Maintaining Chordal Graphs Dynamically: Improved Upper and Lower Bounds

Niranka Banerjee[1](✉), Venkatesh Raman[1], and Srinivasa Rao Satti[2]

[1] The Institute of Mathematical Sciences, HBNI, CIT Campus, Taramani, Chennai 600 113, India
{nirankab,vraman}@imsc.res.in

[2] Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul 151-744, Korea
ssrao@cse.snu.ac.kr

Abstract. We study upper and lower bounds for the problem of maintaining a chordal graph G under edge insertions and deletions. Let G be a chordal graph on n vertices and m edges and let (u, v) be the edge to be deleted or inserted.

– Let k be the size of the maximum clique in G. Our first result is an improved analysis of an earlier approach due to Ibarra [12] to support edge deletions. We can construct a data structure in $O(nk^2)$ time such that we can report in $O(1)$ time if $G\backslash(u, v)$ is chordal and if it is, we can update the structure in $O(n+k^2)$ time. We then show using a charging argument that the update time can be improved to $O(n^2/\Delta + k^2)$ amortized time over a sequence of Δ deletions.
– We develop a data structure to maintain a *perfect elimination ordering* (PEO) of chordal graphs where we can detect whether $G\backslash(u, v)$ is chordal in $O(\min\{degree(u), degree(v)\})$ time, and if it is chordal, we can update the structure in $O(degree(u) + degree(v))$ time. In graphs of bounded degree, our query and update bounds are a constant.
– Finally, we show that we can obtain a PEO of the graph from a clique-tree in $O(n)$ time after an edge insertion or deletion (against a naive $O(m + n)$ time). This answers a question posed by Ibarra [12].

Regarding lower bounds, we show that any dynamic structure to maintain a chordal graph requires $\Omega(\log n)$ amortized time per edge addition or deletion or per query to detect chordality, in the cell probe model with word size $\log n$.

1 Introduction

A graph is chordal if every cycle of size four or more in the graph has a chord. The study of chordal graphs has quite a rich history and the class of graphs has found use in a wide range of areas such as in biology, artificial intelligence, database systems and facility location problems [4,7,18]. There are $O(m + n)$ time algorithms to detect whether a graph on n vertices and m edges is chordal by computing what is called a *perfect elimination ordering* or *a clique tree decomposition* of the graph [17,18]. A perfect elimination ordering in a graph is an ordering of the vertices of the graph such that for each vertex x, x and the neighbors of x that occur after x in the ordering form a clique. A graph is chordal if and only if it has a perfect elimination ordering (PEO) of its vertices. Another characterization of chordal graphs is in the form of clique trees. A *clique tree* of a graph is a tree decomposition of the graph, where the bags in each node of the decomposition induce a maximal clique. A graph is chordal if and only if it has a clique tree [3,19].

We consider the problem of maintaining chordal graphs under edge insertions and deletions. As in the well-studied area of dynamic graph algorithms, we want our algorithm to report and update faster than what would require in the static algorithm to test chordality of the resulting graph from scratch, i.e. better than $O(m + n)$ time. Sometimes it is convenient to restrict the update operations on the graph. If we are allowed only insertions on the graph, then a structure supporting such an operation is said to work in the incremental setting. Similarly, if we are allowed only delete operations, such a structure is said to work in the decremental setting. A structure that allows both insertions and deletions is called a fully dynamic structure.

Our structures (as in the case of previous ones for the problem) always maintain a chordal graph in that whenever the addition or deletion of an edge makes the graph non-chordal, the algorithm reports that it is non-chordal and does not perform the update. We call the operation that detects chordality and returns a yes or no answer as a *query*, and the operation to update the resulting graph (if the resulting graph is chordal) as the *update* operation.

