# Insertion-Only Dynamic Connectivity in General Disk Graphs

**Haim Kaplan[1], Katharina Klost[2], Kristin Knorr*[2], Wolfgang Mulzer†[2], and Liam Roditty[3]**

1 **School of Computer Science, Tel Aviv University**
`haimk@tau.ac.il`

2 **Institut für Informatik, Freie Universität Berlin**
`{kathklost, knorrkri, mulzer}@inf.fu-berlin.de`

3 **Department of Computer Science, Bar Ilan University**
`liamr@macs.biu.ac.il`

## Abstract

Let $S \subseteq \mathbb{R}^2$ be a set of $n$ *sites* in the plane, so that every site $s \in S$ has an *associated radius* $r_s > 0$. Let $\mathcal{D}(S)$ be the *disk intersection graph* defined by $S$, i.e., the graph with vertex set $S$ and an edge between two distinct sites $s, t \in S$ if and only if the disks with centers $s$, $t$ and radii $r_s$, $r_t$ intersect. Our goal is to design data structures that maintain the connectivity structure of $\mathcal{D}(S)$ as $S$ changes dynamically over time.

We consider the incremental case, where new sites can be inserted into $S$. While previous work focuses on data structures whose running time depends on the ratio between the smallest and the largest site in $S$, we present a data structure with $O(\alpha(n))$ amortized query time and $O(\log^6 n)$ expected amortized insertion time.

## 1 Introduction

The question if two vertices in a given graph are connected is crucial for many applications. If multiple such *connectivity queries* need to be answered, it makes sense to preprocess the graph into a suitable data structure. In the static case, where the graph does not change, we get an optimal answer by using a graph search to determine the connected components and by labeling the vertices with their respective components. In the dynamic case, where the graph can change over time, things get more interesting, and many variants of the problem have been studied.

We construct an insertion-only dynamic connectivity data structure for disk graphs. Given a set $S \subseteq \mathbb{R}^2$ of $n$ sites in the plane with associated radii $r_s$ for each site $s$, the *disk graph $\mathcal{D}(S)$ for $S$* is the intersection graph of the disks $D_s$ induced by the sites and their radii. While $\mathcal{D}(S)$ is represented by $O(n)$ numbers describing the disks, it might have $\Theta(n^2)$ edges. Thus, when we start with an empty disk graph and successively insert sites, up to $\Omega(n^2)$ edges may be created. We describe a data structure whose overall running time for any sequence of $n$ site insertions is $o(n^2)$, while allowing for efficient connectivity queries. For unit disk graphs (i.e., all associated radii $r_s = 1$), a fully dynamic data structure with a similar performance guarantee is already given by Kaplan et al. [2]. In the same paper, Kaplan et al. present an incremental data structure whose running time depends on the ratio

* Supported by the German Science Foundation within the research training group ‘Facets of Complexity’ (GRK 2434).
† Supported in part by ERC StG 757609.

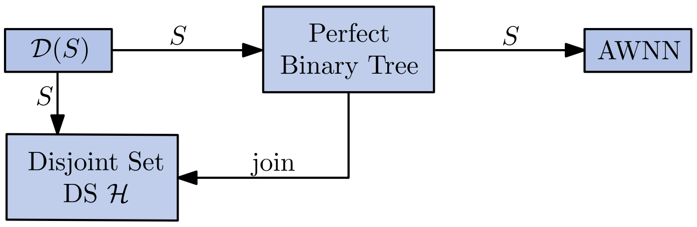

Figure 1 Overview of the data structure (Theorem 2.5)

