# Dynamic Parameterized Problems on Unit Disk Graphs

**Shinwoo An**
Department of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea

**Kyungjin Cho**

Department of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea

**Leo Jang**
Department of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea

**Byeonghyeon Jung**
Department of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea

**Yudam Lee**
Department of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea

**Eunjin Oh**

Department of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea

**Donghun Shin**
Department of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea

**Hyeonjun Shin**

Department of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea

**Chanho Song**

Department of Computer Science and Engineering, POSTECH, Pohang, Republic of Korea

## Abstract

In this paper, we study fundamental parameterized problems such as $k$-Path/Cycle, Vertex Cover, Triangle Hitting Set, Feedback Vertex Set, and Cycle Packing for *dynamic* unit disk graphs. Given a vertex set $V$ changing dynamically under vertex insertions and deletions, our goal is to maintain data structures so that the aforementioned parameterized problems on the unit disk graph induced by $V$ can be solved efficiently. Although dynamic parameterized problems on general graphs have been studied extensively, no previous work focuses on unit disk graphs. In this paper, we present the first data structures for fundamental parameterized problems on dynamic unit disk graphs. More specifically, our data structure supports $2^{O(\sqrt{k})}$ update time and $O(k)$ query time for $k$-Path/Cycle. For the other problems, our data structures support $O(\log n)$ update time and $2^{O(\sqrt{k})}$ query time, where $k$ denotes the output size.

**2012 ACM Subject Classification** Theory of computation → Computational geometry

**Keywords and phrases** Unit disk graphs, dynamic parameterized algorithms, kernelization

**Digital Object Identifier** 10.4230/LIPIcs.ISAAC.2024.6

**Related Version** *Full Version*: `https://arxiv.org/abs/2409.13403`

**Funding** This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00440239, Sublinear Scalable Algorithms for Large-Scale Data Analysis) and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.RS-2024-00358505).
*Kyungjin Cho*: Supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.RS-2024-00410835).

© Shinwoo An, Kyungjin Cho, Leo Jang, Byeonghyeon Jung, Yudam Lee, Eunjin Oh, Donghun Shin, Hyeonjun Shin, and Chanho Song;
licensed under Creative Commons License CC-BY 4.0
35th International Symposium on Algorithms and Computation (ISAAC 2024).
Editors: Julián Mestre and Anthony Wirth; Article No. 6; pp. 6:1–6:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# 1 Introduction

For a set $V$ of $n$ points in the plane, the *unit disk graph* of $V$ is the intersection graph of the unit disks of diameter one centered at the points in $V$, denoted by $\mathsf{UD}(V)$. Unit disk graphs serve as a powerful model for real-world applications such as broadcast networks [28, 29], biological networks [23] and facility location [34]. Due to various applications, unit disk graphs have gained significant attention in computational geometry. Since most of the fundamental NP-hard problems remain NP-hard even in unit disk graphs, the study of NP-hard problems on unit disk graphs focuses on approximation algorithms and parameterized algorithms [4, 6, 17, 21, 37]. From the perspective of parameterized algorithms, the main focus is to design subexponential-time parameterized algorithms for various problems on unit disk graphs. While such algorithms do not exist for general graphs unless ETH fails, lots of problems admit subexponential-time parameterized algorithms for unit disk graphs.

In this paper, we study fundamental graph problems on *dynamic* unit disk graphs. Given a vertex set $V$ that changes under vertex insertions and deletions, our goal is to maintain data structures so that specific problems for $\mathsf{UD}(V)$ can be solved efficiently. This dynamic setting has attracted considerable interest. For instance, the connectivity problem [11, 12], the coloring problem [27], the independent set problem [9, 19], the set cover problem [1], and the vertex cover problem [8] have been studied for dynamic geometric intersection graphs. Here, all problems, except for the connectivity problem, are NP-hard. All previous work on dynamic intersection graphs for those problems study approximation algorithms. However, like the static setting, parameterized algorithms are also a successful approach for addressing NP-hardness in the dynamic setting. There has been significant research on parameterized algorithms for dynamic general graphs [2, 13, 18, 26]. Surprisingly, however, there have been no studies on parameterized algorithms for dynamic unit disk graphs.

In this paper, we initiate the study of fundamental *parameterized* problems on dynamic unit disk graphs. In particular, we study the following five fundamental problems in the dynamic setting. All these problems are NP-hard even for unit disk graphs [14, 25].

- $k$-Path/Cycle asks to find a path/cycle of $G$ with exactly $k$ vertices,
- $k$-Vertex Cover asks to find a set $S$ of $k$ vertices s.t. $G \setminus S$ has no edge,
- $k$-Triangle Hitting Set asks to find a set $S$ of $k$ vertices s.t. $G \setminus S$ has no triangle,
- $k$-Feedback Vertex Set asks to find a set $S$ of $k$ vertices s.t. $G \setminus S$ has no cycle, and
- $k$-Cycle Packing asks to find $k$ vertex-disjoint cycles of $G$.

In the course of vertex updates, we are asked to solve those problems as a query. Except for $k$-Path/cycle, a query is given with an integer $k$. On the other hand, our data structure for $k$-Path/Cycle uses $k$ in the construction time.

**Table 1** Summary of our results. The results marked as * support amortized update times, and the others are worst-case update/query times. Except for $k$-Path/Cycle, no data structure requires $k$ in the construction time; The parameter $k$ is given as a query. Additionally, the data structure for $k$-Path/Cycle can answer a decision query in constant time.

| | Update time | Query time | Space Complexity |
|---|---|---|---|
| $k$-Path/Cycle* | $2^{O(\sqrt{k})}$ | $O(k)$ | $O(kn)$ |
| $k$-Vertex Cover* | $O(1)$ | $2^{O(\sqrt{k})}$ | $O(n)$ |
| $k$-Triangle Hitting Set* | $O(1)$ | $2^{O(\sqrt{k})}$ | $O(n)$ |
| $k$-Feedback Vertex Set | $O(\log n)$ | $2^{O(\sqrt{k})}$ | $O(n)$ |
| $k$-Cycle Packing | $O(\log n)$ | $2^{O(\sqrt{k})}$ | $O(n)$ |

**Our results.** Our results are summarized in Table 1. Note that these are almost ETH-tight. To see this, recall that no problem studied in this paper admits a $2^{o(\sqrt{k})}n^{O(1)}$-time algorithm in the static setting unless ETH fails [16, 20, 22]. Thus for any data structure for these problems on dynamic unit disk graphs with update time $T_u(n,k)$ and query time $T_q(n,k)$, we must have $n \cdot T_u(n,k) + T_q(n,k) = 2^{\Omega(\sqrt{k})}n^{O(1)}$ unless ETH fails. In particular, $n \cdot T_u(n,k) + T_q(n,k)$ is the time for solving the static problem using dynamic data structures; we insert the vertices one by one and then answer the query. In our case, this static running time is $2^{O(\sqrt{k})}n$ for $k$-Path/Cycle, $2^{O(\sqrt{k})} + O(n)$ for $k$-Vertex Cover and $k$-Triangle Hitting Set, and $2^{O(\sqrt{k})} + O(n \log n)$ for $k$-Feedback Vertex Set and $k$-Cycle Packing. Interestingly, as by-products, we slightly improve the running times of the best-known static algorithms in [4, 6] for $k$-Feedback Vertex Set and $k$-Cycle Packing on unit disk graphs from $2^{O(\sqrt{k})}n^{O(1)}$ to $2^{O(\sqrt{k})} + O(n \log n)$.[1]

A main tool used in this paper is *kernelization*, a technique compressing an instance of a problem into a small-sized equivalent instance called a *kernel*. Kernelization is one of the fundamental techniques used in the field of parameterized algorithms [15]. We use the same framework for all problems, except for $k$-Path/Cycle: For each update, we maintain kernels for the current unit disk graph. Given a query, it is sufficient to solve the problem on the kernel instead of the entire unit disk graph. Precisely, if the kernel size exceeds a certain bound, we immediately return a correct answer. Otherwise, the kernel size is small, say $O(k)$, and thus we can answer the query by applying the static algorithms on the kernel.

