Dynamic Subgraph Connectivity with Geometric Applications*

Timothy M. Chan†

April 18, 2006

Abstract

Inspired by dynamic connectivity applications in computational geometry, we consider a problem we call *dynamic subgraph connectivity*: design a data structure for an undirected graph $G = (V, E)$ and a subset of vertices $S \subseteq V$, to support insertions and deletions in S and connectivity queries (are two vertices connected?) in the subgraph induced by S. We develop the first sublinear, fully dynamic method for this problem for general sparse graphs, using a combination of several simple ideas. Our method requires $\widetilde{O}(|E|^{4\omega/(3\omega+3)}) = O(|E|^{0.94})$ amortized update time, and $\widetilde{O}(|E|^{1/3})$ query time, after $\widetilde{O}(|E|^{(5\omega+1)/(3\omega+3)})$ preprocessing time, where ω is the matrix multiplication exponent and $\widetilde{O}$ hides polylogarithmic factors.

Key words. Data structures, dynamic graph algorithms, connectivity, computational geometry

AMS subject classifications. 68Q25, 68P05, 68U05

Abbreviated title. Dynamic subgraph connectivity

1 Introduction

Geometric motivation. *Dynamic graph connectivity*—maintaining an undirected graph under edge insertions and deletions, to answer queries of the form, "are two vertices connected?"—is a basic problem in the area of graph data structures. In the same way, connectivity problems concerning a dynamic collection of geometric objects are fundamental in computational geometry. However, unlike dynamic graph connectivity, which has been extensively studied and has enjoyed much recent success with the discovery of near-logarithmic algorithms [17, 21, 32], progress in *dynamic geometric connectivity* has been scarce. For this reason, we decide to start our investigation with a simple version of the problem:

> Maintain a set of n axis-parallel rectangles in the plane, under insertions and deletions, to answer queries of the form, "given two points a and b, is there a path from a to b that lies inside the union of these rectangles?"

Rectangular connectivity queries have numerous applications, for example, in VLSI design, communication networks, and geographic information systems. (See Figure 1.) Solving this problem might pave the way for the study of dynamic connectivity of other objects, and perhaps, tougher questions like dynamic shortest paths and motion planning.

*A preliminary version of this paper appeared in *Proc. 34th ACM Sympos. Theory Comput.*, pages 7–13, 2002. This work was supported in part by an NSERC Research Grant.

†School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada (tmchan@uwaterloo.ca).

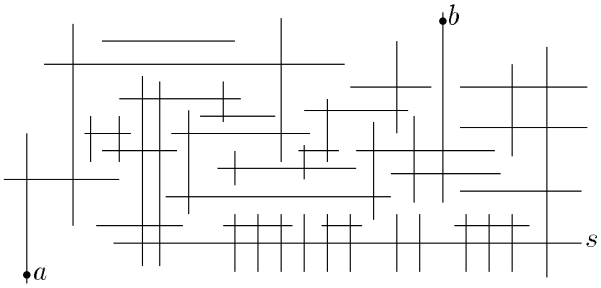

Figure 1: A collection of "roads". Is b reachable from a?

No fully dynamic solution for rectangular connectivity has been reported even for the special case of orthogonal line segments. Existing geometric data structuring techniques (notably, range searching) [2, 29] seem insufficient, because connectivity queries are more global in nature. On the other hand, although our problem is equivalent to dynamic connectivity in the intersection graph (where we place an edge between every pair of intersecting rectangles), a straightforward application of dynamic graph connectivity results would not lead to an efficient solution either, because the intersection graph can have quadratic size, and an insertion/deletion of a rectangle can cause a linear number of edge updates in the worst case. (Imagine deleting and re-inserting a segment like s in Figure 1.) Thus, work on dynamic graph connectivity [13, 15, 18, 17, 21, 31, 32] is only the beginning, if we want to tackle the more challenging dynamic connectivity problems from geometry.

In this paper, we show that a sublinear time bound is indeed theoretically possible for dynamic connectivity of rectangles, and in fact, axis-parallel boxes in any fixed dimension d. Specifically, we achieve $\widetilde{O}(n^{4\omega/(3\omega+3)})$ amortized time for insertions and deletions, and $\widetilde{O}(n^{1/3})$ time for queries, using $\widetilde{O}(n)$ space. Here and throughout the paper, the $\widetilde{O}$ notation hides polylogarithmic factors in n, and ω denotes the matrix multiplication exponent. The current best matrix multiplication result with $\omega < 2.376$ by Coppersmith and Winograd [7] implies an $O(n^{0.94})$ upper bound for updates, but any subcubic method with $\omega < 3$ (such as Strassen's) would already imply a sublinear upper bound.

This result is striking in two respects: (i) the independence of the exponent of our time bounds on the dimension d, and (ii) the usage of fast matrix multiplication, which is rare among algorithms in computational geometry. (Applications of fast matrix multiplication are more common in dynamic graph algorithms, e.g., [9, 24], but the sublinearity of our update bound is still unusual.) In Section 7, we partially explain why a polylogarithmic solution is unlikely given the current state of the art, and why points (i) and (ii) might be inherent to the problem itself, at least for $d \geq 3$.

Previous geometric work. Statically, connectivity for n rectangles in the plane can be decided in $O(n \log n)$ time [23]. Agarwal and van Kreveld [3] gave an approach for the *incremental* (insertion-only) case that, in particular, yielded an $O(\log^2 n)$ amortized time bound for orthogonal segments.

For unit squares and unit hypercubes (and thus for near-equal-size boxes of bounded aspect ratios), a fully dynamic, polylogarithmic method can be obtained by an easy reduction to the maintenance of the L_∞-minimum spanning tree of a point set, which was solved by Eppstein [12] in any fixed dimension (using known dynamic minimum spanning tree results for graphs [21]). For arbitrary rectangles and arbitrary hypercubes, Hershberger and Suri [19, 20] considered the *kinetic* problem of maintaining connectivity as objects move continuously according to given flight plans: allowing a quadratic number of events, we can easily reduce the kinetic problem to dynamic graph connectivity using the intersection graph; Hershberger and Suri showed that the space complexity can be decreased from quadratic to linear while still supporting efficient updating.