1.1 Previous Work

Ibarra [12] developed two fully dynamic algorithms for maintaining chordality. First one has a query and update time of $O(n)$ with a preprocessing time of $O(m + n)$, while the other has a query time of $O(\sqrt{m})$ and the update time of $O(m + n)$ with a preprocessing time of $O(mn + n^2)$. The latter is particularly useful for sparse graphs. Mezzini [13] developed a fully dynamic algorithm with $O(1)$ query and $O(n^2)$ update time, for both addition and deletion of edges. Berry et al. [1] gave an algorithm which takes amortized $O(n)$ time for insertion, deletion and deletion queries and an amortized $O(1)$ time for insertion queries.

Tarjan and Yannakakis [18] give an algorithm to convert what are called acyclic hypergraphs (clique trees are acyclic hypergraphs) to PEO in $O(m + n)$ time. In this paper, we give a data structure to augment clique trees and an algorithm to obtain a PEO from that, to support maintenance of PEO under insertions and deletions of edges in $O(n)$ time.

Logarithmic time lower bounds for update and query times for graph problems were first developed by Fredman et al. [10], who gave an $\Omega(\log n/ \log\log n)$ amortized query and update times for fully dynamic connectivity in the cell probe model with word size $O(\log n)$. The cell probe model [20] is a useful model for proving lower bounds of algorithms; computation in this model is framed as querying a set of memory cells. Patrascu et al. [16] improved the bounds to $\Omega(\log n)$ amortized. For maintaining special classes of graphs, Hell et al. [9] gave an $\Omega(\log n/\log\log n)$ amortized lower bound per update and query operation for fully dynamic recognition of proper interval graphs with word size of $O(\log n)$. It is not difficult to improve the lower bound to $\Omega(\log n)$ by applying the result of Patrascu et al. [16]. Hell et al. [9] also state without proof that a similar technique may be applied to get lower bounds for dynamic recognition of chordal graphs. In this paper, we give a formal proof of the $\Omega(\log n)$ lower bound that also applies for a few other subclasses of chordal graphs.

Amortization bounds in all these structures report the time per update/query required by the algorithm over a long sequence of query and update operations. In particular, the algorithm will not perform an update if the graph property is not satisfied on the resulting graph. For example if at some update, an edge is deleted and the resulting graph is not chordal then we insert the edge back again and continue to the next update.

1.2 Our Results

Our first structure follows Ibarra's approach in maintaining chordality by maintaining a clique tree decomposition of the chordal graph. However, we design and analyze our structures based on the maximum size k of a bag in the clique tree. Specifically we show that we can construct a data structure in $O(nk^2)$ time such that given an edge e to be deleted from G, we can report in $O(1)$ time if $G\backslash e$ is chordal and if it is, we can update the structure in $O(n+k^2)$ time. For example, for planar graphs where the maximum size of a clique is a constant, our structure supports an $O(1)$ query and $O(n)$ update time in the worst case. Using a careful charging argument, we show that the update time is actually $O(n^2/\Delta + k^2)$ amortized over Δ edge deletions. Hence, in particular if Δ is at least n^2/k^2, the amortized bound becomes $O(k^2)$. For example, if $k = \Theta(n^{1/4})$ and the initial chordal graph has $\Theta(n^{3/2}) = \Theta(n^2/k^2)$ edges, over all these edge deletions, our amortized bound for an update is $O(k^2)$ which is $O(\sqrt{n})$ while the query is still supported in constant time.

Our next results uses a perfect elimination ordering of chordal graphs.

- We show that a PEO can be represented by a dynamic list [6] so that given a query edge (u, v) to be deleted, we can detect chordality in $O(\min\{degree(u), degree(v)\})$ and update the resulting chordal graph in $O(degree(u) + degree(v))$ time. Our bounds match the bounds in Ibarra's result for the decremental setting in the worst case, but give better results when u and v have low degree. In particular, for chordal graphs with bounded degree, our method takes a constant time to update a PEO under edge deletions.
- We then give a method to augment a clique-tree decomposition of a chordal graph with simple data structures and show that we can obtain a PEO of the graph from a clique-tree in $O(n)$ time after an edge insertion or deletion(against a naive $O(m + n)$ time algorithm). This answers a question posed

by Ibarra [12]. The only non-trivial algorithms for problems such as minimum coloring, maximum independent set, minimum clique cover on chordal graphs are known via PEO [8]. Thus, our conversion from clique tree to PEO means that all of Ibarra's results which took $O(n)$ query and update time or more can now be directly translated to PEO and hence all these problems on chordal graphs can also be solved more efficiently.