**Related work.** While dynamic parameterized problems on unit disks have not been studied before, static parameterized algorithms have been widely studied for unit disk graphs. For instance, Fomin et al. [20] presented $2^{O(\sqrt{k}\log k)}n^{O(1)}$-time algorithms for $k$-Path/Cycle, $k$-Vertex Cover, $k$-Feedback Vertex Set, and $k$-Cycle Packing problems on static unit disk graphs. Subsequently, the running times were improved to $2^{O(\sqrt{k})}(n+m)$ [4, 6, 15, 16, 21], which are all ETH-tight. Additionally, for disk graphs, subexponential time FPT algorithms were studied for $k$-Vertex Cover and $k$-Feedback Vertex Set [3, 30].

On the other hand, there are several previous works on dynamic parameterized problems on *general graphs*. Alman et al. [2] presented a dynamic algorithm for $k$-Vertex Cover supporting $1.2738^{O(k)}$ query time and $O(1)$ amortized update time. They also presented a dynamic algorithm for $k$-Feedback Vertex Set supporting $O(k)$ query time and $k^{O(k)} \log^{O(1)} n$ amortized update time. Korhonen et al. [26] presented a dynamic algorithm for CMSO testing, parameterized by treewidth. Chen et al. [13] and Dvořák et al. [18] presented a dynamic algorithm for $k$-Path/Cycle and for MSO testing, respectively, parameterized by treedepth. These algorithms admit *edge* insertions and deletions, and Dvořák et al. [18] also admits *isolated vertex* insertions and deletions, while we deal with vertex insertions and deletions.

An alternative way for dealing with NP-hardness is using approximation. There are numerous works on approximation algorithms for dynamic intersection graphs. For disks, one can maintain $(1+\varepsilon)$-approximation of Vertex Cover [8, 24]. Bhore et al. [9] presented a constant-factor approximation algorithm for the maximum independent set problem for disks. They generalized their result on comparable-sized fat object graphs with the approximation factor depending on a given dimension and fatness parameter. For intervals and unit-squares, Agarwal et al. [1] presented a constant-approximation algorithm for the set cover problem and the hitting set problem.

[1] Moreover, they can be improved further to take $2^{O(\sqrt{k})} + O(n)$ time with a slight modification.

## 2 Preliminaries

Throughout this paper, we let $V$ be a set of points in the plane, and we let $\mathsf{UD}(V)$ be the *unit disk graph* of $V$. We interchangeably denote $v \in V$ as a point or as a vertex of $\mathsf{UD}(V)$ if it is clear from the context. For an undirected graph $G$, we often use $V(G)$ and $E(G)$ to denote the vertex set of $G$ and the edge set of $G$, respectively. For convenience, we denote the subgraph of $G$ induced by $V(G) \setminus U$ by $G \setminus U$ for a subset $U$ of $V(G)$.

**Grid.** A grid $\boxplus$ is a partition of the plane into squares (called grid cells) of diameter one. Notice that any two points of $V$ contained in the same grid cell of $\boxplus$ are adjacent in $G$. Each grid cell $\square$ has its own id: $(\lfloor a/\sqrt{2}\rfloor, \lfloor b/\sqrt{2}\rfloor)$, where $a$ and $b$ are the $x$- and $y$-coordinates of a point in $\square$. For any two grid cells $\square, \square'$ with id $(x, x')$ and $(y, y')$, respectively, we let $d(\square, \square') = \max(|x-x'|, |y-y'|)$. For an integer $\ell > 0$, a grid cell $\square$ is called an *$\ell$-neighboring cell* of a grid cell $\square'$ if $d(\square, \square') \leq \ell$. See Figure 1(a) in the full version. We slightly abuse the notation so that $\square$ itself is an $\ell$-neighboring cell of $\square$ for all $\ell > 0$. For a point $v$ in the plane, we use $\square_v$ to denote the cell of $\boxplus$ containing $v$. If it lies on the boundary of a cell, $\square_v$ denotes an arbitrary cell containing $v$ on its boundary.

We do not construct the grid $\boxplus$ explicitly. Instead, we maintain the grid cells containing vertices of $V$ only. We associate each id with a linked list that stores all the vertices of $V$ contained in the grid cell. Once we have the id of $\square$, we fetch the linked list associated with $\square$ in amortized constant time or $O(\log n)$ worst-case time, where $n$ is the number of points of $V$. More specifically, this can be implemented in $O(1)$ amortized time using dynamic perfect hashing (once true randomness is available) and in $O(\log n)$ worst-case time using 1D range search tree along the lexicographic ordering of the ids. Also, each update of the linked list can be done in the same time bound.

In this paper, we access grid cells only for updating data structures, and then we store the necessary grid cells explicitly in our data structures. Consequently, the update times of our data structures are sometimes analyzed using amortized analysis while the query times are always analyzed using worst-case analysis.

**Link-cut tree.** When we update data structures, we use link-cut trees. A *link-cut tree* is a dynamic data structure that maintains a collection of vertex-disjoint rooted trees and supports two kinds of operations: a link operation that combines two trees into one by adding an edge, and a cut operation that divides one tree into two by deleting an edge [33]. See Figure 1(b–c) in the full version. Each operation requires $O(\log n)$ time. More precisely, the data structure supports the following query and update operations in $O(\log n)$ time.

- $\mathsf{Link}(u, v)$: If $v$ is the root of a tree and $u$ is a vertex in another tree, link the trees containing $v$ and $u$ by adding the edge between them, making $u$ the parent of $v$.
- $\mathsf{Cut}(v)$: If $v$ is not a root, this removes the edge between $v$ and its parent, so that the tree containing $v$ is divided two trees containing either $v$ or not.
- $\mathsf{Evert}(v)$: This turns the tree containing $v$ "inside out" by making $v$ the root of the tree.
- $\mathsf{Connected}(u, v)$: This checks if $u$ and $v$ are contained in the same tree.
- $\mathsf{LCA}(u, v)$: This returns the lowest common ancestor of $u$ and $v$ assuming that $u$ and $v$ are contained in the same tree.
- $\mathsf{Root}(u)$: This returns the root of the tree containing $u$.

All missing proofs and details can be found in the full version. In particular, the data structures and their update/query algorithms for $k$-PATH/CYCLE and VERTEX COVER can be found in the full version.

## 3 Dynamic Triangle Hitting Set Problem

In this section, we describe a fully dynamic data structure on the unit disk graph of a vertex set $V$ dynamically changing under vertex insertions and deletions that can answer triangle hitting set queries efficiently. Each query is given with a positive integer $k$ and asks to return a triangle hitting set of $\mathsf{UD}(V)$ of size at most $k$. This data structure will also be used for Feedback Vertex Set and Cycle Packing in Sections 4 and 5.