The dynamic subgraph connectivity problem. As we have pointed out, the intersection graph cannot directly be used for dynamic rectangular connectivity. However, existing geometric range searching results allow us to compress the intersection graph, using so-called "biclique covers" [1, 14]. Although this technique is familiar, it has not been used in connectivity applications, and in our opinion, leads to a conceptually cleaner description of geometric connectivity algorithms because of the separation of geometric and non-geometric elements.

As it turns out (see Section 2), the problem on the compressed intersection graph leads to an interesting generalization of dynamic graph connectivity, which is not as well-studied but we believe is equally fundamental:

> Maintain an undirected graph $G = (V, E)$ with n vertices and m edges, and a subset $S \subseteq V$ of vertices, under the following operations: insert an edge to E, delete an edge from E, insert a vertex to S, and delete a vertex from S. Queries to be answered are of the form, "given two vertices $u, v \in S$, is there a path from u to v that uses only vertices from S?"

We call this problem *dynamic subgraph connectivity.* The difficulty lies not in edge updates to E, but in vertex updates to the subset S. If each vertex is inserted and deleted only once, then we can directly apply data structures for dynamic graph connectivity to maintain the subgraph induced by S; the amortized cost would be near-logarithmic per edge, since each edge is inserted and deleted $O(1)$ times. In particular, we can therefore handle the *incremental* (insertion-only) or *decremental* (deletion-only) case. However, in general, vertices may be deleted and re-inserted to S as often as we like; this naive approach would have cost proportional to the degree of the vertex, which can be linear in n for every update in the worst case. We give a nontrivial sublinear solution for arbitrary update sequences in general sparse graphs, with $\widetilde{O}(m^{4\omega/(3\omega+3)})$ amortized update time, $\widetilde{O}(m^{1/3})$ query time, $\widetilde{O}(m^{(5\omega+1)/(3\omega+3)})$ preprocessing time, and $\widetilde{O}(m)$ space.

To appreciate the result, note that existing polylogarithmic dynamic graph techniques [17, 21, 32] do not work, because the usual "certificate" [13, 24], a spanning forest, can change drastically in a single vertex update. Alternatively, it is possible to reduce the problem to reachability in a dynamic *directed* graph, so that a vertex update can be simulated by a single edge change (by creating two copies of each vertex joined by a directed edge). However, dynamic directed graph reachability [16, 30] is computationally more demanding (a main goal there was to obtain an $o(n^2)$ update bound).

Previous graph work. The dynamic subgraph connectivity problem has indeed been proposed before, in a paper by Frigioni and Italiano [16]. The motivation there was on the connectivity of communication networks, where processors can become faulty and can later go back up; viewed alternatively, vertices can be "switched" on and off. Frigioni and Italiano used the term *complete dynamic graph model* to describe the setting where both edge and vertex updates are supported. They described polylogarithmic connectivity results for the special case of planar graphs (relying on separators), but left the general case (which is necessary in our geometric application) open.

How to solve it. Our solution to dynamic subgraph connectivity involves several techniques. Each in itself is not difficult, and when put together in the right way, they yield the winning combination.

Our recipe includes:

- Processing the update sequence in blocks. Khanna *et al.* [24] have used this trick to design various dynamic graph algorithms for *offline* updates (when the update sequence is given in advance), while the author [5, 6] has applied essentially the same idea to several dynamic geometric problems under *semi-online* updates (when only the deletion times are given in advance) as well as general updates.
- Alon *et al.*'s "high-low" trick [4]. This was originally developed for the problem of finding triangles in sparse graphs. We observe how the idea can yield faster multiplication algorithms for sparse rectangular matrices (see Section 3).
- Precomputing information for high-degree vertices, so that their repeated deletions and re-insertions would not be as costly.
- Finally, amortizing deletion cost via a standard "weighted split" idea, which is analogous to the standard "weighted union" heuristic (e.g., see [8, Chapter 21]).

To illustrate the effects of these ideas progressively, we present the algorithm in stages (in the actual order of discovery), starting with an $O(m^{0.82})$ offline update method (Section 4), extending it to an $O(m^{0.91})$ semi-online update method (Section 5), and ending with the $O(m^{0.94})$ fully dynamic update method (Section 6).

2 From Geometry to Graphs

We first describe how to reduce dynamic geometric connectivity problems to dynamic subgraph connectivity. This section is the only place where computational geometry techniques are used; afterwards, we can devote all our attention to the graph problem.

The reduction is a simple consequence of known compact representations of geometric intersection graphs. Specifically, any intersection graph of boxes can be represented as a union of "bicliques" (complete bipartite subgraphs $A_i \times B_i$ over some vertex subsets A_i and B_i) so that the size of the representation (the total number of vertices in the bicliques) is near-linear.

Lemma 2.1 *Fix any constant d. Given a set of n (axis-parallel) boxes in $\mathbb{R}^d$, we can form subsets A_i and B_i of total size $\widetilde{O}(n)$ in $\widetilde{O}(n)$ time, such that two boxes a and b intersect iff (a, b) or (b, a) is in $A_i \times B_i$ for some index i. This property can be maintained dynamically: each insertion/deletion of a box causes an amortized $\widetilde{O}(1)$ number of insertions/deletions in the A_i's and B_i's.*

Proof: We first consider the static case. The following *orthogonal range (or intersection) searching* problem [2, 11, 26, 27, 28, 29] is well-studied in computational geometry: preprocess a given set of n boxes in a data structure so that given a query box b, we can quickly report all boxes intersecting b. A standard solution to this problem is the *range tree* (or a multi-level *segment tree*). The precise details behind the data structure are not important here (see the above references for more information). The main properties to note are that the data structure consists of $O(n \log^{d-1} n)$ "nodes", each of which stores a subset of boxes which we refer to as a *canonical subset*, and that given a query box b, the set of boxes intersecting b can be returned as the union of the canonical subsets at $O(\log^d n)$ nodes of the data structure. The total size of the canonical subsets is $O(n \log^d n)$; the preprocessing time is $O(n \log^d n)$ and the query time is $O(\log^d n)$.

