

Faster Dynamic Algorithms for Chordal Graphs, and an Application to Phylogeny

Anne Berry¹, Alain Sigayret¹, and Jeremy Spinrad²

¹ LIMOS UMR, bat. ISIMA, 63173 Aubière cedex, France
{berry, sigayret}@isima.fr

² EECS dept, Vanderbilt University, Nashville, TN 37235, USA
spin@vuse.vanderbilt.edu

Abstract. We improve the current complexities for maintaining a chordal graph by starting with an empty graph and repeatedly adding or deleting edges.

1 Introduction

Our motivation for this paper stems from the biology-based problem of improving the matrix representing an evolutionary tree (phylogeny) which contains errors. To solve this, Berry, Sigayret and Sinoquet in [2] needed to start with an independent set and repeatedly add an edge of minimum weight, while maintaining a chordal graph. However, the most efficient algorithms for dynamically maintaining a chordal graph (see Ibarra [8]) were insufficient to ensure a complexity which could be used in practice.

In this paper, we improve the time complexity for dynamic algorithms for chordal graphs. Ibarra studied the problem of maintaining a chordal graph as edges are inserted or deleted. Operations considered were insert, delete, insert query and delete query; the last two operations ask whether deletion/addition of a given edge xy preserves chordality. He gave three implementations of the algorithm. In the first implementation, all operations take $O(n)$ time. In the second, deletions take $O(n \log n)$ time, deletion queries and insertion run in $O(n)$ time, while insertion queries run in $O(\log^2 n)$ time. The third variant was designed for sparse chordal graphs and will not be addressed in this paper.

Our new running times are $O(n)$ for insertion and deletion, $O(1)$ for insertion queries, and $O(n)$ for deletion queries. All data structures used are simple. We do make one extra assumption, which is not made in the Ibarra paper. We assume that we start with an empty graph; if this assumption is not made, there is an initial start-up cost of $O(m\Delta)$, where Δ is the maximum degree of a vertex, or $O(n^\alpha)$, where n^α is the cost of doing matrix multiplication. We use this result to improve the time bound for the original phylogeny problem, from $O(n^4)$ to $O(n^3)$.

2 Preliminaries

In this paper, $G = (V, E)$ will be a graph with n vertices and m edges. We will use non-formal notations such as $G - S$ instead of $G(V - S)$. An xy -separator in

a connected graph G is a non-empty set S of vertices such that there is no path from x to y in $G-S$. S is a minimal xy -separator if S does not properly contain any xy -separator. Whenever there exists a pair $\{x, y\}$ of vertices such that S is a minimal xy -separator, S is called a minimal separator.

A graph is *chordal* if every cycle of length greater than three has a chord. Chordal graphs have a long history of study; see, for example [1,6]. A *clique tree* of a chordal graph G is a tree T such that nodes have a 1-1 correspondence with maximal cliques of G , edges correspond to non empty intersections of pairs of maximal cliques, and for all vertices v in G , the set of maximal cliques which contain v induces a subtree of T . A graph is chordal iff it has a clique tree ([5], [3], [12]). Each edge $S = K_1 \cap K_2$ of T corresponds to one minimal separator S of G ; conversely, each minimal separator of G is represented by at least one edge of T . A chordal graph may thus have several clique trees. A clique tree has $O(n)$ nodes. There are known algorithms (see for example [10]) which find a clique tree of a chordal graph in $O(m+n)$ time.