Our strategy is to maintain a *kernel* of $(\mathsf{UD}(V), k)$. More specifically, consider the set $V_{\mathsf{tri}}$ of vertices contained in triangles of $\mathsf{UD}(V)$. Then $(\mathsf{UD}(V_{\mathsf{tri}}), k)$ is a **yes**-instance if and only if $(\mathsf{UD}(V), k)$ is a **yes**-instance, i.e., it is a kernel of $(\mathsf{UD}(V), k)$. We can show that the size of $V_{\mathsf{tri}}$ is $O(k)$ if $(\mathsf{UD}(V), k)$ is a **yes**-instance. Therefore, it is sufficient to maintain $V_{\mathsf{tri}}$ for answering queries. However, it seems unclear if $V_{\mathsf{tri}}$ can be updated in $O(1)$ time, which is the desired update time. In particular, imagine that a vertex $v$ is inserted to $V$, and we are to determine if $v$ is contained in a triangle of $\mathsf{UD}(V)$. In the case that two neighboring cells $\square_1$ and $\square_2$ of $\square_v$ contain $\omega(k)$ vertices of $V$, we need to determine if there are vertices $x_1 \in \square_1$ and $x_2 \in \square_2$ such that $x_1, x_2$ and $v$ form a triangle of $\mathsf{UD}(V)$.

To overcome this issue, we use a superset of $V_{\mathsf{tri}}$ as a kernel. Notice that as long as a subset of $V$ contains $V_{\mathsf{tri}}$, it is a kernel of $(\mathsf{UD}(V), k)$.

**Kernel: $\boxplus_{\mathsf{core}}$ and $V_{\mathsf{core}}$.** The *core grid cluster*, denoted by $\boxplus_{\mathsf{core}}$, is defined as the union of the 5-neighboring cells of the grid cells containing a vertex of $V_{\mathsf{tri}}$ and the 10-neighboring cells of the grid cells containing at least three vertices of $V$. Then let $V_{\mathsf{core}}$ be the set of vertices of $V$ contained in $\boxplus_{\mathsf{core}}$. See Figure 3 in the full version. Note that the degree of every vertex of $V \setminus V_{\mathsf{core}}$ in $\mathsf{UD}(V)$ is $O(1)$. We will use this property for designing the update algorithm.

▶ **Lemma 1.** *The size of $V_{\mathsf{core}}$ is $O(k)$ if $\mathsf{UD}(V)$ has at most $k$ vertex-disjoint triangles.*

▶ **Observation 2.** *Given a vertex $v \in V \setminus V_{\mathsf{core}}$, we can compute its neighbors in $\mathsf{UD}(V)$ in $O(1)$ time.*

**Query algorithm.** Using Lemma 1, we only consider the case that the size of $V_{\mathsf{core}}$ is $O(k)$. Otherwise, we return **no**. Then, we can compute the minimum triangle hitting set of $\mathsf{UD}(V)$ in $2^{O(\sqrt{k})}$ time using the standard dynamic programming algorithm observed in [16]. See the full version for details.

**Update algorithm.** Suppose that we already have $V_{\mathsf{core}}$ for the current vertex set $V$. Imagine that we aim to insert a vertex $v$ to $V$. Then we need to update $V_{\mathsf{core}}$. For this, it suffices to determine if $\square$ must be contained in $\boxplus_{\mathsf{core}}$ for every 10-neighboring cell $\square$ of $\square_v$. By Observation 3, this takes $O(1)$ time. The deletion of a vertex $v$ from $V$ can be handled analogously in $O(1)$ time.

▶ **Observation 3.** *We can check if a grid cell $\square$ is contained in $\boxplus_{\mathsf{core}}$ in $O(1)$ time.*

▶ **Theorem 4.** *There is an $O(n)$-sized fully dynamic data structure on the unit disk graph induced by a vertex set $V$ supporting $O(1)$ update time that allows us to compute a triangle hitting set of size at most $k$ in $2^{O(\sqrt{k})}$ time.*

## 4 Dynamic Feedback Vertex Set Problem

In this section, we describe a fully dynamic data structure on the unit disk graph of a vertex set $V$ dynamically changing under vertex insertions and deletions that can answer feedback vertex set queries efficiently. Each query is given with a positive integer $k$ and asks to return

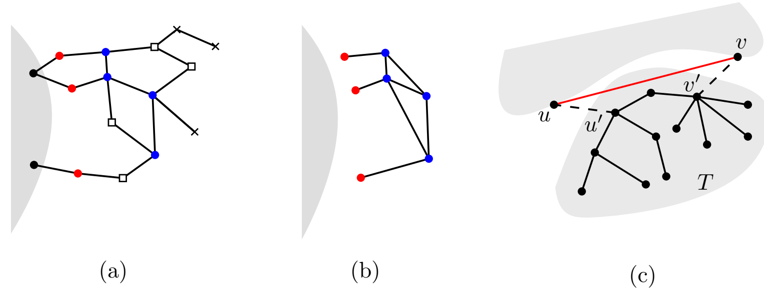


**Figure 1** (a) Illustration of $\mathsf{UD}(V)$. The vertices of $\mathsf{UD}(V)$ in $V_{\mathsf{core}}$, the boundary vertices, and the non-boundary vertices in $M$ are marked by the black, red, and blue vertices, respectively. The removed vertices and the contracted vertices in the construction of $M$ are marked by cross vertices and boxes, respectively. (b) Illustration of $M$. (c) In the construction of $M$, we are left with the induced path between $u$ and $v$, which will be contracted to the red edge in $M$. The two end edges of the path compose the bridge set of $T$.

a feedback vertex set of $\mathsf{UD}(V)$ of size at most $k$. As a data structure, we use the core grid cluster $\boxplus_{\mathsf{core}}$ introduced in Section 3. In addition to this, we design a new data structure, which will also be used for Cycle Packing in Section 5.

## 4.1 Data Structure

Note that a cycle of $\mathsf{UD}(V)$ might contain a vertex lying outside of $\boxplus_{\mathsf{core}}$. Thus we need to consider the part of $\mathsf{UD}(V)$ lying outside of $\boxplus_{\mathsf{core}}$. Since the complexity of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$ can be $\Theta(n)$ in the worst case, we cannot afford to look at all such vertices to handle a query. Instead, we maintain a minor $M$ of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$ of complexity $O(k)$, which will be called the *skeleton* of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$, such that the graph obtained by gluing $\mathsf{UD}(V_{\mathsf{core}})$ and $M$ has a feedback vertex set of size $k$ if and only if $\mathsf{UD}(V)$ has a feedback vertex set of size $k$. Then it suffices to look at $V_{\mathsf{core}}$ and $M$ to handle a query. Given an update of $V$, we need to update both $V_{\mathsf{core}}$ and $M$. Since $M$ is a minor of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$, some vertices of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$ do not appear in $M$. To handle the update of $V_{\mathsf{core}}$ and $M$ efficiently, we construct an auxiliary data structure $\mathcal{T}$, which maintains the vertices of $\mathsf{UD}(V)$ not appearing in $M$ efficiently.

**Skeleton $M$ of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$.** Given a query, we will glue $M$ and $\mathsf{UD}(V_{\mathsf{core}})$ together. For this purpose, we need to keep all vertices of $V \setminus V_{\mathsf{core}}$ adjacent to a vertex of $V_{\mathsf{core}}$ in the construction of $M$. We call such a vertex a *boundary vertex*. We let $M$ be the graph obtained from $\mathsf{UD}(V \setminus V_{\mathsf{core}})$ by removing non-boundary vertices with the degree at most one in $M$ repeatedly, and then by contracting every maximal induced path consisting of only non-boundary vertices into a single edge. Note that the resulting graph $M$ is a minor of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$. Each vertex of $M$ corresponds to a vertex of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$ of degree at least three or a boundary vertex. Furthermore, each edge of $M$ corresponds to an edge of $\mathsf{UD}(V)$ or an induced path of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$. See Figure 1(a–b). Note that $M$ is planar since every triangle-free disk graph is planar [10].