Finally, we give the first non-trivial lower bound for the fully dynamic maintenance of chordal graphs. By giving a reduction from the problem of dynamically maintaining a forest under edge insertions and deletions, we show that any structure to maintain a chordal graph requires $\Omega(\log n)$ amortized time for a query or an update.

1.3 Organization of the Paper

In Sect. 2, we develop a structure using clique trees if only deletion of edges are allowed. We analyze this structure in two different ways to give a worst case and an amortized bound. Section 3 gives a worst case algorithm, on deletion of edges using a PEO ordering of the graph.

We then give an algorithm to obtain a PEO from a clique tree efficiently. Section 4 gives the lower bound for our problem.

We conclude in Sect. 5 with open problems and further directions of research.

2 Decremental Algorithms Using Clique Tree

We assume that the vertices in the graph are labelled 1, 2,...,n. The neighborhood of a vertex u refers to the adjacent vertices of u in the graph and $degree(u)$ refers to the number of neighbors of a vertex u. We start with the definiton of a clique tree decomposition of a graph. Given a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, a tree decomposition of G is a pair $(T, \{X_i\}_{i \in V(T)})$, where T is a tree, $V(T)$ is its vertex set, and there is a set $X_i \subseteq V$ associated with each node i of the tree, with the following properties [5]: (1) The union of all sets X_i equals V. (2) For every edge (u, v) in the graph, there is a subset X_i that contains both u and v. (3) For each vertex v of the graph, all the nodes X_i that v belongs to form a subtree of T.

A clique tree of a graph G is a tree decomposition where the subsets X_i in each node induce a maximal clique. To distinguish between vertices of the graph and its associated tree decomposition, we call the vertices of the tree as nodes. We will use bags and nodes interchangeably to denote the sets X_i when there is no confusion. We define the neighbors of a node in the clique tree to be its parent and all its children.

2.1 Structure with a Worst Case Update Time

We provide structures here that use the clique-tree decomposition of chordal graphs. They mainly use the following characterizations of chordal graphs.

Theorem 1 [3,19]. *A graph G is chordal if and only if G has a clique tree.*

Lemma 1 [12]. *Given a chordal graph G, and an edge $e = (u, v)$, $G\backslash e$ is chordal if and only if u and v are together present in exactly one maximal clique, and hence in only one bag of the clique tree.*

Using this, we prove the following:

Theorem 2. *Let G be a chordal graph. Let k be the maximum size of a clique in G. We can construct a data structure in $O(nk^2)$ time such that given an edge (u, v) to be deleted from G, we can report in $O(1)$ time if $G\backslash(u, v)$ is chordal and if it is, we can update the structure in $O(n + k^2)$ time.*

Proof. We first give a high level description of Ibarra's algorithm in maintaining a clique tree of the chordal graph under edge deletions. We then explain the data structures to implement it to support the operations in the claimed bounds.

Algorithm

1. Check if the given edge (u, v) is present in only one bag, if not report a negative answer, and if yes, then we need to update the clique tree.
2. If Y is the unique bag containing the edge (u, v), the node corresponding to Y is split into two nodes, Y_1 and Y_2. Y_1 now contains $Y\backslash u$ and Y_2 contains $Y\backslash v$. Y_2 becomes the parent of Y_1. From the children of Y_1 remove all nodes which contain u and make them children of Y_2. The other children remain as children of Y_1.
3. Check whether the bags of any neighbor of these newly formed nodes is a superset of the node. If yes, we "absorb" these nodes into the corresponding neighbor.
   To check whether one node is a superset of the other, Ibarra maintains the intersection size of two adjacent nodes X and Y. We denote this to be the *int* value between nodes X and Y. Let Y be the node which has been split and let ℓ be the size of Y before splitting. If $|X \cap Y| = \ell - 1$ then X absorbs the new Y. Check [12] for details.