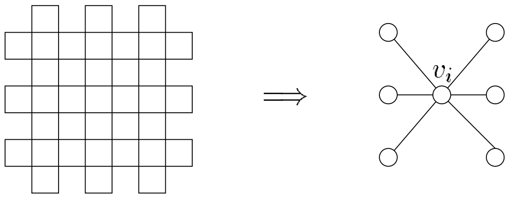

Figure 2: From dynamic box connectivity to dynamic subgraph connectivity.

To form the subsets A_i and B_i, we build the data structure for the given input boxes and perform queries for all the boxes. For each node i, we let A_i be the canonical subset at i and B_i be all boxes whose query returns the canonical subset at i. The total size of the A_i's and B_i's is $O(n \log^d n)$, and correctness is obvious.

For the dynamic case, we need a version of the orthogonal range searching problem that supports insertion. A weight-balanced range tree, for instance, can be used [26, 28]. Again the precise details behind the data structure are not important. The main additional properties are that although each canonical subset does not change, new nodes and canonical subsets may be created, and the total size of all canonical subsets created by a sequence of n insertions is bounded by $O(n \log^{d+1} n)$. (With more care, a log factor can probably be saved.)

To update the subsets A_i and B_i under the insertion of a new box b, we perform a query for b and insert b to every set B_i such that the query returns the canonical subset at node i. We then insert b to the data structure itself, and for every new node i created, we insert all elements of the new canonical subset to A_i and initialize $B_i = \emptyset$. (We do not delete canonical subsets of old nodes destroyed.) The amortized number of insertions to the A_i's and B_i's is $O(\log^{d+1} n)$.

To update the subsets A_i and B_i under the deletion of a box b, we simply remove b from each subset that contains b. The cost of deletion can be "charged" to the insertion cost by amortization. (If n is to denote the number of current boxes instead of the number of insertions, we need another standard amortization trick, of rebuilding the whole data structure whenever the value of n is halved [26, 28].) $\square$

Theorem 2.2 *Fix any constant d. The dynamic connectivity problem for boxes in $\mathbb{R}^d$ can be reduced to the dynamic subgraph connectivity problem on a graph with $m = \widetilde{O}(n)$ edges. Each box update causes an amortized $\widetilde{O}(1)$ number of graph updates.*

Proof: Given a set of boxes, we apply Lemma 2.1 and define the following graph G: for each box, we create a vertex; in addition, for each subset pair (A_i, B_i), we create a new vertex v_i and place edges from v_i to all members of $A_i \cup B_i$. (See Figure 2 for an idealized example.) Initially, all vertices are put in the subset S. Whenever A_i or B_i becomes empty, we delete the vertex v_i from S. When both A_i and B_i are nonempty, we re-insert v_i to S.

Each insertion/deletion of a box causes the subsets A_i and B_i to undergo $\widetilde{O}(1)$ amortized number of insertions/deletions, which in turn causes the above graph G and subset S to undergo $\widetilde{O}(1)$ amortized number of edge and vertex insertions/deletions.

To test whether two query boxes are connected in the intersection graph, we just test whether the two boxes are connected in the subgraph of G induced by S. Correctness is evident (if A_i and B_i are nonempty, the members of $A_i \cup B_i$ are indeed all connected to each other in the intersection graph). To test whether two query points are connected, we first find a box containing each point (which takes polylogarithmic time by known data structures for orthogonal range searching) and then test whether these two boxes are connected. □

Remarks. The above reduction also works in the static, incremental (insertion-only), decremental (deletion-only), offline, and semi-online settings:

In the static case, graph and subgraph connectivity can be solved in linear time by depth-first search, so we automatically get an $O(n \operatorname{polylog} n)$ algorithm for the box connectivity problem. This result was known before, as noted in Section 1.

In the incremental/decremental case, subgraph connectivity reduces to graph connectivity, as we have observed in Section 1 (by explicitly maintaining the induced subgraph). Incremental graph connectivity is equivalent to the union-find problem [8], and decremental graph connectivity can be solved in near-logarithmic time [31, 32], so we automatically get $O(\operatorname{polylog} n)$ incremental/decremental box connectivity algorithms. (We do not state the precise polylogarithmic bounds, because slight improvements are likely possible by a more direct approach.) The incremental box connectivity result was already known [3], but the decremental box connectivity has not been addressed before.

In the offline, semi-online, and fully dynamic cases, the subgraph connectivity results in Sections 4–6 will imply corresponding box connectivity results, up to polylogarithmic factors.

Other classes of objects can be considered. For example, we can obtain the same results for line segments that have a fixed number of slopes, by using variants of orthogonal range searching in the proof of Lemma 2.1.

For arbitrary line segments in the plane, we need to use known non-orthogonal range searching data structures [2] in the proof of Lemma 2.1, but these data structures require canonical subsets of total size $\widetilde{O}(n^{4/3})$ instead of $O(n \operatorname{polylog} n)$; as a result, the graph in Theorem 2.2 now has $m = \widetilde{O}(n^{4/3})$ edges, and each segment update causes an amortized $\widetilde{O}(n^{1/3})$ number of graph updates. Unfortunately, our dynamic subgraph connectivity results are too weak to yield meaningful bounds in this case. Still, we can obtain an $\widetilde{O}(n^{4/3})$ result for static connectivity and an $\widetilde{O}(n^{1/3})$ result for incremental and decremental connectivity for arbitrary line segments. This static result was known [25], so was the incremental result [3], but the decremental result appears new.

3 Fast Sparse Matrix Multiplication

Our sublinear results for dynamic subgraph connectivity will require a time bound for matrix multiplication that is sensitive to the sparseness of the given matrices. Although obtaining such an input-sensitive bound is in general an outstanding problem for the standard square-matrix case (see [34] for an independent, recent development), in this section we note a simple input-sensitive bound for a rectangular-matrix case that is sufficient for our application.

