & DETECTION

6.1 Theta, Pyramid, and Beetle

Each previous algorithm for detecting a family F of graphs in G via three-in-a-tree identifies a set G of a polynomial number of subgraphs H of G, each associated with a set L(H) of three terminals, such that G is F-free if and only if each graph H in G does not admit an induced tree containing L(H). In addition to Theorem 1.1, our improvement are obtained via exploiting that the graphs H in G need not be subgraphs of G. For instance, if F are thetas, then Chudnovsky and Seymour [18] obtained a set G of O(n^5) subgraphs of G. Each H ∈ G with L(H) = {a_1, a_2, a_3} is uniquely determined from vertices b, b_1, b_2, b_3, a_1, a_2, a_3 of G such that b*b_1b_2b_3, b_1a_1b_2a_3, b_2a_1b_3 are the distinct edges of G[(b, b_1, b_2, b_3, a_1, a_2, a_3)]. We observe that the requirement that a_1b_1, a_2b_2, a_3b_3 are the distinct edges of G[(b, b_1, b_2, b_3, a_1, a_2, a_3)] can be achieved by making the neighbors of each b_i with i ∈ {1, 2, 3} in V(G) \ {b, b_1, b_2, b_3} a clique. As a result, each H ∈ G is determined from four vertices b, b_1, b_2, b_3 such that b*b_1b_2b_3, b_1a_1b_2a_3, b_2a_1b_3 are the distinct edges of G[(b, b_1, b_2, b_3)]. Thus, there is a set G of O(n^4)

n-vertex graphs H with L(H) = {b_1, b_2, b_3, a_1, a_2, a_3} such that G is theta-free if and only if each graph H in G does not admit an induced tree containing L(H). An n^2-factor is reduced from the number of the three-in-a-tree problems to be solved in order to determine whether G is theta-free. Beetle detection can be improved similarly. Improving the algorithm for pyramid detection needs additional care, since a pyramid has to contain exactly one triangle. Theorem 1.2 is immediate from Theorem 1.1 and the next lemma.

LEMMA 6.1. Let T be a graph in G can be detected by solving the three-in-a-tree problem on O(mn^2) linear-time-obtainable n-vertex graphs.

A pyramid [18] of G is the subgraph induced by the vertices of an induced subtree T of G – {b_1b_2b_2b_3b_3b_1} with exactly 3 leaves b_1, b_2, b_3 such that G[{b_1, b_2, b_3}] is the triangle of G[T]. Theorem 1.3 is immediate from Theorem 1.1 and the next lemma.

LEMMA 6.2. Let T be a graph in G can be detected by solving the three-in-a-tree problem on O(mn^2) linear-time-obtainable n-vertex graphs.

A beetle [11] of G is an induced subgraph of G consisting of a cycle b_1b_2b_3b_4b_1 with a chord b_2b_4 (i.e., a diamond [25, 45] of G) and a tree T of G – b_1, having exactly 3 leaves b_1, b_2, and b_3. Theorem 1.4 is immediate from Theorem 1.1 and the next lemma.

LEMMA 6.3. Let T be a graph in G can be detected by solving the three-in-a-tree problem on O(mn^2) linear-time-obtainable n-vertex graphs.

6.2 Perfect Graph

As summarized by Maffray and Trotignon [51, §2], the algorithm of Chudnovsky et al. [13] consists of two O(n^6)-time phases. Phase 1 (a) detects pyramids in G in O(n^6) time, (b) detects the so-called T_C configurations with i ∈ {1, 2, 3} in O(n^6) time, and (c) detects pyramids in G in O(n^6) time. If any of them is detected, then either G or G contains odd holes, implying that G is not perfect. Otherwise, each shortest odd hole C of G is amenable, i.e., any anti-connected component of the C-major vertices is contained by N_G(u) ∩ N_G(v) for some edge uv of C. Phase 2 (a) computes in O(n^8) time a set X of O(n^6) subsets of V(G) such that if G contains an amenable shortest odd hole, then X contains a near cleaner of G and (b) spends O(n^8) time on each x ∈ X to either obtain an odd hole of G or ensure that X is not a near cleaner of G. Theorem 1.3 reduces the time of detecting pyramids to O(n^8). Lemma 6.5 reduces the time of Phase 2(b) from O(n^6) to the time of performing O(n) multiplications of Boolean n × n matrices [27, 47, 61]. Therefore, the time of recognizing perfect graphs is already reduced to O(n^8) without resorting to our improved odd-hole detection algorithm.