Now we give details of the structures used to implement the algorithm. First, we build a clique tree from the given graph G. The clique tree can be represented by a pointer representation where each node points to its parent in the tree. Furthermore, we maintain the following structures.

- For each edge in the graph G we store,
  - a counter indicating the number of nodes of the clique tree to which the edge belongs, and
  - two way pointers from/to each edge to/from all the nodes it belongs to.

  We can store this structure as an adjacency matrix, with each position (u, v) in the matrix having the counter and the list of pointers. Accessing the information corresponding to edge (u, v) can be done in $O(1)$ time.
- Similarly for each vertex of the graph G we maintain a counter and a list of two way pointers to all the nodes it belongs to.

- For each node X in the clique tree, we store
  - the list of vertices sorted according to their labels,
  - For each node Y in the clique tree which is a neighbor of X, we store $|X \cap Y|$ in non-increasing order of values in an array associated with the bag X with a pointer from each cell in the array to the node it corresponds to.

Now we explain how to support the delete operation. Given a query (u, v), we first look at the counter value of (u, v) in the adjacency matrix. If it is more than one, we report that $G \backslash (u, v)$ is not chordal. Otherwise, we need to update the clique tree.

Let the node which is pointed to by the cell (u, v) in the adjacency matrix be Y and its parent be Y_{par}. Also let $|Y| = \ell$. We create two new nodes Y_1 which contains $Y \backslash u$ and Y_2 that contains $Y \backslash v$. Y_2 becomes the parent of Y_1. Y_{par} becomes Y_2's parent. For all the children of Y, all nodes which contain u become children of Y_2 and all nodes which contain v become children of Y_1. The connected subtree property ensures that v does not appear in Y_2 or any of its ancestors. So the *int* values between the nodes Y_{par}, Y_1, Y_2 and all its children remain the same as it was between Y and these children. The new nodes formed may now be subsets of any of its adjacent nodes. To maintain the clique tree, we now consider four distinct cases:

Case 1. If none of the intersections between Y_1, Y_2 and their neighborhood is $\ell - 1$ (we can find this from the sorted lists associated with Y_1 and Y_2) then neither Y_1 nor Y_2 is absorbed. In this case, update the *int* value of Y_2 and its parent by creating a sorted list for Y_2 and inserting the *int* value of $Y_2 \cap Y_{par}$ in the array of Y_{par}. Add pointers of all edges and vertices which are part of Y_2 to point at the node and change the counters.

Case 2. If Y_1 gets absorbed into one of its neighbors (check *int* of Y_1 with its neighbors and see which one is $\ell - 1$, in case of a tie choose any one), delete the node Y_1, adjust the parent pointer of its neighbor to now point at Y_2, and adjust all the parent pointers of all the other neighbors of Y_1, to point at this new node. Merge the two sorted *int* lists of Y_1 and its neighbor together.

Case 3. Our algorithm is similar to Case 2 if Y_2 gets absorbed into one of its neighbors.

Case 4. If both Y_1 and Y_2 are absorbed into one of their neighbors, the parent pointer of the neighbor which absorbs Y_1, now points to the neighbor which absorbed Y_2. The *int* value between these two nodes becomes $\ell - 2$.

Update the sorted lists in each of the new nodes formed in the clique tree.

Now, we analyze the runtime for construction of our structures. Building a clique tree requires $O(m + n)$ time. Let k be the maximum size of a node in the clique tree. From the property of chordal graphs, we know that there are a maximum of n nodes in the tree. Then there are a total of $m = O(nk^2)$ edges of the graph in the clique tree. Thus, building a clique tree takes $O(nk^2)$ time giving a total preprocessing time of $O(nk^2)$. Storing a counter and the pointers for each edge and vertex of the clique tree takes a total of $O(nk^2)$ time.