Specifically, we consider the complexity $M(n, q \mid m)$ of multiplying a $q \times n$ 0-1 matrix A with an $n \times q$ 0-1 matrix B, where m is the number of nonzero entries per matrix, with $m = \Omega(n)$. This is asymptotically equivalent to the complexity of the following graph problem: given an m-edge bipartite graph with a set P of n vertices on one side and a set Q of q vertices on the other, count the number $C[u, v]$ of vertices adjacent to both u and v, for every vertex pair $u, v \in Q$. (To reduce the graph problem to matrix multiplication, let $a_{vw} = 1$ iff $b_{wv} = 1$ iff $w \in P$ and $v \in Q$ are adjacent; then $C[u, v] = \sum_{w \in P} a_{uw} b_{wv}$. Conversely, to reduce matrix multiplication to the graph problem, let $P = \{1, \ldots, n\}$ and $Q = \{1, \ldots, q\} \times \{1, 2\}$ (of size $2q$), and place an edge between k and $(j, 1)$ if $a_{jk} = 1$, and an edge between k and $(j, 2)$ if $b_{jk} = 1$; then $C[(i, 1), (j, 2)] = \sum_{k=1}^{n} a_{ik} b_{kj}$.)

We are primarily interested in the case when q is small. For dense matrices, we have the upper bound $M(n, q \mid m) = O(nq^{\omega-1} + q^{\omega})$, since we can solve the problem by multiplying $\lceil n/q \rceil$ pairs of $q \times q$ square submatrices. To take the sparseness m into account, we adapt a simple trick by Alon *et al.* [4]:

Lemma 3.1 $M(n, q \mid m) = O(mq^{(\omega-1)/2} + q^{\omega})$.

Proof: Consider the graph formulation. Divide P into two groups: P_H, vertices of degree $> r$, and P_L, vertices of degree $\leq r$. Then $|P_H| = O(m/r)$. We first set $C[u, v]$ to be the number of vertices in P_H adjacent to both u and v, for every pair $u, v \in Q$; this takes time $O(M(m/r, q \mid m)) = O(mq^{\omega-1}/r + q^{\omega})$ by the dense-matrix bound. To complete the overall count, we can examine each edge uw incident to a vertex $w \in P_L$, go through all adjacent edges wv, and increment $C[u, v]$; this takes $O(mr)$ time. Setting $r = q^{(\omega-1)/2}$ yields the desired bound. $\square$

Remark. There are improved rectangular matrix multiplication methods [22] for dense matrices, which may lead to slight improvements to Lemma 3.1 for certain ranges of parameters. However, these improvements do not seem to matter here, since we will eventually select parameters to equalize the contribution of the terms $mq^{(\omega-1)/2}$ and q^{ω}, and the critical case will occur when the dense submatrices are essentially square matrices.

4 An Offline Solution

In the next three sections, we present our algorithms for dynamic subgraph connectivity. We begin by considering the offline case, where we are given the update sequence in advance (or the ability to look sufficiently far ahead in the update sequence). The approach is simple: we divide a dynamic set into two subsets, P and Q, where P is static and Q is dynamic but small; to keep Q small, we rebuild the data structure from scratch periodically, after a certain number of updates. Khanna *et al.* [24] have adopted this very approach for strongly connected components and reachability in directed graphs. We obtain a sublinear result in m by, in addition, employing Lemma 3.1.

In what follows, let $G = (V, E)$ be the input graph, with n vertices and m edges. Without loss of generality, assume that $m = \Omega(n)$. For clarity sake, we focus only on the more difficult update operations, i.e., vertex insertions/deletions in S; edge insertions/deletions in G will be treated later, in the remarks after Theorem 6.2.

Lemma 4.1 *Given a static subset $P \subseteq V$ and a static subset $Q_0 \subseteq V$ of size q_0, we can design a data structure to maintain a subset $Q \subseteq Q_0$, where preprocessing takes $\widetilde{O}(mq_0^{(\omega-1)/2} + q_0^{\omega})$ time, updates to Q take $\widetilde{O}(q_0)$ amortized time, and connectivity queries on the subgraph induced by $P \cup Q$ take $\widetilde{O}(q_0)$ time.*

Proof: We first compute the connected components of the subgraph induced by P in linear time by depth-first search [8]. Our data structure consists of three parts:

- We form a bipartite multigraph Γ, with the connected components as vertices on one side and V as vertices on the other side: for each edge $uv \in E$ with $u \in P$, we place a corresponding edge γv in Γ for the component γ containing u. The $O(m)$ edges of Γ are stored in a dictionary [8]. (For example, a balanced search tree can support updates and lookups in $O(\log n)$ time; alternatively, hashing can support these operations in $O(1)$ randomized time.)

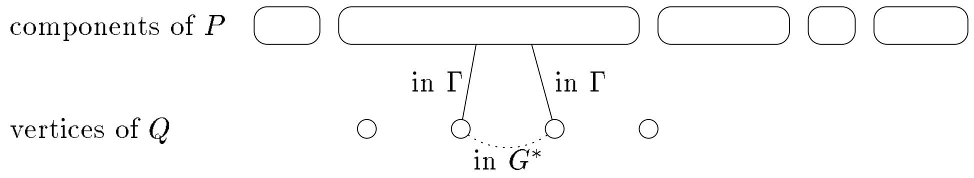

Figure 3: Overview of the data structure.

- We precompute and store the following values

$$C[u,v] \;=\; \text{number of components adjacent to both } u \text{ and } v \text{ in } \Gamma, \qquad (1)$$

  over all $u,v \in Q_0$. This preprocessing step takes $O(mq_0^{(\omega-1)/2}+q_0^{\omega})$ time by applying Lemma 3.1 to a bipartite subgraph of Γ (with Q_0 on one side and edge multiplicities ignored).