▶ **Lemma 5.** *If $(\mathsf{UD}(V), k)$ is a **yes**-instance of* Feedback Vertex Set, *$|V(M)| = O(k)$.*

**Contracted forest $\mathcal{T}$ concerning $M$.** We will see that it suffices to maintain $V_{\mathsf{core}}$ and $M$ for the query algorithm. However, to update $M$ efficiently, we need a data structure for the subgraph of $\mathsf{UD}(V)$ induced by $V \setminus (V_{\mathsf{core}} \cup V(M))$. Note that the subgraph is a forest. We

maintain the trees of this forest using the link-cut tree data structure. Let $\mathcal{T}$ denote the link-cut tree data structure. If it is clear from the context, we sometimes use $\mathcal{T}$ to denote the forest itself. Along with the link-cut trees, we associate each tree $T$ of $\mathcal{T}$ with a *bridge set*. An edge of $\mathsf{UD}(V)$ incident to both $V(M)$ and $V \setminus (V_{\mathsf{core}} \cup V(M))$ is called a *bridge*. Then the bridge set of $T$ is the set of bridges incident to $T$. See Figure 1(c).

▶ **Observation 6.** *Each tree of $\mathcal{T}$ is incident to at most two bridges.*

## 4.2 Query Algorithm

In this subsection, we show how to compute a feedback vertex set of size $k$ of $\mathsf{UD}(V)$ in $2^{O(\sqrt{k})}$ time using $V_{\mathsf{core}}$ and $M$ only. Let $G$ be the graph obtained by gluing $\mathsf{UD}(V_{\mathsf{core}})$ and $M$. More precisely, the vertex set of $G$ is the union of $V_{\mathsf{core}}$ and $V(M)$. There is an edge $uv$ in $G$ if and only if $uv$ is either an edge of $\mathsf{UD}(V_{\mathsf{core}})$, an edge of $M$, or an edge of $\mathsf{UD}(V)$ between $u \in V_{\mathsf{core}}$ and $v \in V(M)$. Notice that $v$ is a boundary vertex of $M$ in the third case.

We use the algorithm proposed by An and Oh [4]. The algorithm of [4] computes a feedback vertex set of size $k$ of a unit disk graph with $n$ vertices in $2^{O(\sqrt{k})}n^{O(1)}$ time. In our case, $G$ is not necessarily a unit disk graph. Thus we need to modify the algorithm of [4] slightly. The details of our algorithm can be found in the full version, and it concludes the following theorem.

▶ **Theorem 7.** *Given $M$ and $V_{\mathsf{core}}$, we can compute a feedback vertex set of $\mathsf{UD}(V)$ of size $k$ in $2^{O(\sqrt{k})}$ time if it exists.*

## 4.3 Update Algorithm

In this subsection, we illustrate how to update data structures $V_{\mathsf{core}}$, $M$, and $\mathcal{T}$ with respect to vertex insertions and deletions. We call $\mathsf{UD}(V \setminus V_{\mathsf{core}})$ the *shell* of $\mathsf{UD}(V)$. Here, we slightly abuse the notion of the skeleton so that we can define additional *special vertices* other than the boundary vertices. Imagine that several vertices of $V$ are predetermined as *special* vertices. The other vertices are called *ordinary* vertices. The skeleton $M$ of the shell of $\mathsf{UD}(V)$ with predetermined special vertices is defined as the minor of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$ obtained by removing all degree-1 ordinary vertices of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$ repeatedly and then contracting each maximal induced path consisting of ordinary vertices. Notice that if only the boundary vertices of $\mathsf{UD}(V)$ are set to the special vertices, the two definitions of the skeleton coincide.

Given an update of $V$, we first update $V_{\mathsf{core}}$ accordingly using the update algorithm in Section 3. Recall that the number of vertices newly added to $V_{\mathsf{core}}$ or removed from $V_{\mathsf{core}}$ is $O(1)$. We are to add the vertices removed from $V_{\mathsf{core}}$ to the shell of $\mathsf{UD}(V)$, and remove the vertices newly added to $V_{\mathsf{core}}$ from the shell of $\mathsf{UD}(V)$. Additionally, $O(1)$ vertices of $V \setminus V_{\mathsf{core}}$ become boundary vertices (when we handle the deletion operation), and $O(1)$ vertices of $V \setminus V_{\mathsf{core}}$ become non-boundary vertices (when we handle the insertion operation.) These are the only changes in the shell of $\mathsf{UD}(V)$ due to the update of $V$. Note that we just need to add $v$ to the shell of $\mathsf{UD}(V)$ if $v$ is not in $V_{\mathsf{core}}$. Let $S$ be the shell of $\mathsf{UD}(V)$ before the update. Let $M$ be the skeleton of $S$ we currently maintain, and let $\mathcal{T}$ be the contracted forest concerning $M$ we currently maintain.

In the following, we show how to update $M$ and $\mathcal{T}$ for the change of $S$. The update algorithm consists of two steps: *push-pop step* and *cleaning step*. In the push-pop step, we set several vertices of $S$ as special vertices. Specifically, the new vertices to be added to $S$ are set as special vertices. The neighbors in $S$ of the vertices to be added to $S$ or to be removed from $S$ are set to special vertices. Then we update $M$ and $\mathcal{T}$ to the skeleton of the new set

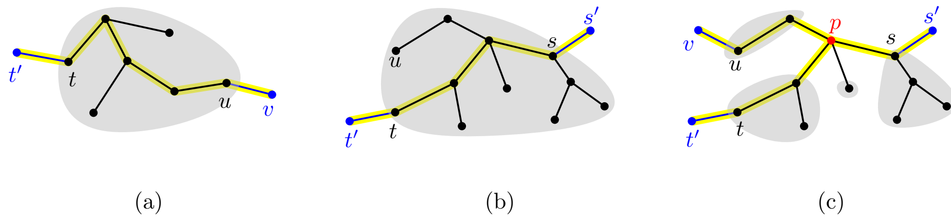


**Figure 2** (a) The case that $T$ has exactly one bridge. By the insertion of $uv$, the path from $t'$ to $v$, highlighted in yellow, is contracted. (b) The case that $T$ has exactly two bridges before inserting $uv$. (c) The same case after inserting $uv$. The vertex $p$ and the edges $pv$, $pt'$, and $ps'$ are inserted into $M$.

$S$ and the contracted forest concerning the new skeleton, respectively. In the cleaning step, we make the non-boundary vertices ordinary vertices. Note that this changes the structure of the skeleton as well, and thus we need to update $M$ and $\mathcal{T}$.

### 4.3.1 Push-Pop Step

Recall that $S$ is the shell of $\mathsf{UD}(V)$ before the update. Notice that the complexity of the new shell of $\mathsf{UD}(V)$ can be $\Theta(n)$ in the worst case. However, the update algorithm in Section 3 determines the vertices and edges added to $S$ and removed from $S$ to obtain the new shell. By construction, the number of such vertices and edges is $O(1)$. Moreover, we can determine the vertices set to the special vertices of $S$ in $O(1)$ time by Observation 2. To update $M$ and $\mathcal{T}$, we add (or remove) the special vertices and their incident edges one by one to (or from) $S$ until $S$ becomes the desired set. Precisely, we add each special vertex $v$ using the *push* subroutine, which adds $v$ into $M$ and updates $M$ and $\mathcal{T}$ accordingly. In the push subroutine, we assume that $v$ was not contained in $S$. Recall that $M$ is the skeleton of $S$ obtained by removing and contracting some ordinary vertices, not special vertices. And, we remove $v$ using the *pop* subroutine, which removes $v$ from $S$ and update $M$ and $\mathcal{T}$. Note that some special vertices $v$ are already in $S$. In this case, we cannot make $v$ special using the push subroutine as it assumes that $v$ is not contained in $S$. For such special vertices $v$, we first pop $v$ from $S$ and then push it back to $S$. In this way, we can obtain the skeleton of the new shell and the contracted forest correctly.