For deletion query, we look at the concerned cell of the adjacency matrix and check if the counter value is 1 in $O(1)$ time.

For update, for each of the cases above it takes $O(k^2)$ time to update the counters and the pointers for all edges (there are at most k^2 edges in a node) and $O(k)$ time to update for all the vertices. In $O(1)$ time we can, from the sorted *int* lists find if a node will be absorbed or not. In Case 1, updating the parent pointer information takes $O(1)$ time. Updating the sorted *int* list takes $O(\log n)$ time. In Cases 2, 3 and 4, where the nodes Y_1 and/or Y get absorbed, we need to update the pointers of all neighbors of Y and also merge two sorted lists. This takes $O(n)$ time. Updating the vertex information of each node takes $O(k)$ time. Thus the total time taken to update is $O(n + k^2)$. □

2.2 Amortized Analysis

We now give a better amortized runtime bound for the above algorithm by analyzing it differently. We show

Theorem 3. *Let G be a chordal graph. We can, in $O(nk^2)$ time, construct a data structure such that given a sequence of Δ edge deletions, we can support deletion query in $O(1)$ time and deletion update in $O(n^2/\Delta + k^2)$ amortized time.*

Proof. Updation of the structures involve the time to split a node in the clique tree and also to absorb the node into one of its neighbors and updating the clique tree. We deal with the total time taken to perform the split and absorb operations seperately.

First, we look at the total time spent for the split operations for each node. Let Y be a node in the clique tree before any deletion operation and let d be the degree of the node Y. Over the course of edge deletions, Y gets split into multiple nodes. Let us denote these set of nodes to be Y_{split}. Whenever a node from Y_{split} splits the node size decreases by one and the total cost incurred is the degree of that node. To analyze the runtime we can imagine a binary tree whose root node is Y with a node size of k. Y has two children each (because of a split) with each node of size $k - 1$. They have four children each of which correspond to a node of size $k - 2$ and so on. The total cost incurred at each level is d. The maximum height of this tree is k. So the total time spent by Y is $O(kd)$. Now, $k \sum d$ is at most $k(n - 1)$ and hence we have the total time taken by the algorithm for splitting nodes is $O(kn)$.

Now, let us analyze the total runtime for absorption of nodes in the algorithm. Let $Y's$ neighbor where it gets absorbed be Y_{nbr}. Let d be the degree of the node Y, and d_{nbr} be the degree of the node Y_{nbr} before absorption. The cost of absorption to update the pointers of Y_{nbr} is equal to d.

We associate a charge with every node to account for part of the work done during the absorption. Eventually the sum of the charges in the (existing) nodes account for the total work done for absorptions. Let Y be a node which is absorbed into Y_{nbr} at some point in the sequence of deletions. The amount of work done for this absorption is the number of children of Y (which now become the children of Y_{nbr}) to update the child pointers of Y_{nbr}. We account for this by adding a charge of d to the node Y_{nbr}. In addition, we pass the charge accumulated in Y to Y_{nbr}. So the new charge at Y_{nbr} is the old charge in that node plus the charge at Y plus d.

We first claim that the charge accumulated at any node with degree d is at most d^2. If this was true before, then the new charge at Y_{nbr} is at most its old charge plus $d + d^2$ (as the charge at Y was at most d^2 by induction hypothesis and its degree is at most d). The old charge at Y_{nbr} by induction hypothesis is at most d_{nbr}^2, and its new degree is $d + d_{nbr}$. The new charge is at most $d_{nbr}^2 + d^2 + d \leq (d + d_{nbr})^2$ which proves the claim.