- Finally, we maintain a small dynamic graph G^* over the vertex set Q, where for every $u,v \in Q$,

$$uv \text{ is in } G^* \text{ iff } C[u,v] > 0 \text{ or } uv \in E. \qquad (2)$$

  (See Figure 3.) This graph is stored in a polylogarithmic data structure for dynamic graph connectivity. (For example, Holm *et al.*'s method [21] can support updates in $O(\log^2 n)$ amortized time and queries in $O(\log n/\log\log n)$ time; alternatively, Thorup's improved method [32] can support updates in $O(\log n \log^3 \log n)$ randomized amortized time and queries in $O(\log n/\log\log\log n)$ time.)

Insertions/deletions in Q: To insert a vertex u to Q, we simply insert edges uv to G^* for all $v \in Q$ with $C[u,v] > 0$ or $uv \in E$. To delete u from Q, we delete all edges incident to u from G^*. For $|Q| = q$, this requires at most q edge updates to the dynamic graph connectivity structure for G^* and costs $\widetilde{O}(q)$ amortized time.

Queries on $P \cup Q$: Given vertices $u,v \in P \cup Q$, we want to test whether u and v are connected in the subgraph induced by $P \cup Q$.

- Easiest Case: $u,v \in Q$. We can simply test whether u and v are connected in G^*, in $\widetilde{O}(1)$ time by the dynamic graph connectivity structure for G^*. Correctness is evident (if there is a path connecting u and v in the subgraph induced by $P \cup Q$, then the path must alternate between some vertices of Q and some component of P, so there must be a corresponding path in G^*).

- Hardest Case: $u,v \in P$. Here, we first find the components γ_u and γ_v containing u and v. We then find any $u' \in Q$ with $\gamma_u u' \in \Gamma$ and any $v' \in Q$ with $\gamma_v v' \in \Gamma$ by performing $O(q)$ dictionary lookups in $\widetilde{O}(q)$ time. If u' or v' does not exist, the answer is no, unless $\gamma_u = \gamma_v$. Otherwise, we can just test whether u' and v' are connected by the previous case.

The remaining cases are similar. □

Theorem 4.2 *We can design a data structure to maintain a subset $S \subseteq V$ under any* offline *update sequence, where preprocessing takes $\widetilde{O}(m^{2\omega/(\omega+1)}) = O(m^{1.41})$ time, updates to S take $\widetilde{O}(m^{2(\omega-1)/(\omega+1)}) = O(m^{0.82})$ amortized time, and connectivity queries on the subgraph induced by S take $\widetilde{O}(m^{2/(\omega+1)}) = O(m^{0.60})$ time.*

Proof: At the beginning of each block of q_0 updates, we set Q_0 to be the vertices involved in the coming q_0 updates, set $P = S \setminus Q_0$, set $Q = S \cap Q_0$, and rebuild the data structure from Lemma 4.1. Updates to S are applied to Q, with amortized cost

$$\widetilde{O}\left(\frac{mq_0^{(\omega-1)/2} + q_0^\omega}{q_0} + q_0\right),$$

which is asymptotically minimized by setting $q_0 = m^{2/(\omega+1)}$. Queries can be answered in $\widetilde{O}(q_0)$ time, since $S = P \cup Q$ at all times. □

5 A Semi-Online Solution

It is perhaps not surprising that an offline problem can be solved more quickly by batching and performing fast matrix multiplication. It is interesting, however, that with more effort a similar strategy can lead to a fully dynamic solution to our problem. Although we cannot predict which vertices are about to be inserted in advance, we can concentrate preprocessing on vertices of high degrees, since these vertices are the costly ones. To obtain a sublinear bound, we now have to rebuild the data structure more frequently, as the setting of parameters becomes more delicate.

Before describing the fully dynamic algorithm, we consider the semi-online case [10], where we are told when a vertex will next be deleted at the time it is inserted. Because of the extra information, we can force all deletions to occur in the small subset Q, thereby ensuring that P is static (see [6] for more examples of this kind of dynamization).

Lemma 5.1 *Given a subset $P \subseteq V$ and a parameter $q_0 \le m$, we can design a data structure to maintain a subset $Q \subseteq V$ of size $q \le q_0$, where preprocessing takes $\widetilde{O}(mq_0^{(\omega-1)/2} + q_0^\omega)$ time, updates to Q take $\widetilde{O}(mq/q_0)$ amortized time, and connectivity queries on the subgraph induced by $P \cup Q$ take $\widetilde{O}(q)$ time.*

Proof: Set Q_0 to contain all vertices in V of degree $> m/q_0$ in Γ. Then $|Q_0| = O(q_0)$. As in the proof of Lemma 4.1, we form the same bipartite multigraph Γ and precompute the same values $C[u, v]$ (as defined by (1)) for all $u, v \in Q_0$. In addition, we now maintain $C[u, v]$ for all $u, v \in Q$ as well, in order to keep track of the graph G^* over the vertex set Q (as defined by (2)). Preprocessing time is still $O(mq_0^{(\omega-1)/2} + q_0^\omega)$, and queries can be answered in the same way in $\widetilde{O}(q)$ time.

Insertions/deletions in Q: Deletions are as before. Insertions are more involved, because Q is not necessarily a subset of Q_0, so new entries of $C[\cdot, \cdot]$ need to be computed. To insert a vertex u to Q, consider two cases:

- Case 1: $u \notin Q_0$, i.e., u has degree $\le m/q_0$ in Γ. For each $v \in Q$, we can compute $C[u, v]$ by going through each of the $O(m/q_0)$ different components γ adjacent to u in Γ and testing whether $\gamma v \in \Gamma$ in $\widetilde{O}(1)$ time via a dictionary lookup. The total time of this step is $\widetilde{O}(qm/q_0)$.
- Case 2: $u \in Q_0$. For each $v \in Q_0$, $C[u, v]$ has already been precomputed. For each $v \in Q \setminus Q_0$, since v has degree $\le m/q_0$ in Γ, we can compute $C[u, v]$, by going through each of the $O(m/q_0)$ different components γ adjacent to v in Γ and testing whether $\gamma u \in \Gamma$ in $\widetilde{O}(1)$ time via a dictionary lookup. The total time in this case is also $\widetilde{O}(qm/q_0)$.