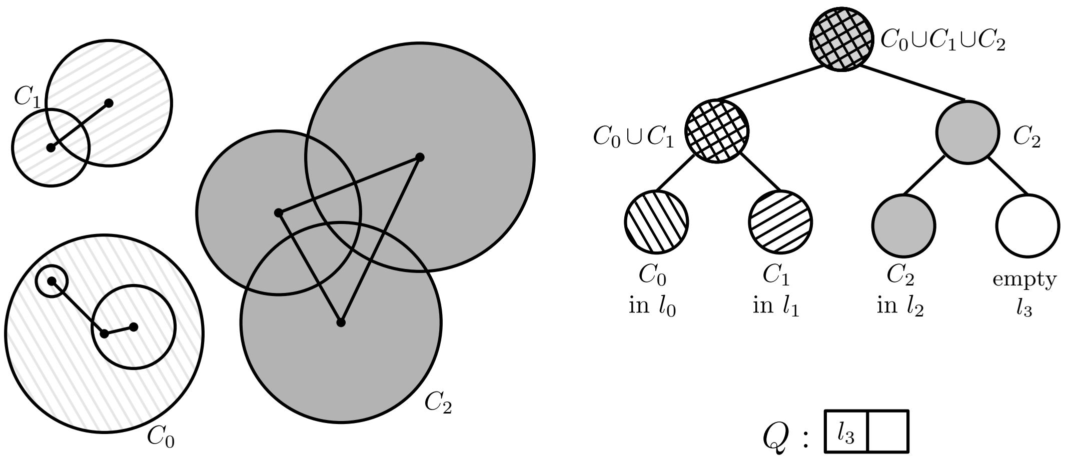

Figure 2 Disk graph with associated component tree and queue $Q$ (tiling of a component corresponds to the tiling of nodes storing its disks).

$\Psi$ of the smallest and the largest radius in $S$. In this setting, they achieve $O(\alpha(n))$ amortized query time and $O(\log(\Psi)\log^4 n)$ expected amortized insertion time.

We focus on general disk graphs, with no assumption on the radius ratio. Our approach has two main ingredients. First, we simply represent the connected components in a data structure for the disjoint set union problem [1]. This allows for fast queries, in $O(\alpha(n))$ amortized time, also mentioned by Reif [5]. The second ingredient is an efficient data structure to find all components in $\mathcal{D}(S)$ that are intersected by any given disk (and hence tells us which components need to be merged after an insertion of a new site). A schematic overview of the data structure is given in Figure 1.

## 2 Insertion-Only Data Structure for Unbounded Radii

As in the incremental data structure for disk graphs with bounded radius ratio by Kaplan et al. [2], we use a disjoint set union data structure to represent the connected components of $\mathcal{D}(S)$ and to perform the connectivity queries. To insert a new site $s$ into $S$, we first find the set $\mathcal{C}_s$ of components in $\mathcal{D}(S)$ that are intersected by $D_s$. Then, once $\mathcal{C}_s$ is known, we can simply update the disjoint set union structure to support further queries.

In order to identify $\mathcal{C}_s$ efficiently, we use a *component tree* $T_{\mathcal{C}}$. This is a binary tree whose leaves store the connected components of $\mathcal{D}(S)$. The idea is illustrated in Figure 2 showing a disk graph and its associated component tree. We require that $T_{\mathcal{C}}$ is a complete binary tree, and some of its leaves may not have a connected component assigned to them, like $l_3$ in Figure 2. Those leaves are *empty*. Typically, we will not distinguish the leaf

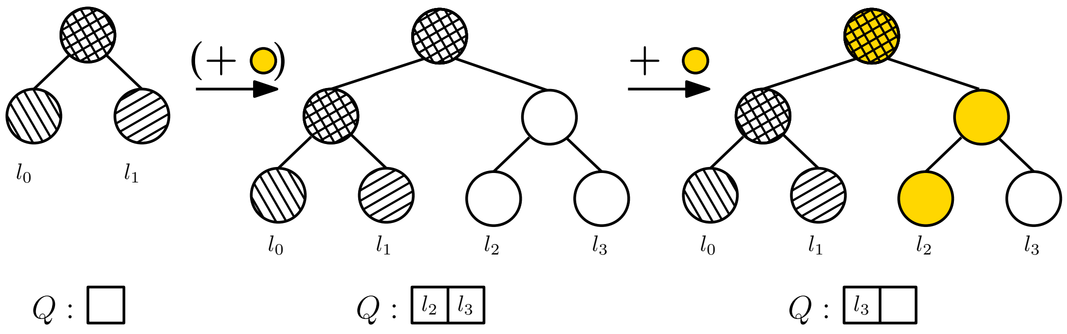

**Figure 3** If the tree has no empty leaves before the insertion of an isolated component, a new root and an empty subtree are added (second tree is an intermediate state before actual insertion).

storing a connected component and the component itself. Also when suitable, we will treat a connected component as a set of sites. Every node of $T_{\mathcal{C}}$ stores a fully dynamic additively weighted nearest neighbor data structure (AWNN). An AWNN stores a set $P$ of $n$ points, each associated with a weight $w_p$. On a nearest neighbor query with a point $q \in \mathbb{R}^2$ it returns the point $p \in P$ that minimizes $\|pq\| + w_p$. For this data structure, we use the following result by Kaplan et al. [3] with an improvement by Liu [4].