The discussions in this paper will use the following theorems which are easy to prove using well-known results on chordal graphs and a characterization from [2]:

Characterization 1. ([2]) *Let $G = (V, E)$ be a chordal graph, $xy \notin E$. Then $G + xy$ is chordal iff $\{x, y\}$ is a 2-pair of G (i.e. all chordless paths between x and y are of length 2).*

Theorem 2. *Edge xy can be deleted from a chordal graph G without causing a chordless cycle iff x and y are not together in any minimal separator of G .*

Theorem 3. *Edge xy can be added to a connected chordal graph G without causing a chordless cycle iff x and y are both adjacent to every vertex in some minimal xy -separator S . Furthermore $S = N(x) \cap N(y)$.*

Theorem 4. *The number of maximal cliques containing a vertex x in a chordal graph is at most $|N(x)|$. The total number of vertices in all maximal cliques of a clique tree is $O(m)$.*

3 Data Structures

We maintain a clique tree of the current chordal graph G with the following modifications. For each maximal clique K and minimal separator S in the clique tree, and each vertex x , we keep variables $neighnum(x, K)$ and $neighnum(x, S)$ denoting the number of neighbors of x in K and S respectively.

We also maintain an array *Insertable*; $Insertable(x, y) = 1$ iff xy can be inserted while maintaining chordality.

We will discuss how to calculate initial values if we are given a start graph G after Theorem 7. If we start with an edgeless graph, all values are initially 0.

4 Algorithms

We now discuss how to implement the various operations. Some of these operations are identical to those in [8], but are repeated here so that the reader can have easy access to the full algorithm.

The simplest primitive, given our data structures, is **Insert Query**. We look up in our array *Insertable* whether $Insertable(x, y) = 1$. The other simple primitive to describe is **Delete Query**; to achieve an $O(n)$ bound, step through the tree and test whether more than one maximal clique contains both x and y . If it is desired, this can be reduced to $O(\min\{degree(x), degree(y)\})$ by using Theorem 4 and letting each vertex x keep a list pointing to each clique which contains x .

Operations **Insert** and **Delete** are very similar to each other. In each case, we modify the clique tree and then update the array *Insertable* to decide which edges may now be added to the graph while preserving chordality.

We first deal with the insertion of xy .

The first part of the modification, finding a clique tree for $G + xy$, may work exactly as in Ibarra's paper, though we must make sure to update our new data structures as well. For completeness, we will give the full process here.

Let xy be the edge to be inserted. There will be exactly one new maximal clique, which is $y + x + (N(x) \cap N(y))$. At most 2 maximal cliques of G are deleted, which are $x + (N(x) \cap N(y))$ and $y + (N(x) \cap N(y))$ if these cliques are currently maximal in G . For simplicity, we will treat the cases of 2, 1 or 0 of $x + (N(x) \cap N(y))$, $y + (N(x) \cap N(y))$ existing in the clique tree separately. To determine whether one or both of these cliques are in the current tree, let K_1 be any maximal clique containing x and K_2 be any maximal clique containing y . We find the path from K_1 to K_2 in the clique tree. Let K_x be the last clique on this path which contains x , and let K_y be the first clique on the path which contains y . It is not hard to see that if $x + (N(x) \cap N(y))$ is in the tree, then it must be K_x and if $y + (N(x) \cap N(y))$ is in the tree, it must be K_y . Thus, we can determine which cliques appear and disappear with the addition of edge xy in $O(n)$ time.

- Case 1. both $x + N(x) \cap N(y)$ and $y + N(x) \cap N(y)$ are maximal cliques of G . There is a separator S_{xy} on the path from $K_x = x + N(x) \cap N(y)$ to $K_y = y + N(x) \cap N(y)$ which is exactly equal to $N(x) \cap N(y)$, else xy could not have been inserted while maintaining chordality. We delete this edge and unify the two nodes representing K_x and K_y into node $K_{xy} = x + y + (N(x) \cap N(y))$ as in Figure 1.
- Case 2. we now consider the case in which one of the two possible maximal cliques ceases to be maximal due to the addition of xy ; w.l.o.g., let us assume that $K_x = x + (N(x) \cap N(y))$ is a maximal clique of G .

Let K_y be the first clique containing y on the path from K_1 to K_2 ; as noted earlier, $K_x = x + (N(x) \cap N(y))$ will be the last clique containing x on this path. We add an edge from the K_x to K_y , and delete edge $N(x) \cap N(y)$ on the path from x to y . We then add y to K_x as in Figure 2.