We can now maintain the graph G^*, as in the proof of Lemma 4.1, by at most q edge updates to the dynamic graph connectivity structure, in additional $\widetilde{O}(q)$ time. □

Theorem 5.2 *We can design a data structure to maintain a subset $S \subseteq V$ under any* semi-online *update sequence, where preprocessing takes $\widetilde{O}(m^{2\omega/(\omega+1)}) = O(m^{1.41})$ time, updates to S take $\widetilde{O}(m^{(3\omega-1)/(2\omega+2)}) = O(m^{0.91})$ amortized time, and connectivity queries on the subgraph induced by S take $\widetilde{O}(m^{1/2})$ time.*

Proof: At the beginning of each block of q updates, we set Q to contain the vertices in S with the q smallest deletion times, set $P = S \setminus Q$, and rebuild the data structure from Lemma 5.1. Each insertion in S is applied to Q with $|Q| \leq 2q$ at all times, and deletions in S can occur to Q only. The amortized update cost is

$$\widetilde{O}\left(\frac{mq_0^{(\omega-1)/2} + q_0^{\omega}}{q} + \frac{mq}{q_0}\right),$$

which is asymptotically minimized by setting $q_0 = m^{2/(\omega+1)}$ and $q = m^{1/2}$. □

6 A Fully Dynamic Solution

In the fully dynamic problem, we have no control on future deletions, so the subset P can no longer remain static. However, since P undergoes deletions only, we can bring in amortization techniques (see [5] for a similar, geometric example). Deletions cause splitting of components, and a standard idea is to always split the smaller set from the larger set. Remarkably, the matrix-multiplication output can be updated efficiently during this process, due to the linear dependence on m in Lemma 3.1.

Lemma 6.1 *We can make the data structure in Lemma 5.1 support an additional operation: deletion in P. The total cost of ℓ such deletions is bounded by $\widetilde{O}(mq_0^{(\omega-1)/2} + \ell q_0^{\omega})$.*

Proof: The data structure is the same as in the proof of Lemma 5.1, but with one additional ingredient: To maintain the connected components of P, we store the subgraph induced by P in a decremental graph connectivity data structure [17, 31, 32]; the total maintenance cost over a sequence of deletions is $\widetilde{O}(m)$. Note that such a structure can handle auxiliary operations, such as reporting the size of a component, or enumerating the vertices of a component. Insertions and deletions in Q are done exactly as in the proof of Lemma 5.1, it remains to describe how to perform deletions in P.

Deletions in P: To delete a vertex w from P, we find the component γ containing w and observe how γ is split into several components $\gamma_1, \ldots, \gamma_k$ from the decremental graph connectivity structure. Without loss of generality, suppose that γ_1 has the largest size. Note that each γ_i $(i = 2, \ldots, k)$ has at most half the size of γ. Let m' be the sum of the degrees over all vertices of $\{w\} \cup \gamma_2 \cup \cdots \cup \gamma_k$ in G.

Both Γ and $C[\cdot,\cdot]$ change as a result of the split.

- To update the multigraph Γ, we perform the following steps: For each edge $uv \in E$ with $u \in \gamma_i$ $(i = 2, \ldots, k)$, we insert a copy of $\gamma_i v$ and remove a copy of γv. For each $wv \in E$, we remove a copy of γv. Finally, we remove $\{w\} \cup \gamma_2 \cup \cdots \cup \gamma_k$ from γ so that γ becomes γ_1. These steps require $O(m')$ dictionary updates and take $\widetilde{O}(m')$ time.

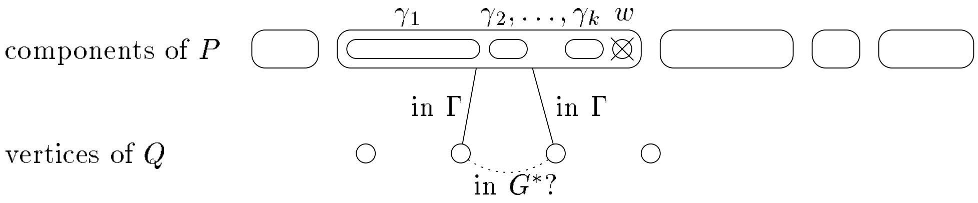

Figure 4: Making the data structure fully dynamic.

- To update the $C[\cdot,\cdot]$ values, we perform the following steps: For each $(u, v) \in (Q_0 \times Q_0) \cup (Q \times Q)$, if $\gamma u, \gamma v \in \Gamma$ before, we decrement $C[u, v]$. For each $(u, v) \in (Q_0 \times Q_0) \cup (Q \times Q)$, if $\gamma_1 u, \gamma_1 v \in \Gamma$, we increment $C[u, v]$. These steps require $O(q^2)$ dictionary lookups and take $\widetilde{O}(q_0^2)$ time. Finally, for each $(u, v) \in (Q_0 \times Q_0) \cup (Q \times Q)$, we add to $C[u, v]$ the number of components from $\{\gamma_2, \ldots, \gamma_k\}$ adjacent to both u and v in Γ. These numbers can be computed in $O(m' q_0^{(\omega-1)/2} + q_0^\omega)$ total time by applying Lemma 3.1 to a bipartite subgraph of Γ with $O(m')$ edges (with $\gamma_2, \ldots, \gamma_k$ on one side and Q_0 or Q on the other).

We can now rebuild the graph G^* from scratch (according to the definition (2)) in time $O(q^2)$, which is absorbed by the other cost.