**Push subroutine.** Given the current shell $S$, we are to add $v$ and its incident edges to $S$. Let $E_v$ be the set of edges incident to $v$ to be added to $S$. We can ensure that the other endpoints of the edges of $E_v$ are in $S$. We first add $v$ and the edges of $E_v$ incident to vertices of $M$ into $S$. At this moment, it suffices to add $v$ and these edges to $M$. We do not need to update $\mathcal{T}$. After that, we insert the remaining edges of $E_v$ into $S$ one by one as follows. Note that those edges are incident to vertices of $\mathcal{T}$. Let $T$ be the tree in $\mathcal{T}$ containing another endpoint $u$ of an edge of $E_v$. Before the insertion of $uv$, $T$ has its bridge set. There are three cases: $T$ has either exactly zero, one, or two bridges. In particular, in the case that $T$ has exactly one or two bridges we modify $M$. If $T$ has exactly one bridge, we add an edge into $M$. And, if $T$ has exactly two bridges, we delete a contracted edge corresponding to an induced path in $T$ and add one vertex and three edges into $M$. See Figure 2. The details of the procedure for inserting an edge $uv$ of $E_v$ with $u \in \mathcal{T}$ into $S$ are described in the full version.

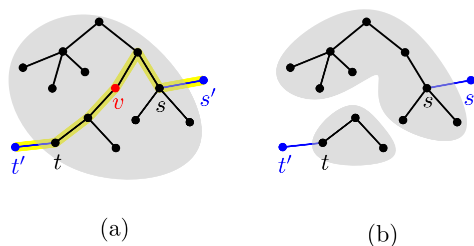


**Figure 3** (a) Illustration of pop subroutine in the case that the $t$-$s$ path contains $v$. (b) Illustration of the tree $T$ after $v$ is deleted.

**Pop subroutine.** We are to delete a vertex $v$ and all of its incident edges in $S$ from $S$. We consider two cases separately: $v$ is in $M$ or $\mathcal{T}$. For the case that $v$ is in $M$, we simply remove it and its incident edges from $M$ and the bridge sets of $\mathcal{T}$. Specifically, for each edge $e$ incident to $v$ in $S$, we remove it from $M$ if it is in $M$. If it is not in $M$, it is in a bridge set of a tree of $\mathcal{T}$. We remove it from every bridge set. If a bridge set containing $e$ has another edge, then there is a contracted edge of $M$, and thus we remove it. Then we are done.

Now consider the case that $v$ is in a tree $T$ of $\mathcal{T}$. In this case, the deletion of $v$ from $S$ changes $M$ if and only if $T$ has two bridges $tt'$, $ss'$ with $t, s \in V(T)$ and $t', s' \in V(M)$ such that the $t$-$s$ path in $T$ contains $v$. In particular, $M$ has the edge $t's'$, and $t's'$ must be removed from $M$ by the deletion of $v$ from $S$. See Figure 3. We can check if the $t$-$s$ path in $T$ contains $v$ by applying $\mathsf{Evert}(v)$ and $\mathsf{LCA}(t, s)$. Then we are to remove $v$ and it incident edges from $T$ and the bridge set of $T$. Observe that an edge incident to $v$ is in $T$ or the bridge set of $T$. We rotate $T$ in a way that $v$ becomes the root of $T$ by using $\mathsf{Evert}(v)$, and remove $v$ from $T$ by applying $\mathsf{Cut}(\cdot)$ operations. Then we are given a constant number of child subtrees of $v$ since $v$ is in $V \setminus V_{\mathsf{core}}$. For a child subtree $T'$ of $T$ at $v$, we insert the bridges of $T$ incident to $T'$ into the bridge set of $T'$. Note that, given $T'$ and a bridge of $T$, we can check if $T'$ contains a vertex incident to the bridge using $\mathsf{Connected}(\cdot)$. In this way, we can remove $v$ from $\mathcal{T}$ and the bridge sets, and we can update $M$ accordingly in $O(\log |V|)$ time.

### 4.3.2 Cleaning Step

So far, we have treated all vertices that can cause some changes due to the update of $V$ and special vertices, and we have computed the skeleton of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$. However, some special vertices should not be considered as special vertices if they are not boundary vertices. To handle this, we need a *cleaning process* to maintain the degree of every vertex in $M$ is at least three except the boundary vertices.

We handle the vertices of $M$ of degree at most two one by one and set each of them as an ordinary vertex if it is a special vertex, as follows. Let $v$ be a vertex we are to handle. First, we can check in $O(1)$ time if $v$ is a boundary vertex using Observation 2. If $v$ is a boundary vertex, we are done. Otherwise, it suffices to handle the case that the degree of $v$ in $M$ is less than three. The update of $M$ is simple: we remove $v$ from $M$ if its degree is one in $M$, and contract a maximal induced path containing $v$ in $M$ if the degree of $v$ is two in $M$. Then we need to update $\mathcal{T}$ accordingly as follows. It suffices to merge all trees in $\mathcal{T}$ incident to $v$ together with their bridges incident to $v$ using $\mathsf{Link}(\cdot)$ sequentially. Then the bridge set of the resulting tree is the union of all bridges of merged trees except the ones incident to $v$. We can handle $v$ in $O(\log |V|)$ time. However, this process might decrease the degree of some other vertex of $M$ of degree in $M$ to less than three. For each such vertex, we remove it or contract a maximal induced path containing it as we did for $v$.

▶ **Lemma 8.** *The cleaning step takes $O(\log |V|)$ time in total.*

Since both steps can be done in $O(\log |V|)$ time, we have the following lemma.

▶ **Lemma 9.** *Given a vertex update of $V$, we can update $V_{\mathsf{core}}$, $M$ and $\mathcal{T}$ in $O(\log |V|)$ time.*

▶ **Theorem 10.** *There is an $O(n)$-sized fully dynamic data structure on the unit disk graph induced by a vertex set $V$ supporting $O(\log |V|)$ update time that allows us to compute a feedback vertex set of size at most $k$ in $2^{O(\sqrt{k})}$ time.*

## 5 Dynamic Cycle Packing Problem

In this section, we describe a fully dynamic data structure on the unit disk graph of a vertex set $V$ dynamically changing under vertex insertions and deletions that can answer cycle packing queries efficiently. Each query is given with a positive integer $k$ and asks to return a set of $k$ vertex-disjoint cycles of $\mathsf{UD}(V)$ if it exists. Here, we use the core grid cluster $\boxplus_{\mathsf{core}}$, the set $V_{\mathsf{core}}$ of vertices contained in $\boxplus_{\mathsf{core}}$, the skeleton $M$ of $\mathsf{UD}(V \setminus V_{\mathsf{core}})$, and the contracted forest $\mathcal{T}$ along with the bridge sets. They can be maintained in $O(\log |V|)$ time as shown in Section 4.3. Note that $\mathcal{T}$ and the bridges are the auxiliary data structures for updating $M$ efficiently. In this section, we present a query algorithm for Cycle Packing assuming that we have $\boxplus_{\mathsf{core}}$, $V_{\mathsf{core}}$, and $M$. The following lemma was given by An and Oh [6].

▶ **Lemma 11.** *If $(\mathsf{UD}(V), k)$ is a **no**-instance for Cycle Packing, then $|V(M)| = O(k)$.*

As in Section 4.2, we first compute the graph $G$ by gluing $\mathsf{UD}(V_{\mathsf{core}})$ and $M$. More precisely, the vertex set of $G$ is the union of $V_{\mathsf{core}}$ and $V(M)$. There is an edge $uv$ in $G$ if and only if $uv$ is either an edge of $\mathsf{UD}(V_{\mathsf{core}})$, an edge of $M$, or an edge of $\mathsf{UD}(V)$ between $u \in V_{\mathsf{core}}$ and $v \in V(M)$. Notice that $v$ is a boundary vertex of $M$ in the third case.