▶ **Lemma 2.1** (Kaplan et al. [3, Theorem 8.3, Section 9], Liu [4, Corollary 4.3]). *There is a fully dynamic AWNN data structure that allows insertions in $O(\log^2 n)$ amortized expected time and deletions in $O(\log^4 n)$ amortized expected time. Furthermore, a nearest neighbor query takes $O(\log^2 n)$ worst case time. The data structure requires $O(n \log n)$ space.*

The component tree maintains the following invariants. Invariant 2 allows us to use a query to the AWNN to find the disk whose boundary is closed to a query point.

**Invariant 1:** Every connected component of $\mathcal{D}(S)$ is stored in exactly one leaf of $T_{\mathcal{C}}$, and

**Invariant 2:** The AWNN of a node $u \in T_{\mathcal{C}}$ contains the sites of all connected components that lie in the subtree rooted at $u$, where a site $s \in S$ has assigned weight $-r_s$.

In addition to $T_{\mathcal{C}}$, we store a queue $Q_{\mathcal{C}}$ that contains exactly the empty leaves in $T_{\mathcal{C}}$.

We now describe how to update $T_{\mathcal{C}}$ when a new site $s$ is inserted. We maintain $T_{\mathcal{C}}$ in such a way that the structure of $T_{\mathcal{C}}$ changes only when $s$ creates a new isolated component in $\mathcal{D}(S)$. In the following lemma, we consider the slightly more general case of inserting a new connected component $C$ that does not intersect any connected component already stored in $T_{\mathcal{C}}$.

▶ **Lemma 2.2.** *Let $T_{\mathcal{C}}$ be a component tree of height $h$ with $n$ sites and let $C$ be an isolated connected component. We can insert $C$ into $T_{\mathcal{C}}$ in amortized time $O(h \cdot |C| \cdot \log^2 n)$.*

**Proof.** The insertion performs two basic steps: first, we find or create an empty leaf $l_i$ into which $C$ can be inserted. Second, the AWNN structures along the path from $l_i$ to the root of $T_{\mathcal{C}}$ are updated.

For the first step, we check if $Q_{\mathcal{C}}$ is non-empty. If so, we extract the first element from $Q_{\mathcal{C}}$ to obtain our empty leaf $l_i$. If $Q_{\mathcal{C}}$ is empty, there are no empty leaves, and we have to expand the component tree. For this, we create a new root for $T_{\mathcal{C}}$, and we attach the old tree as one child. The other child is an empty complete tree of the same size as the old tree. This creates a complete binary tree, see intemediate state in Figure 3. We copy the AWNN of the former root to the new root, we add all new empty leaves to $Q_{\mathcal{C}}$, and we extract $l_i$ from $Q_{\mathcal{C}}$.

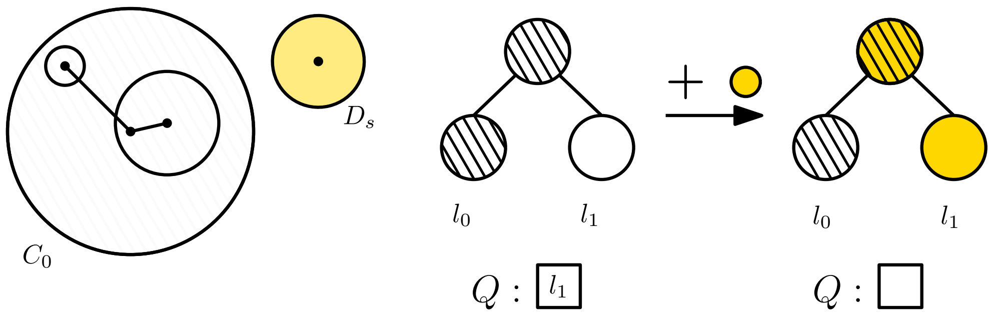

**Figure 4** Inserting a component (potentially isolated disk $D_s$) into an empty leaf ($C_i$ stored in $l_i$): The isolated component $C$, yellow disk $D_s$, is inserted in the empty leaf $l_1$ and all its ancestors (indicated by coloring).