Let G be an n-vertex m-edge graph. A k-hole (respectively, k-cycle and k-path) is a k-vertex hole (respectively, cycle and path). For any odd hole C of G, a vertex x ∈ V(G) \ V(C) is C-major [13] if N_G(x, C) is not contained by any 3-path of C. Let M_G(C) consist of the C-major vertices. We have M_G(C) ∩ V(C) = ∅. A shortest odd hole C of G is clean if G does not contain any C-major vertex. A set X ⊆ V(G) is a near cleaner [13] if there is a shortest odd hole C of G such that (1) C[X] is contained by a 3-path of C and (2) all C-major vertices of G are in X. A jewel of G is an O(n^8)-time detectable induced subgraph of G [13]. If G contains jewels or beetles, then G contains odd holes. Let G denote the complement of graph G.

1As in [51] we omit the complicated definitions of T_i configurations, which are not needed by our improved algorithms.
Lemma 6.4 (Chudnovsky et al. [13, 4.1]). Let \( u \) and \( v \) be distinct vertices of a clean shortest odd hole \( C \) of a pyramid-free jewel-free graph \( G \). (1) The shortest \( u \)-path of \( C \) is a shortest \( u \)-path of \( G \). (2) The graph obtained from \( C \) by replacing the shortest \( u \)-path of \( C \) with a shortest \( u \)-path of \( G \) remains a clean shortest odd hole of \( G \).

We already have a faster algorithm for recognizing perfect graphs without using Theorem 1.4(1). Lemma 6.5 reduces the time of Chudnovsky et al.'s algorithms [13, 4.2 and 5.1] from \( O(n^3) \) to \( O(n^{3.377}) \).

Lemma 6.5. For any given vertex set \( X \) of an \( n \)-vertex pyramid-free jewel-free graph \( G \), it takes the time of performing \( O(n) \) multiplications of \( n \times n \) Boolean matrices to either obtain an odd hole of \( G \) or ensure that \( X \) is not a near cleaner of a shortest odd hole of \( G \).

Lemma 6.6 (Chudnovsky et al. [13]). Let \( G \) be an \( n \)-vertex graph such that \( G \) and \( \bar{G} \) are pyramid-and-jewel-free. It takes \( O(n^6) \) time to (1) ensure that \( G \) contains odd holes or (2) obtain a set \( X \) of \( O(n^5) \) vertex subsets of \( G \) such that if \( G \) contains odd holes, then \( X \) contains a near cleaner of \( G \).

By Theorem 1.3, it takes \( O(n^6) \) time to detect pyramids or jewels in \( G \) and \( \bar{G} \). If \( G \) or \( \bar{G} \) contains pyramids or jewels, then \( G \) is not perfect. By Lemma 6.6, it suffices to consider the case that we are given a set \( X \) of \( O(n^5) \) vertex subsets such that if \( G \) or \( \bar{G} \) is not odd-hole-free, then \( X \) contains a near cleaner of \( G \) or \( \bar{G} \). By Lemma 6.5, it takes overall \( O(n^{8.377}) \) time [27, 47, 61] to either obtain an odd hole of \( G \) or \( \bar{G} \) or ensure that both \( G \) and \( \bar{G} \) are odd-hole-free.

6.3 Odd Hole
Chudnovsky et al.'s odd-hole detection algorithm has seven \( O(n^9) \)-time bottleneck subroutines. One is for pyramid detection, which is eliminated by Theorem 1.3. The remaining six are in two groups [17, §4]. The first (respectively, second) group handles the case that the longest \( x \)-gap (i.e., a path \( D \) of \( C \) such that \( G[D \cup \{x\}] \) is a hole of \( G \)) over all \( C \)-major vertices \( x \) for a shortest odd hole \( C \) is shorter (respectively, longer) than one half of \( C \). We give a two-phase algorithm to handle both cases in \( O(n^2) \) time. For the first case, Phase 1 tries all \( O(n^2) \) choices of 5 vertices to obtain an approximate cleaner for \( C \), with which a shortest odd hole can be found in \( O(n^3) \) time via Lemma 6.5. For the second case, Phase 2 tries all \( O(n^2) \) choices of six vertices to obtain an approximate cleaner for \( C \), with which a shortest odd hole can be identified in \( O(n^2) \) time.