We use the algorithm proposed by An and Oh [6]. The algorithm of [6] computes a cycle packing of size $k$ of a unit disk graph with $n$ vertices in $2^{O(\sqrt{k})}n^{O(1)}$ time. This algorithm uses the geometric representation of a given graph, and thus the drawing of $G$ is needed. Specifically, we need a drawing of $G$ such that every vertex of $G$ is on its geometric representation and every edge of $M$ does not intersect $\boxplus_{\mathsf{core}}$. For the desired running time, we consider a drawing of $G$ of complexity $\mathrm{poly}(k)$. Since the number of vertices of $V_{\mathsf{core}}$ is $O(k)$, it suffices to draw $M$ and the edges between $V(M)$ and $V_{\mathsf{core}}$ properly. In Section 5.1, we will see that we can compute a desired drawing of such subgraph of $G$ of complexity $\mathrm{poly}(k)$ in $\mathrm{poly}(k)$ time. Then, we can draw $G$ by merging this drawing with the drawing of $G \setminus M$. Furthermore, in our case, $G$ is not necessarily a unit disk graph. Thus we need to modify the algorithm of [4] slightly. The details of our algorithm can be found in the full version, and it concludes the following theorem.

▶ **Theorem 12.** *Given the core grid cluster $\boxplus_{\mathsf{core}}$ and the skeleton $M$, Cycle Packing can be solved in $2^{O(\sqrt{k})}$ time.*

### 5.1 Planar Drawing of the Closure of $M$

In this subsection, we illustrate how to compute a planar drawing of the closure of $M$ into the plane such that the vertices are drawn on their corresponding points in $V$, and the drawing does not intersect the boundary of $\boxplus_{\mathsf{core}}$. Here, it suffices to prove the following lemma. In our case, each vertex of the closure of $M$ is prespecified, and each connected region of $\mathbb{R}^2 \setminus \boxplus_{\mathsf{core}}$ is a polygonal domain $\Sigma$. By applying the following lemma for each connected region of $\mathbb{R}^2 \setminus \boxplus_{\mathsf{core}}$, we can compute a planar drawing of the closure of $M$ of complexity $\mathrm{poly}(k)$ with the desired properties in $\mathrm{poly}(k)$ time.

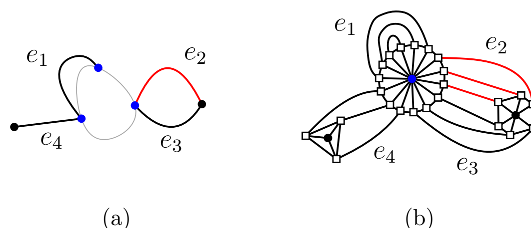


**Figure 4** (a) The blue vertices denote the vertices of $L$ on the boundary of a hole. (b) The vertices on a hole are contracted into one blue vertex on the boundary of the hole. And, the black boxes denote the vertices of gadgets. Each edge is copied into three edges. For clarity, the edges corresponding to $e_2$ are colored in red.

▶ **Lemma 13.** *Any planar graph $L$ admits a planar drawing on a polygonal domain $\Sigma$ that maps each vertex to its prespecified location and each edge to a polygonal curve with $O(|\Sigma| \cdot |E(L)|^3)$ bends, where $|\Sigma|$ denotes the total complexity of boundaries of $\Sigma$. Moreover, we can draw such one in $O(|\Sigma| \cdot |E(L)|^4)$ time if we know the proper ordering of edges incident to each vertex.*

We demonstrate Lemma 13 by Witney's theorem [36, 35] along with the algorithm in [32]. Specifically, Witney's theorem guarantees that a *3-connected* planar graph admits a topologically unique planar embedding.

For clarity, we assume that $L$ is connected, and we demonstrate how to draw a planar drawing of $L$ that maps each vertex to its prespecified location and each edge to a polygonal curve with $O(|\Sigma| \cdot |E(L)|)$ bends in $O(|\Sigma| \cdot |E(L)^2|)$ time. When $L$ is not connected, we can draw the desired planar drawing of $L$ by drawing each component of $L$ using the aforementioned algorithm, sequentially. Precisely, after we compute a drawing of a connected component of $L$, we add the region containing the drawing into $\Sigma$ as a hole. We can compute such region in time $O(|\Sigma| \cdot |E(L)|^2)$ time by unifying all faces of the drawing except the outer face. Note that the total complexity of the boundaries of such regions is $O(|\Sigma| \cdot |E(L)|^2)$. Thus we can compute the desired planar drawing of $L$ in $O(|\Sigma| \cdot |E(L)|^4)$ time.

▶ **Lemma 14** (Theorem 1 in [32]). *Every planar graph $L$ admits a planar drawing that maps each vertex to an arbitrarily prespecified distinct location and each edge to a polygonal curve with $O(|V(L)|)$ bends. Moreover, such a drawing can be constructed in $O(|V(L)|^2)$ time.*

We first contract each hole in $\Sigma$ into a single point and compute the planar drawing of $L$ in a plane without holes using Lemma 14. After that, we recover the holes in $L$ at the prespecified locations. In the recovering step, we want to ensure that the given topological ordering is maintained in the planar drawing of $L$. To do this, we use Witney's theorem. However, $L$ is not necessarily 3-connected. For this reason, we slightly modify $L$ so that it becomes 3-connected.

**Modification for $L_{\mathsf{tc}}$ and drawing $\mathcal{D}_{\mathsf{tc}}$.** To utilize Witney's theorem and Lemma 14, we obtain a *3-connected* planar graph $L_{\mathsf{tc}}$ by modifying $L$ with respect to the holes in the polygonal domain $\Sigma$, and we compute a polygonal drawing $\mathcal{D}_{\mathsf{tc}}$ of $L_{\mathsf{tc}}$ using Lemma 14. We assume that each vertex in $L$ is the distance at least $0 < \varepsilon < 1$ from any other vertex in $L$.

We first contract each hole into a single point in the plane. In this way, all vertices of $L$ lying on the boundary of the same hole or the outer boundary of $\Sigma$ are contracted to a single vertex on the boundary accordingly. Next, we add a *wheel graph* centered on $v$ for each vertex $v$ of $L$ as a gadget. Specifically, the gadget is formed by connecting a single

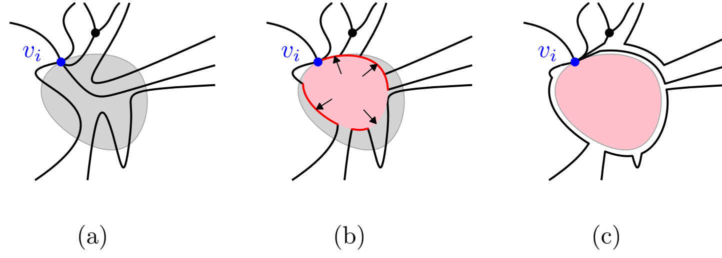


**Figure 5** (a) A part of $D_{\mathsf{tc}}$ after removing paths. The gray region is a hole $A_i$, and the blue vertex is $v_i$. and (b) The pink region is $A'_i$. While blowing $A'_i$, we push subcurves on its boundary as the red curves if they intersect $A_i$. (c) After blowing up $A'_i$, we perturb the drawing of $\mathcal{D}$ without crossing.

universal vertex to the vertices of a cycle with $3d_v$ vertices of diameter $\varepsilon/100$, where $d_v$ is the degree of $v$ in $L$. Then we replace each edge $uv$ of $L$ with three edges: we choose three consecutive vertices from the gadget for $u$ and three consecutive vertices from the gadget for $v$, then we connect them. See Figure 4. We can do this for all edges of $L$ without crossing by maintaining the proper ordering of the edges around each vertex of $L$. Recall that we have the proper ordering of incident edges at each vertex in $L$. We denote the result graph as $L_{\mathsf{tc}}$. Note that $L_{\mathsf{tc}}$ is a 3-connected planar graph since a wheel graph with at least four vertices is 3-connected.