For the second step, we insert $C$ into $l_i$, and we store an AWNN structure with the sites from $C$ in $l_i$. Then, the AWNN structures on all ancestors of $l_i$ are updated by inserting the sites of $C$, see Figure 4 and Figure 3 in case of tree extension respectively.

This procedure maintains both invariants: since an isolated component does not affect the remaining connected components of $\mathcal{D}(S)$, it has to be inserted into a new leaf, maintaining the first invariant. The second invariant is taken care of in the second step, by construction. Afterwards, the queue has the correct state, since we extract the leaf used in the insertion.

The running time for finding or creating an empty leaf is amortized $O(1)$. This is immediate if $Q_{\mathcal{C}} \neq \emptyset$, and otherwise, we can charge the cost of building the empty tree, inserting the empty leaves into $Q_{\mathcal{C}}$, and producing an AWNN structure for the new root to the previous insertions. The most expensive step consists in updating the AWNN structures for the new component. In each of the $h$ AWNN structures of the ancestors of $l_i$, we must insert $|C|$ disks. By Lemma 2.1, this results into an expected amortized time of $O(h \cdot |C| \cdot \log^2 n)$. ◀

Next, we describe how to find the set $\mathcal{C}_s$ of connected components that are intersected by a disk $D_s$.

▶ **Lemma 2.3.** *Let $T_{\mathcal{C}}$ be a component tree of height $h$ that stores $n$ disks. We can find $\mathcal{C}_s$ in worst case time $O(\max\{|\mathcal{C}_s| \cdot h, 1\} \cdot \log^2 n\})$.*

**Proof.** First, observe that if the site returned by a query to an AWNN structure with $s$ does not intersect $D_s$, then $D_s$ does not intersect any disk for the sites stored in this AWNN. Thus, the case where $\mathcal{C}_s = \emptyset$ can be identified by a query to the AWNN structure in the root of $T_{\mathcal{C}}$, in $O(\log^2 n)$ time.

In any other case, we perform a top down traversal of $T_{\mathcal{C}}$. Let $u$ be the current node. We query the AWNN structures of both children of $u$ with $s$, and we recurse only into the children where the nearest neighbor intersects $D_s$. The set $\mathcal{C}_s$ then contains exactly the connected components of all leaves where $D_s$ intersects its weighted nearest neighbor. Since every leaf found corresponds to one connected component intersecting $D_s$ and we recurse into all subtrees whose union of sites have a non-empty intersection with $s$, we do not miss any connected components.

For every connected component, there are at most $h$ queries to AWNN structures along the path from the root to the components. A query takes $O(\log^2 n)$ amortized time by Lemma 2.1, giving an amortized time of $O(|\mathcal{C}_s| \cdot h \cdot \log^2 n)$, if $s$ is not isolated. The overall time follows from taking the maximum of both cases. ◀

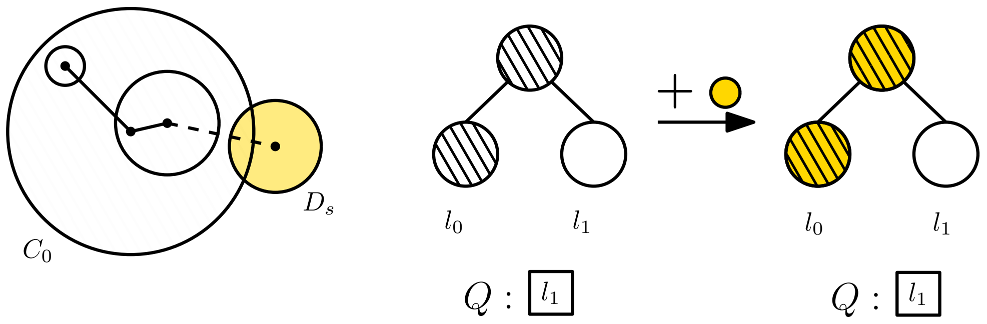

**Figure 5** Inserting a site if $|\mathcal{C}_s| = 1$ ($C_i$ stored in $l_i$): The yellow disk $D_s$ intersects $C_0$ (dashed edge). Site $s$ is added to the AWNN of the associated leaf $l_0$ and its ancestors.