6.4 Even Hole
Chang et al.'s algorithm consists of two \( O(n^{11}) \)-time phases. Phase 1 detects beetles in \( O(n^{11}) \) time, which is reduced to \( O(n^7) \) time by Theorem 1.5. Phase 2 maintains a set \( T \) of induced subgraphs of \( G \) with the property that if \( G \) is even-hole-free, then so is each graph in \( T \) until \( T \) becomes empty or an \( H \in T \) is found to contain even holes. The initial \( T \) consists of \( O(n^3) \) graphs obtained from guesses of (1) a 3-path \( P \) on a shortest even hole \( C \) of \( G \), (2) an \( X \subseteq V(G) \) that contains the major vertices of \( C \) disjoint from \( C \), and (3) a 3-path \( Y \subseteq V(G) \) that contains \( N^2 \) of \( C \) disjoint from \( C \). Each iteration of Phase 2 takes \( O(n^4) \) time to ensure that an \( H \in T \) is an extended clique tree that contains even holes or replace \( H \) with 0 (resp., 1 and 2) smaller graphs via ensuring that \( H \) is an even-hole-free extended clique tree (resp., decomposing \( H \) by a star-cutset and decomposing \( H \) by a 2-join). The guessed \( P \) and \( Y \) are crucial in arguing that \( H \) can be decomposed by a star-cutset without increasing \( |T| \), implying that each initial \( H \in T \) incurs \( O(n) \) decompositions by star-cutsets. Therefore, the overall time for decompositions by star-cutsets is \( O(n^{10}) \), i.e., \( O(n^{5}) \) times the initial \( |T| \). Each initial \( H \in T \) incurs \( O(n^2) \) decompositions by 2-joins, implying that the overall time for detecting even holes in extended clique trees and decompositions by 2-joins is \( O(n^{11}) \), i.e., \( O(n^6) \) times the initial \( |T| \). We reduce the time of Phase 2 from \( O(n^{11}) \) to \( O(n^7) \). A factor of \( n \) is removed by reducing the initial \( |T| \) from \( O(n^4) \) to \( O(n^3) \) via ignoring \( Y \) and the internal vertex of \( P \). Guessing only \( X \) and the end-vertices of \( P \) does complicate the task of decomposing \( H \) by a star-cutset, but we manage to handle each decomposition by a star-cutset in the same time bound. Another factor of \( n \) is removed by reducing the number of decompositions by 2-joins incurred by each initial \( H \in T \) from \( O(n^2) \) to \( O(n) \) via carefully handling the boundary cases.

7 CONCLUDING REMARKS
We solve the three-in-a-tree problem on an \( n \)-vertex \( m \)-edge undirected graph in \( O(m \log^2 n) \) time, leading to improved algorithms for recognizing perfect graphs and detecting pyramids, pyramids, beetles, and odd and even holes. It would be interesting to see if the complexity of the three-in-a-tree problem can be further reduced. The amortized cost of maintaining the connectivity information for the dynamic graph \( G - X \) can be improved to \( O(\log^2 n \log \log n) \) using [63] or even to \( O(\log n \log \log O(1)) \) using [57]. Since \( G - X \) is purely decremental, we can use the randomized algorithm in [56] for further speedup. However, this is not our only \( O(\log^2 n) \) bottleneck: At the moment we pay \( O(\log n) \) time for each neighbor of a vertex in \( X \) when it changes color, so if it changes color \( O(\log n) \) times, then it will be hard to beat the \( O(\log^2 n) \) factor.

ACKNOWLEDGMENTS
We thank the anonymous reviewers and Evangelos Kipouridis for helpful comments. We thank Ho-Lin Chen and Meng-Tsung Tsai for commenting on a preliminary version [46] of the paper.

REFERENCES