The number of vertices in $L_{\mathsf{tc}}$ is at most $6|E(H)| + |V(H)|$. By Lemma 14, we can compute a planar drawing of $L_{\mathsf{tc}}$ in the plane where each edge has at most $O(|E(H)|)$ bends. Furthermore, it is the unique topological embedding by Witney's theorem since $L_{\mathsf{tc}}$ is a 3-connected planar graph. We denote the drawing by $\mathcal{D}_{\mathsf{tc}}$. In the following, we recover a polygonal drawing $\mathcal{D}$ of $L$ in $\Sigma$.

**Recovering $\mathcal{D}$ from $\mathcal{D}_{\mathsf{tc}}$.** We recover a drawing $\mathcal{D}$ of $L$ in $\Sigma$ from $\mathcal{D}_{\mathsf{tc}}$. For each edge $uv$ in $L$, it corresponds to three paths of length three in $L_{\mathsf{tc}}$ connecting the universal vertices of the gadgets for $u$ and $v$. Among them, except the middle one, we remove two paths from $\mathcal{D}_{\mathsf{tc}}$. While keeping the drawing of $\mathcal{D}_{\mathsf{tc}}$ of the remaining path between the universal vertices for $u$ and $v$ as the drawing of $uv$ of $L$, we remove the vertices in $L_{\mathsf{tc}}$ which are not in $L$. This process increases the number of bends of each edge by a factor of at least three compared to $\mathcal{D}_{\mathsf{tc}}$. We refer to the obtained drawing as $\mathcal{D}$.

In the following, we recover the boundaries of $\Sigma$. Let $A_1, A_2, \ldots,$ and $A_\ell$ be holes of $\Sigma$, Note that the vertices of $L$ on the boundary of $A_i$ are contracted into one vertex in $L_{\mathsf{tc}}$. We refer to the contracted vertex as $v_i$ for each $A_i$.

For each $A_i$, we modify $\mathcal{D}$ so that the resulting drawing avoids $A_i$. Precisely, we blow up a region $A'_i$, which is initially the point $v_i$, within a face adjacent to $v_i$ until it becomes $A_i$. While blowing up $A'_i$, we push the subcurves of the curves of $\mathcal{D}$ which intersect $A'_i$ onto the boundary of $A'_i$. Notice that we push such subcurves, avoiding $v_i$, except the boundary curves of the face containing $A'_i$. See Figure 5(b). By repeating such a process, we make the interior of $A_i$ empty. Then, by perturbing the drawing $\mathcal{D}$, we can modify the drawing to avoid $A_i$ and crossing. See Figure 5(c). This process increases the bends by a factor of $|\Sigma|$ compared to $\mathcal{D}_{\mathsf{tc}}$.

In the following, we uncontract the vertices $v_i$'s for each $A_i$'s. Let $\ell$ and $\ell'$ be short line segments incident to $v_i$ drawn in opposite directions within the face containing $A_i$ so that they intersect $\mathcal{D}$ and $A_i$ only at $v_i$. See Figure 6(b). Note that the edges in $\mathcal{D}$ incident

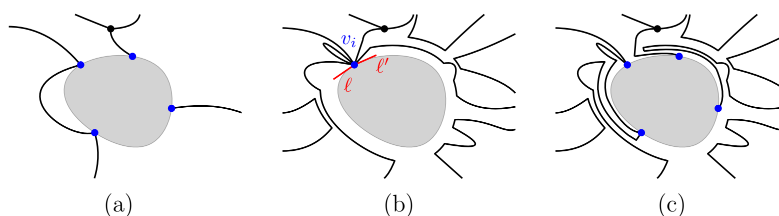


**Figure 6** (a) A part of $L$. The gray region is a hole $A_i$, and the blue vertices are the vertices of $L$ on the boundary of $A_i$. (b) A part of drawing $D$ after making $A_i$ empty. The blue vertex is $v_i$, and the red lines are $l$ and $l'$, respectively. (c) We uncontract $v_i$ into the four blue vertices of $L$.

to $v_i$ have an endpoint on the boundary of $A_i$ in $L$, and the ordering is the same as the prespecified ordering along the boundary of $A_i$ in $L$ due to Witney's theorem. We choose the closest incident edge $e$ of $v_i$ to the boundary of $A_i$, and we uncontract an endpoint $u$ of the edge $e'$ in $L$ corresponding to $e$ contracted into $v_i$. Precisely, when there is no incident edge of $v_i$ between $e$ and $\ell$ (or $\ell'$), we extend the drawing of $e$ along the boundary of $A_i$ in the counterclockwise direction (or clockwise direction) until $u$ is located at its prespecified location. If both endpoints of $e'$ are contracted into $v_i$, we uncontract the endpoint preceding in the clockwise order (or counterclockwise order). By repeating such a process, we uncontract all the vertices in $L$ on the boundary $A_i$ at the prespecified location. See Figure 6(c). This process increases the bends of each edge by a factor of at most $|\Sigma|$.

In conclusion, the obtained $\mathcal{D}$ is a polygonal drawing of $L$ in $\Sigma$ each of which edge has at most $O(|\Sigma| \cdot |E(H)|)$ bends, where $|\Sigma|$ denotes the complexity of the boundary of $\Sigma$. Furthermore, the above processes take $O(|\Sigma| \cdot |E(H)|^2)$ time in total. This completes the proof of Lemma 13.

## 6 Conclusion

In this paper, we initiate the study of fundamental parameterized problems for dynamic unit disk graphs. including $k$-Path/Cycle, Vertex Cover, Triangle Hitting Set, Feedback Vertex Set, and Cycle Packing. Our data structure supports $2^{O(\sqrt{k})}$ update time and $O(k)$ query time for $k$-Path/Cycle. For the other problems, our data structures support $O(\log n)$ update time and $2^{O(\sqrt{k})}$ query time, where $k$ denotes the output size. Despite the progress made in this work, there remain numerous open problems. First, can we obtain a trade-off between query times and update times? Second, one might consider other classes of geometric intersection graphs in the dynamic setting such as disk graphs [3, 30], outerstring graphs [7], transmission graphs [5] and hyperbolic unit disk graphs [31]. To the best of our knowledge, there have been no known results on parameterized algorithms for those graph classes.

## References

1 Pankaj Agarwal, Hsien-Chih Chang, Subhash Suri, Allen Xiao, and Jie Xue. Dynamic geometric set cover and hitting set. *ACM Transactions on Algorithms (TALG)*, 18(4):1–37, 2022. doi:10.1145/3551639.

2 Josh Alman, Matthias Mnich, and Virginia Vassilevska Williams. Dynamic parameterized problems and algorithms. *ACM Transactions on Algorithms (TALG)*, 16(4):1–46, 2020. doi:10.1145/3395037.