We can account for the $O(m' q_0^{(\omega-1)/2} + q_0^\omega)$ cost by charging $O(\deg(v) q_0^{(\omega-1)/2})$ units to each vertex $v \in \{w\} \cup \gamma_2 \cup \cdots \cup \gamma_k$, and charging $O(q_0^\omega)$ units to the deletion operation itself. Here, $\deg(v)$ denotes the degree of v in G. Each vertex v is charged at most $O(\log n)$ times overall, since each time v is charged, the component containing v shrinks at least by a factor of two in size (and components never expand, as P undergoes only deletions). Therefore, the total cost charged to vertices is bounded by $O(\sum_{v \in V} \deg(v) q_0^{(\omega-1)/2} \log n) = \widetilde{O}(m q_0^{(\omega-1)/2})$. The total cost charged to the deletion operations is $O(\ell q_0^\omega)$. $\square$

Theorem 6.2 *We can design a data structure to maintain a subset $S \subseteq V$ under any* online *update sequence, where preprocessing takes $\widetilde{O}(m^{(5\omega+1)/(3\omega+3)}) = O(m^{1.28})$ amortized time, updates to S take $\widetilde{O}(m^{4\omega/(3\omega+3)}) = O(m^{0.94})$ amortized time, and connectivity queries on the subgraph induced by S take $\widetilde{O}(m^{1/3})$ time.*

Proof: At the beginning of each block of q updates, we set $P = S$, set $Q = \emptyset$, and rebuild the data structure from Lemma 6.1. Insertions in S are applied to Q, with $|Q| \leq q$ at all times, but deletions can occur to both P and Q. The amortized update cost is

$$\widetilde{O}\left(\frac{m q_0^{(\omega-1)/2} + q q_0^\omega}{q} + \frac{mq}{q_0}\right),$$

which is asymptotically minimized by setting $q_0 = q^{4/(\omega+1)}$ and $q = m^{1/3}$. $\square$

Remarks. The space requirement of the data structure is indeed $\widetilde{O}(m)$: The dynamic graph connectivity structures take $\widetilde{O}(m)$ space, the dictionary for Γ takes $O(m)$ space, and the $C[\cdot,\cdot]$ entries take $O(q_0^2)$ space, which is sublinear for the parameters chosen in Theorem 6.2's proof.

We can obtain a query-update tradeoff version of Theorem 6.2: With $\widetilde{O}(q)$ query time for any given parameter $q \leq m^{1/3}$, the amortized update time is $\widetilde{O}(m/q^{(3-\omega)/(\omega+1)}) = \widetilde{O}(m/q^{0.18})$.

Edge updates are indeed less difficult than vertex updates: For example, to insert a new edge vw to E, we can create a new dummy vertex u joined to v and w and then insert u to S (i.e., Q). In Lemma 5.1's proof, since u has degree 2 only, we can update Γ in $\widetilde{O}(1)$ time and follow Case 1 to update $C[\cdot,\cdot]$ in less than $\widetilde{O}(q_0 m/q)$ time (note that Q_0 does not change). Later, to delete the edge vw, we can simply delete the dummy vertex u from S. Therefore, the cost of an edge insertion/deletion is at most the cost of a vertex insertion/deletion.

7 Discussions

Is fast matrix multiplication necessary? Our algorithms have limited practical appeal because of the use of fast matrix multiplication (FMM). One may wonder whether FMM is essential to solve our problem. We suspect that the answer might be yes, in view of the following observations:

Observation 7.1

1. *The problem of multiplying a $\sqrt{n} \times n$ Boolean matrix with an $n \times \sqrt{n}$ Boolean matrix with m nonzero entries can be reduced to offline dynamic subgraph connectivity on a graph with n vertices and m edges using $O(n)$ updates and queries.*

2. *The problem of detecting a triangle (a 3-cycle) in a directed graph with m edges can be reduced to offline dynamic subgraph connectivity using $O(m)$ updates and queries.*

3. *The problem of detecting a quadilateral (a 4-cycle) in a directed graph with m edges can be reduced to offline dynamic subgraph connectivity using $O(m)$ updates and queries.*

Proof:

1. As noted essentially in Section 3, an equivalent problem is the following: given a bipartite graph G with a set P of n vertices on one side and a set Q of $O(\sqrt{n})$ vertices on the other side, decide whether u and v are adjacent to a common vertex for every pair $u, v \in Q$. To solve this problem, we first put all vertices of P in the subset S. For each pair $u, v \in Q$, we insert u and v to S, test whether u and v are connected in the subgraph induced by S, and then delete u and v from S. The number of queries and vertex updates to S is $O(|Q|^2) = O(n)$.

2. Let $H = (V, E)$ be the given graph. Define an undirected graph G with vertex set $V \times \{(1, 2, 3)\}$, where we create edges $(u, 1)(v, 2)$ and $(u, 2)(v, 3)$ whenever $uv \in E$. Initially, we put all vertices of the form $(w, 2)$ in S. To detect a triangle in H, we go through each edge $(v, u) \in E$, insert $(u, 1)$ and $(v, 3)$ to S, test whether $(u, 1)$ and $(v, 3)$ are connected, and then delete $(u, 1)$ and $(v, 3)$ from S. The answer is yes iff one of the tests returns true.

3. Define G as before, but with the addition of a vertex s adjacent to all vertices of the form $(u, 1)$, and another vertex t adjacent to all vertices of the form $(v, 3)$. Initially, we put s and t in S, along with all vertices of the form $(w, 2)$. To detect a quadilateral in H, we go through each vertex $z \in V$, insert all vertices in the subsets $A = \{(u, 1) \mid (z, u) \in E\}$ and $B = \{(v, 3) \mid (v, z) \in E\}$ to S, test whether s and t are connected, and then delete all vertices in A and B from S. The answer is yes iff one of the tests returns true (if s and t are connected, there is a length-2 path from A to B, yielding a quadilaterial through z). The number of vertex updates to S is $O(\sum_{z \in V} \deg(z)) = O(m)$. $\square$

For the first problem, the best bound we know that does not use FMM is $O(m\sqrt{n})$ (by Lemma 3.1, with $\omega = 3$). So, it is unlikely that offline dynamic subgraph connectivity can be solved in $o(n^{1/2})$ time without some kind of FMM (though this doesn't rule out the possibility of a sublinear bound without FMM). For the second and third problem, the best algorithms known for sparse graphs, due to Alon *et al.* [4, 33], both require FMM, and run in time $O(m^{2\omega/(\omega+1)}) = O(m^{1.41})$ and (a little less than) $O(m^{(4\omega-1)/(2\omega+1)}) = O(m^{1.48})$ respectively.