Hence the total charge on the existing nodes at any point of time is at most $4n^2$ as the sum of the degrees is at most $2n$. We spend another $O(k^2)$ time for each update to update the nodes corresponding to every pair of vertices in the bag that got split. Thus, the amortized time for Δ edge deletions is $O(n^2/\Delta + k^2)$. □

We can, in $O(nk^2)$ time, construct a data structure such that given a sequence of $\Omega(n^2/k^2)$ edge deletions, we can support deletion query in $O(1)$ time and deletion update in $O(k^2)$ total time. In particular if the graph has at least $m = \Omega(n^{3/2})$ edges and the size of the maximum clique is $O(n^{1/4})$, then we have an $O(1)$ query and $O(n^{1/2})$ update time.

3 Dynamic Maintenance of Perfect Elimination Ordering

3.1 Decremental Algorithm

We now give a decremental algorithm using perfect elimination ordering (PEO). Towards that we first state the following characterization.

Lemma 2 [14]**.** *Let G be a chordal graph, and let $e = (u, v)$ be an edge. $G \backslash (u, v)$ is chordal if and only if all the common neighbors of u and v are adjacent to each other, i.e., they form a clique.*

Using the above characterization and the well-known dynamic list to represent the PEO, we obtain the following result.

Theorem 4. $\star$[1] *Let G be a chordal graph represented by its adjacency list and adjacency matrix. We can, in $O(m + n)$ time, construct a PEO of G, such that whenever an edge (u, v) is deleted, we can determine if $G \backslash (u, v)$ is chordal in $O(\min\{degree(u), degree(v)\})$ time, and update the structures if it is the case, in $O(degree(u) + degree(v))$ time.*

If the chordal graph has bounded degree, we get the following.

Corollary 1. *Let G be a chordal graph with bounded degree given by its adjacency matrix and adjacency list. We can in $O(m + n)$ time construct a PEO of the vertices of G such that whenever an edge (u, v) is deleted, we can in $O(1)$ time determine if $G \backslash (u, v)$ is chordal and if yes, we can update the structure in $O(1)$ time.*

[1] Proof deferred to the full version.

3.2 Fully Dynamic Maintenance of PEO

We show how to convert a clique tree decomposition of a chordal graph to its PEO ordering in $O(n)$ time even under edge insertions and deletions, thus answering a question posed by Ibarra [12]. In $O(m + n)$ time we can obtain a clique tree from a graph G as well as store the intersection values between every pair of adjacent nodes in a clique tree [17,18]. In addition, we show that for the lists associated with two adjacent nodes, defined as A and B, we can also store the values $A \backslash B$ and $B \backslash A$ and update them on edge addition and deletion efficiently. This added information helps to convert from a clique tree to PEO efficiently in $O(n)$ time. Using the help of the following lemma (proof in appendix) we show how to maintain this structure for edge additions/deletions.

Lemma 3. $\star$[2] *Let G be a chordal graph given with its clique-tree decomposition. We can construct a data structure in $O(nk \log k)$ time (where k refers to the maximum node size in the clique tree) and store the vertices differing between two adjacent nodes in the clique tree i.e. for the lists A and B associated with the two adjacent nodes, we store the values $A \backslash B$ and $B \backslash A$ on the edge connecting the nodes. We can update this structure and the clique-tree in $O(n)$ time on addition or deletion of an edge from the graph G.*

Using this lemma, we look at converting a clique tree to a PEO ordering efficiently. The proof of the following theorem gives details of implementation in $O(n)$ time.

Theorem 5. *Let G be a chordal graph given with its clique-tree decomposition. We can augment it in $O(nk \log k)$ time (where k refers to the maximum node size in the clique tree) such that we can convert the clique tree to a PEO in $O(n)$ time on addition or deletion of an edge provided the modified graph remains chordal.*

Proof. Let A and B denote the lists associated with two adjacent nodes. Define $P = A \backslash B$ and $Q = B \backslash A$. Using Lemma 3 we construct and store the clique tree data structure augmented with the sets P and Q on each edge. Arbitrarily root the clique tree and do a depth first search traversal ordered by the start times of the nodes. We take all the vertices in that node and push it into a stack. We continue our DFS traversal and whenever we arrive at a node A, we push the vertices $A \backslash B$ into the stack, where B is its parent. At the end of the traversal, pop the vertices from the stack. The order in which they are popped is the order of the PEO.