3. Shinwoo An, Kyungjin Cho, and Eunjin Oh. Faster algorithms for cycle hitting problems on disk graphs. In *Proceedings of the 18th Algorithms and Data Structures Symposium (WADS 2023)*, pages 29–42, 2023. doi:10.1007/978-3-031-38906-1_3.
4. Shinwoo An and Eunjin Oh. Feedback vertex set on geometric intersection graphs. In *Proceedings of the 32nd International Symposium on Algorithms and Computation (ISAAC 2021)*, pages 47:1–47:12, 2021. doi:10.4230/LIPICS.ISAAC.2021.47.
5. Shinwoo An and Eunjin Oh. Reachability problems for transmission graphs. *Algorithmica*, 84(10):2820–2841, 2022. doi:10.1007/S00453-022-00985-1.
6. Shinwoo An and Eunjin Oh. ETH-tight algorithm for cycle packing on unit disk graphs. In *Proceedings of the 40th International Symposium on Computational Geometry (SoCG 2024)*, pages 7:1–7:15, 2024. doi:10.4230/LIPICS.SOCG.2024.7.
7. Shinwoo An, Eunjin Oh, and Jie Xue. Sparse outerstring graphs have logarithmic treewidth. In *32nd Annual European Symposium on Algorithms (ESA 2024)*, pages 10:1–10:18, 2024. doi:10.4230/LIPICS.ESA.2024.10.
8. Sujoy Bhore and Timothy M. Chan. Fully dynamic geometric vertex cover and matching. *arXiv preprint*, 2024. doi:10.48550/arXiv.2402.07441.
9. Sujoy Bhore, Martin Nöllenburg, Csaba D. Tóth, and Jules Wulms. Fully dynamic maximum independent sets of disks in polylogarithmic update time. In *Proceedings of the 40th International Symposium on Computational Geometry (SoCG 2024)*, pages 19:1–19:16, 2024. doi:10.4230/LIPICS.SOCG.2024.19.
10. Heinz Breu. *Algorithmic aspects of constrained unit disk graphs.* PhD thesis, University of British Columbia, 1996.
11. Timothy M. Chan and Zhengcheng Huang. Dynamic geometric connectivity in the plane with constant query time. In *Proceedings of the 40th International Symposium on Computational Geometry (SoCG 2024)*, pages 36:1–36:13, 2024. doi:10.4230/LIPICS.SOCG.2024.36.
12. Timothy M. Chan, Mihai Pătraşcu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. *SIAM Journal on Computing*, 40(2):333–349, 2011. doi:10.1137/090751670.
13. Jiehua Chen, Wojciech Czerwiński, Yann Disser, Andreas Emil Feldmann, Danny Hermelin, Wojciech Nadara, Marcin Pilipczuk, Michał Pilipczuk, Manuel Sorge, Bartłomiej Wróblewski, et al. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. In *Proceedings of the 32th ACM-SIAM Symposium on Discrete Algorithms (SODA 2021)*, pages 796–809. SIAM, 2021.
14. Brent N. Clark, Charles J. Colbourn, and David S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86(1-3):165–177, 1990. doi:10.1016/0012-365X(90)90358-O.
15. Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 5. Springer, 2015. doi:10.1007/978-3-319-21275-3.
16. Mark de Berg, Hans L. Bodlaender, Sándor Kisfaludi-Bak, Dániel Marx, and Tom C. van der Zanden. A framework for exponential-time-hypothesis–tight algorithms and lower bounds in geometric intersection graphs. *SIAM Journal on Computing*, 49(6):1291–1331, 2020. doi:10.1137/20M1320870.
17. Erik D. Demaine, Fedor V. Fomin, Mohammadtaghi Hajiaghayi, and Dimitrios M. Thilikos. Subexponential parameterized algorithms on bounded-genus graphs and $H$-minor-free graphs. *Journal of the ACM (JACM)*, 52(6):866–893, 2005. doi:10.1145/1101821.1101823.
18. Zdeněk Dvořák, Martin Kupec, and Vojtěch Tůma. A dynamic data structure for MSO properties in graphs with bounded tree-depth. In *Proceedings of the 22th Annual European Symposium (ESA 2014)*, pages 334–345, 2014.
19. Thomas Erlebach, Klaus Jansen, and Eike Seidel. Polynomial-time approximation schemes for geometric intersection graphs. *SIAM Journal on Computing*, 34(6):1302–1323, 2005. doi:10.1137/S0097539702402676.

20 Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, Saket Saurabh, and Meirav Zehavi. Finding, hitting and packing cycles in subexponential time on unit disk graphs. *Discrete & Computational Geometry*, 62:879–911, 2019. doi:10.1007/S00454-018-00054-X.

21 Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, Saket Saurabh, and Meirav Zehavi. ETH-tight algorithms for long path and cycle on unit disk graphs. *Journal of Computational Geometry*, 12(2):126–148, 2021. doi:10.20382/JOCG.V12I2A6.

22 Fedor V Fomin, Daniel Lokshtanov, and Saket Saurabh. Bidimensionality and geometric graphs. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, pages 1563–1575, 2012. doi:10.1137/1.9781611973099.124.

23 Timothy F. Havel, Gordon M. Crippen, Irwin D. Kuntz, and Jeffrey M. Blaney. The combinatorial distance geometry method for the calculation of molecular conformation ii. sample problems and computational statistics. *Journal of Theoretical Biology*, 104(3):383–400, 1983.

24 Dorit S. Hochbaum and Wolfgang Maass. Approximation schemes for covering and packing problems in image processing and VLSI. *Journal of the ACM (JACM)*, 32(1):130–136, 1985. doi:10.1145/2455.214106.

25 Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, 1982. doi:10.1137/0211056.

26 Tuukka Korhonen, Konrad Majewski, Wojciech Nadara, Michał Pilipczuk, and Marek Sokołowski. Dynamic treewidth. In *Proceedings of the 64th Annual Symposium on Foundations of Computer Science (FOCS 2023)*, pages 1734–1744, 2023.

27 Tomasz Krawczyk and Bartosz Walczak. On-line approach to off-line coloring problems on graphs with geometric representations. *Combinatorica*, 37(6):1139–1179, 2017. doi:10.1007/S00493-016-3414-X.

28 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Unit disk graph approximation. In *Proceedings of the 2004 joint workshop on Foundations of mobile computing*, pages 17–23, 2004. doi:10.1145/1022630.1022634.

29 Fabian Kuhn, Roger Wattenhofer, and Aaron Zollinger. Ad-hoc networks beyond unit disk graphs. In *Proceedings of the 2003 joint workshop on Foundations of mobile computing*, pages 69–78, 2003.

30 Daniel Lokshtanov, Fahad Panolan, Saket Saurabh, Jie Xue, and Meirav Zehavi. Subexponential parameterized algorithms on disk graphs (extended abstract)*. In *Proceedings of the 33rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)*, pages 2005–2031, 2022. doi:10.1137/1.9781611977073.80.

31 Eunjin Oh and Seunghyeok Oh. Algorithms for Computing Maximum Cliques in Hyperbolic Random Graphs. In *31st Annual European Symposium on Algorithms (ESA 2023)*, pages 85:1–85:15, 2023. doi:10.4230/LIPICS.ESA.2023.85.

32 János Pach and Rephael Wenger. Embedding planar graphs at fixed vertex locations. *Graphs and Combinatorics*, 17:717–728, 2001. doi:10.1007/PL00007258.

33 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC 1981)*, pages 114–122, 1981.

34 Da-Wei Wang and Yue-Sun Kuo. A study on two geometric location problems. *Information processing letters*, 28(6):281–286, 1988. doi:10.1016/0020-0190(88)90174-3.

35 Hassler Whitney. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54(1):150–168, 1932.

36 Hassler Whitney. 2-isomorphic graphs. *American Journal of Mathematics*, 55(1):245–254, 1933.

37 Weili Wu, Hongwei Du, Xiaohua Jia, Yingshu Li, and Scott C.-H. Huang. Minimum connected dominating sets and maximal independent sets in unit disk graphs. *Theoretical Computer Science*, 352(1-3):1–7, 2006. doi:10.1016/J.TCS.2005.08.037.