Is the graph problem necessary? The problem we start with is geometric, but the solution we give is mostly graph-theoretic. One may wonder whether this is the right approach. For $d = 3$, the answer is yes, as shown below:

Observation 7.2 *The dynamic subgraph connectivity problem can be reduced to the dynamic box connectivity problem in* $\mathbb{R}^3$.

Proof: We use only orthogonal segments (degenerate boxes) in $\mathbb{R}^3$: For each vertex i of the given graph, construct a line ℓ_i from $(i, -\infty, 0)$ to $(i, \infty, 0)$. For the k-th edge ij, create a path π_k of three segments through the points $(i, k, 0), (i, k, 1), (j, k, 1), (j, k, 0)$. Then i and j are connected iff ℓ_i and ℓ_j are connected. Inserting/deleting a vertex i in S corresponds to inserting/deleting ℓ_i. Inserting/deleting an edge in E corresponds to inserting/deleting a π_k. $\square$

Thus, by Theorem 2.2, subgraph connnectivity is *equivalent* to box connectivity in three dimensions, up to polylogarithmic factors. In particular, box connectivity in any fixed dimension ≥ 3 is equally difficult, up to polylogarithmic factors. It is intriguing to contemplate whether one can exploit the geometry of the rectangular connectivity problem to get a faster algorithm for $d = 2$ (perhaps without FMM).

Global connectivity? On a final note, we have purposely defined connectivity queries as just deciding whether two points, or two vertices, are connected. One may wonder about other kinds of connectivity queries, for example, "is the union of the boxes connected?", or "is the subgraph induced by S connected?"

Obtaining nontrivial results for such queries appear difficult, even in the offline setting. A major obstacle appears to be the following dynamic set union problem: Given a collection $\mathcal{C}$ of n subsets of U, of total size m, maintain a subcollection $\mathcal{S} \subseteq \mathcal{C}$ under insertions and deletions of subsets, to answer the following query, "is the union of $\mathcal{S}$ equal to U?"

References

[1] P. K. Agarwal, N. Alon, B. Aronov, and S. Suri. Can visibility graphs be represented compactly? *Discrete Comput. Geom.*, 12:347–365, 1994.

[2] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry* (B. Chazelle, J. E. Goodman, and R. Pollack, eds.), pp. 1–56, AMS Press, 1999.

[3] P. K. Agarwal and M. van Kreveld. Polygon and connected component intersection searching. *Algorithmica*, 15:626–660, 1996.

[4] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.

[5] T. M. Chan. A fully dynamic algorithm for planar width. *Discrete Comput. Geom.*, 30:17–24, 2003.

[6] T. M. Chan. Semi-online maintenance of geometric optima and measures. *SIAM J. Comput.*, 32:700–716, 2003.

[7] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9:251–280, 1990.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 2nd ed., 2001.

[9] C. Demetrescu and G. F. Italiano. Trade-offs for fully dynamic transitive closure on DAGs: breaking through the $O(n^2)$ barrier. *J. ACM*, 52:147–156, 2005.

[10] D. Dobkin and S. Suri. Maintenance of geometric extrema. *J. ACM*, 38:275–298, 1991.

[11] H. Edelsbrunner and H. A. Maurer. On the intersection of orthogonal objects. *Inform. Process. Lett.*, 13:177–181, 1981.

[12] D. Eppstein. Dynamic Euclidean minimum spanning trees and extrema of binary functions. *Discrete Comput. Geom.*, 13:111–122, 1995.

[13] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparisification: a technique for speeding up dynamic graph algorithms. *J. ACM*, 44:669–696, 1997.

[14] T. Feder and R. Motwani. Clique partitions, graph compression and speeding up algorithms. *J. Comput. Sys. Sci.*, 51:261–272, 1995.

[15] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14:781–798, 1985.

[16] D. Frigioni and G. F. Italiano. Dynamically switching vertices in planar graphs. *Algorithmica*, 28:76–103, 2000.

[17] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46:502–516, 1999.

[18] M. R. Henzinger and V. King. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. Comput.*, 31:364–374, 2001.

[19] J. Hershberger and S. Suri. Kinetic connectivity of rectangles. In *Proc. 15th ACM Sympos. Comput. Geom.*, pages 237–246, 1999.

[20] J. Hershberger and S. Suri. Simplified kinetic connectivity for rectangles and hypercubes. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, pages 158–167, 2001.

[21] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 2001.

[22] X. Huang and V. Y. Pan. Fast rectangular matrix multiplication and applications. *J. Complexity*, 14:257–299, 1998.

[23] H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *J. Algorithms*, 4:310–323, 1983.

[24] S. Khanna, R. Motwani, and R. H. Wilson. On certificates and lookahead on dynamic graph problems. *Algorithmica*, 21:377–394, 1998.

[25] M. A. Lopez and R. Thurimella. On computing connected components of line segments. *IEEE Trans. Comput.*, 44:597–601, 1995.

[26] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry.* Springer-Verlag, Heidelberg, 1984.

[27] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[28] M. H. Overmars. *The Design of Dynamic Data Structures.* Lect. Notes in Comput. Sci., vol. 156, Springer-Verlag, 1983.

[29] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[30] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proc. 43rd IEEE Sympos. Found. Comput. Sci.*, pages 679–689, 2002.

[31] M. Thorup. Decremental dynamic connectivity. *J. Algorithms*, 33:229–243, 1999.

[32] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proc. 32nd ACM Sympos. Theory Comput.*, pages 343–350, 2000.

[33] R. Yuster and U. Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proc. 15th ACM-SIAM Sympos. Discrete Algorithms*, pages 247–253, 2004.

[34] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1:2–13, 2005.