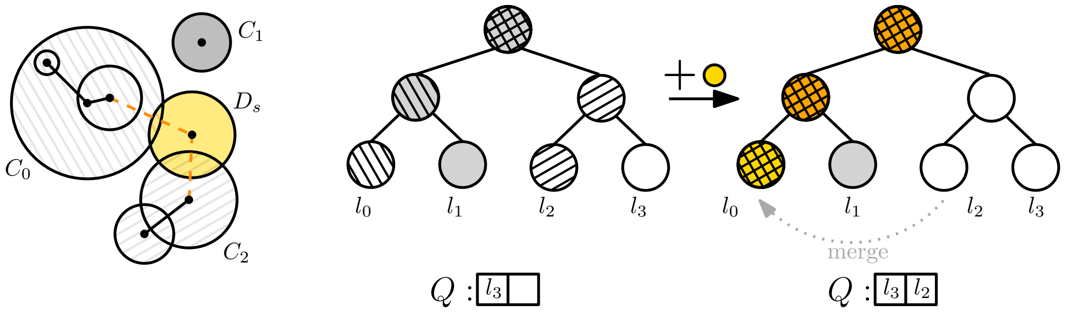

**Figure 6** Inserting a site if $|\mathcal{C}_s| > 1$ ($C_i$ stored in $l_i$): $D_s$ intesects $C_0$ and $C_2$. Since $|C_0| = 3$ and $|C_2| = 2$, the largest component $C_L$ is $C_0$. Thus, $D_s$ and $C_2$ are merged into $C_0$ and $C_2$ is removed from $l_2$ up to the lca, the root. The empty leave $l_2$ is enqueued.

Using Lemma 2.2 and Lemma 2.3, we can now describe how to insert a single disk into a component tree.

▶ **Lemma 2.4.** *Let $T_\mathcal{C}$ be a component tree that stores $n$ sites. A new site $s$ can be inserted into $T_\mathcal{C}$ in $O(\log^6 n)$ amortized expected time.*

**Proof.** First, we use the algorithm from Lemma 2.3. to find $\mathcal{C}_s$. If $|\mathcal{C}_s| = 0$, we use Lemma 2.2 to insert $s$ as a singleton isolated connected component.

Otherwise, if $|\mathcal{C}_s| \geq 1$, let $C_L = \arg\max_{C \in \mathcal{C}_s} |C|$ be a largest connected component in $\mathcal{C}_s$. We insert $s$ into $C_L$ and into the AWNN structures of all ancestors of $C_L$. Then, if $|\mathcal{C}_s| = 1$, we are done, see Figure 5. If $|\mathcal{C}_s| > 1$, all components in $\mathcal{C}_s$ now form a new, larger, component in $\mathcal{D}(S)$. We perform the following *clean-up* step in $T_\mathcal{C}$.

For each component $C_i \in \mathcal{C}_s \setminus \{C_L\}$, all sites from $C_i$ are inserted into $C_L$. Let lca be the lowest common ancestor of the leaves for $C_i$ and $C_L$ in $T_\mathcal{C}$. Then all sites from $C_i$ are deleted from the AWNN structures along the path from $C_i$ to lca, and reinserted along the path from $C_L$ to lca. Finally, all newly empty leaves are inserted into $Q_\mathcal{C}$. For an illustration of the insertion of $s$ and the clean-up step, see Figure 6.

To show correctness, we again argue that the invariants are maintained. If $|\mathcal{C}_s| = 0$ this follows by Lemma 2.2. In the other case, we directly insert $s$ into a connected component intersected by $s$ and update all AWNN structures along the way. As Lemma 2.3 correctly finds all relevant connected components, and we explicitly move the sites in these components to $C_L$ during clean-up, Invariant 1 is fulfilled. In a similar vein, we update all AWNN structures of sites that move to a new connected component, satisfying Invariant 2. Moreover, we keep

$Q_{\mathcal{C}}$ updated by inserting or removing empty leaves when needed during the algorithm.