For correctness we need to show that this traversal maintains the PEO at any time instant. Initially when we consider the root node, they form a maximal clique, so pushing them in any order in the stack does not violate the PEO property. Let us take a node A at some intermediate step of the traversal and push $A \backslash B$ into the stack. For a vertex $a \in A \backslash B$ to violate the PEO ordering, a has to be a neighbor of b and c, two vertices already in the stack below a but

[2] Proof deferred to the full version.

b and c are not adjacent to each other. But this cannot happen. As b and c are already in the stack they have been visited earlier in the traversal. We show b and c are both in node A. If not, they cannot appear after A in the traversal as it violates the connected subtree property. A is the first node in the traversal which contains a and as (a, b) and (a, c) are neighbors they have to appear in some node by definition of clique trees. Therefore, the vertices b and c are both in node A as well. As each node is a maximal clique, vertices b and c are also neighbors. So we see that the algorithm does not violate the PEO ordering.

During the traversal we push each vertex into the stack only once and pop them out once. Hence, the total time spent is $O(n)$. □

4 Lower Bound

We first observe that the reduction [16] from the Query-Sum problem to dynamic connectivity also holds for fully dynamic connectivity on forests to show the following.

Theorem 6 [16]**.** *Consider any dynamic data structure that performs a sequence of n edge insertions and deletions that maintain the forest structure starting from an edgeless graph. Suppose the structure also supports queries of the form whether a pair of vertices are in the same connected component. Then such a structure requires $\Omega(\log n)$ amortized time per query and update to support a sequence of n query and update operations in the cell probe model of word size $\log n$.*

We use this observation to give a reduction to our problem to prove a similar lower bound.

Theorem 7. *Any dynamic structure that maintains a chordal graph under edge insertions and deletions requires $\Omega(\log n)$ amortized time per update or query in the cell probe model of word size $\log n$.*

Proof. The main idea is to ensure that when a query for a pair (u, v) comes, we add a new path of length three between u and v and check whether the resulting graph is chordal. If the pair of vertices are in different components, then the new additions don't add any cycle, and if they are in the same component, then new additions create a chordless cycle of length greater than three. Hence we can test the reachability question using the chordality query. We give the details below.

Given an instance I of the fully dynamic connectivity problem on forests with n vertices, we create a graph on $n + 2$ vertices where the first n vertices correspond to the original vertices, and there are two new vertices s and t with an edge between s and t. Whenever an edge $\{u, v\}$ is added to I, we call the addition of edge $\{u, v\}$ to I'. Whenever an edge $\{u, v\}$ is deleted from I, we delete the same edge from I'. The forest maintenance property of the instance I ensures that these addition or deletion of edges always ensures a forest is maintained in I' as well.

When a query between a pair of vertices u and v comes, we simply add the edges $\{u, s\}$ and $\{t, v\}$ and ask whether the resulting graph is chordal. If it is, then we declare that u and v are in different components of the forest, and otherwise they are in the same component. We then delete the edges $\{u, s\}$ and $\{t, v\}$ from the graph. If u and v are in the same component, then the path in the component between u and v along with edges $\{u, s\}, \{s, t\}\, and\, \{t, v\}$ form a chordless cycle. This proves the correctness of the reduction.

Thus every connectivity query in I is implemented by two edge additions, a chordality query and two edge deletions in I'. Furthermore, every update in I is implemented by the same update in I'. Thus from Theorem 6, the theorem follows.