To complete the proof, it remains to analyze the running time. In the worst case, where all components are singletons, a component tree that stores $n$ sites has height $O(\log n)$. If $|\mathcal{C}_s| = 0$ the running time for finding $\mathcal{C}_s$ is $O(\log^2 n)$ by Lemma 2.3. The insertion and restructuring is done with Lemma 2.2, yielding an expected amortized time of $O(\log^3 n)$. In the case $|\mathcal{C}_s| = 1$, with $h = O(\log n)$ a running time of $O(\log^3 n)$ for finding $\mathcal{C}_s$ follows by Lemma 2.3. Following similar arguments to the case $|\mathcal{C}_s| = 0$, the time needed for the insertion and restructuring is expected amortized $O(\log^3 n)$.

Finally, we consider the case $|\mathcal{C}_s| > 1$. By Lemma 2.3, finding $\mathcal{C}_s$ takes worst case time $O(|\mathcal{C}_s| \cdot \log^3 n)$. Then the insertion of $s$ can be done in expected amortized time $O(\log^3 n)$, as in the cases above. It remains to analyze the running time of the clean-up step. We know that the first common ancestor might be the root of $T_{\mathcal{C}}$. Hence, in the worst case, we have to perform $\sum_{C_i \in (\mathcal{C}_s \setminus \{C_L\})} |C_i| \cdot O(\log n)$ insertions and deletions for a single clean-up step. As the time for the deletions in the AWNN structures dominates, this is expected amortized $\sum_{C_i \in \mathcal{C}_s \setminus \{C_L\}} O(|C_i| \cdot \log^5 n)$ worst case time. Note that since $|C_i| \geq 1$ and we have to insert $s$, the running time of Lemma 2.3 is dominated by the clean-up step.

The overall time spent on all clean-up steps over all insertions is then upper bounded by $\sum_{s \in S} \sum_{C_i \in (\mathcal{C}_s \setminus \{C_L\})} O(|C_i| \cdot \log^5 n)$. Observe, that during the lifetime of the component tree, each disk can only be merged into $O(\log n)$ connected components. Thus, we have

$$\sum_{s \in S} \sum_{C_i \in \mathcal{C}_s \setminus \{C_L\}} |C_i| = O(n \log n),$$

and the overall expected time spent on clean-up steps is $O(n \log^6 n)$. As the case $|\mathcal{C}_s| > 1$ turned out to be the most complex case, the overall running time follows. ◀

▶ **Theorem 2.5.** *There is an incremental data structure for connectivity queries in disk graphs with $O(\alpha(n))$ amortized query time and $O(\log^6 n)$ expected amortized update time.*

**Proof.** We use a component tree as described above to maintain the connected components. Additionally, we maintain a disjoint set data structure $\mathcal{H}$, where each connected component forms a set, see Figure 1. Queries are performed directly in $\mathcal{H}$ in $O(\alpha(n))$ amortized time.

When inserting an isolated component during the update, this component is added to $\mathcal{H}$. When merging several connected components in the clean-up step, this change is reflected in $\mathcal{H}$ by suitable union operations. The time for updates in the component tree dominates the updates in $\mathcal{H}$, leading to an expected amortized update time of $O(\log^6 n)$. ◀

## 3 Conclusion

We introduced a data structure that solves the incremental connectivity problem in general disk graphs with $O(\alpha(n))$ amortized query and $O(\log^6 n)$ amortized expected update time. The question of finding efficient fully-dynamic data structures for both the general and the bounded case remains open.

## References

1 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

2 Haim Kaplan, Alexander Kauer, Katharina Klost, Kristin Knorr, Wolfgang Mulzer, Liam Roditty, and Paul Seiferth. Dynamic connectivity in disk graphs. In *38th International*

*Symposium on Computational Geometry (SoCG 2022)*, pages 49:1–49:17, 2022. `doi:10.4230/LIPIcs.SoCG.2022.49`.

3 Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. *Discrete Comput. Geom.*, 64(3):838–904, 2020. `doi:10.1007/s00454-020-00243-7`.

4 Chih-Hung Liu. Nearly optimal planar $k$ nearest neighbors queries under general distance functions. *SIAM Journal on Computing*, 51(3):723–765, 2022. `doi:10.1137/20m1388371`.

5 John H. Reif. A topological approach to dynamic graph connectivity. *Information Processing Letters*, 25(1):65–70, 1987. `doi:10.1016/0020-0190(87)90095-0`.