We observe that the only property of chordal graphs we used in the above reduction is that trees are chordal and any induced cycle of length greater than three is not chordal. Hence the same reduction works for any subclass of chordal graphs that contains the class of trees. Thus we have

Corollary 2. *Any dynamic structure that maintains a Ptolemaic graph or a k-tree or a strongly chordal graph (for definitions refer* [2,11,15]*) under edge insertions and deletions requires $\Omega(\log n)$ amortized time per update or query.*

5 Conclusions

We have presented improved upper and lower bounds for maintaining chordal graphs under edge deletions and insertions. graphs. We also showed that we can shift between different decompositions of chordal graphs in $O(n)$ time which helps to solve applications that require different decompositions. An interesting open problem is to prove a super logarithmic lower bound for the query and update operations for maintenance of chordal graphs. We have given a structure to maintain a PEO under edge insertions and deletions in $O(n)$ time by augmenting the clique tree decomposition. It would be an interesting problem to see if the optimization problems (like maximum clique and independent set) that use PEO can be updated in $O(n)$ time under edge insertions and deletions.

Acknowledgement. The first author would like to thank Keerti Choudhary for useful discussions leading to Theorem 5.

References

1. Berry, A., Sigayret, A., Spinrad, J.: Faster dynamic algorithms for chordal graphs, and an application to phylogeny. In: Kratsch, D. (ed.) WG 2005. LNCS, vol. 3787, pp. 445–455. Springer, Heidelberg (2005). https://doi.org/10.1007/11604686_39
2. Brandstädt, A., Dragan, F.F., Chepoi, V., Voloshin, V.I.: Dually chordal graphs. SIAM J. Discrete Math. **11**(3), 437–455 (1998)
3. Buneman, P.: A characterisation of rigid circuit graphs. Discrete Math. **9**(3), 205–212 (1974)

4. Deshpande, A., Garofalakis, M.N., Jordan, M.I.: Efficient stepwise selection in decomposable models. In: UAI 2001: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, 2–5 Aug 2001, pp. 128–135 (2001)
5. Diestel, R.: Graph Theory. GTM, vol. 173, 4th edn. Springer, Heidelberg (2012)
6. Dietz, P.F., Sleator, D.D.: Two algorithms for maintaining order in a list. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing 1987, New York, NY, USA, pp. 365–372 (1987)
7. Fagin, R.: Degrees of acyclicity for hypergraphs and relational database schemes. J. ACM **30**(3), 514–550 (1983)
8. Gavril, F.: Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. SIAM J. Comput. **1**(2), 180–187 (1972)
9. Hell, P., Shamir, R., Sharan, R.: A fully dynamic algorithm for recognizing and representing proper interval graphs. SIAM J. Comput. **31**(1), 289–305 (2001)
10. Henzinger, M.R., Fredman, M.L.: Lower bounds for fully dynamic connectivity problems in graphs. Algorithmica **22**(3), 351–362 (1998)
11. Howorka, E.: A characterization of ptolemaic graphs. J. Graph Theory **5**(3), 323–331 (1981)
12. Ibarra, L.: Fully dynamic algorithms for chordal graphs and split graphs. ACM Trans. Algorithms **4**(4), 40:1–40:20 (2008)
13. Mezzini, M.: Fully dynamic algorithm for chordal graphs with O(1) query-time and O(n^2) update-time. Theor. Comput. Sci. **445**, 82–92 (2012)
14. Mezzini, M., Moscarini, M.: Simple algorithms for minimal triangulation of a graph and backward selection of a decomposable markov network. Theor. Comput. Sci. **411**(7–9), 958–966 (2010)
15. Nesetril, J.: Structural properties of sparse graphs. Electron. Notes Discrete Math. **31**, 247–251 (2008)
16. Patrascu, M., Demaine, E.D.: Logarithmic lower bounds in the cell-probe model. SIAM J. Comput. **35**(4), 932–963 (2006)
17. Rose, D.J., Tarjan, R.E., Lueker, G.S.: Algorithmic aspects of vertex elimination on graphs. SIAM J. Comput. **5**(2), 266–283 (1976)
18. Tarjan, R.E., Yannakakis, M.: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM J. Comput. **13**(3), 566–579 (1984)
19. Walter, J.R.: Representations of chordal graphs as subtrees of a tree. J. Graph Theory **2**(3), 265–267 (1978)
20. Yao, A.C.-C.: Should tables be sorted? J. ACM **28**(3), 615–628 (1981)