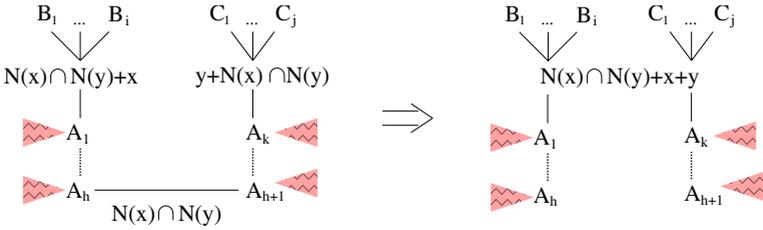


Fig. 1. Insert — case 1

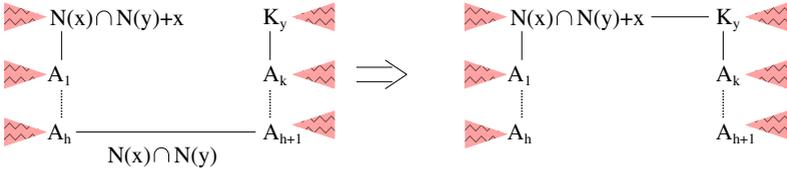


Fig. 2. Insert — case 2

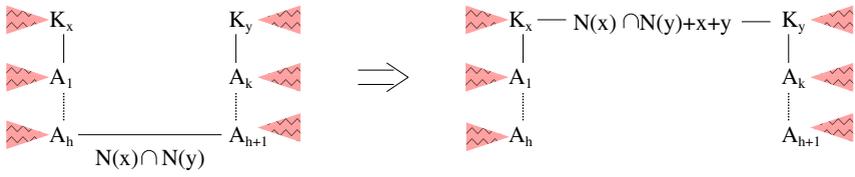


Fig. 3. Insert — case 3

- Case 3. in this case, we add a new node to the clique tree corresponding to $x + y + (N(x) \cap N(y))$, and add edges from this node to K_x and K_y , and delete S_{xy} as in Figure 3.

We need to update the variables $neighnum(v, K)$ and $neighnum(v, S)$ for each maximal clique K and minimal separator S of the tree to reflect the changes caused by insertion of xy . For each clique K and separator S containing y , add 1 to $neighnum(x, K)$ and $neighnum(x, S)$; similarly, increment the values of $neighnum(y, K)$ and $neighnum(y, S)$ for each clique or separator containing y . We have at most one new maximal clique K_{xy} in the tree: $x + y + (N(x) \cap N(y))$; for each vertex v , we let $neighnum(v, K_{xy}) = neighnum(v, S_{xy})$ plus the number of neighbors of v in $\{x, y\}$.

We now have to update the values for separators which were changed by the insertion of xy . Note that in case 1, no separators change, in case 2 a single separator (which goes between K_x and K_y) has a vertex added to it, and in case 3 two separators are added corresponding to the edges around the new node of the clique tree.

The only separators changed are those adjacent to the new maximal clique $x + y + (N(x) \cap N(y))$, and correspond to $N(x) \cap N(y)$ plus possibly a vertex from $\{x, y\}$. Since we know the number of neighbors of each vertex w.r.t. $N(x) \cap N(y)$, it is easy to calculate the number of neighbors of each vertex w.r.t. to the new separators in constant time.

The array `Insertable` must be updated after addition of edge xy . We will defer discussion of this step until after we discuss deletion of an edge, since both steps make use of a routine which takes an input vertex x and a clique tree, and finds all z such that xz is insertable in $O(n)$ time.

We will now examine deletion.

The deletion of an edge xy causes the maximal clique $x + y + (N(x) \cap N(y))$ to disappear from the clique tree. At most two new maximal cliques may appear: $x + (N(x) \cap N(y))$ and $y + (N(x) \cap N(y))$. As in the case of insertion, we discuss the cases of 0, 1 or 2 maximal cliques appearing separately. As observed by Ibarra [8], it is easy to determine whether each of these cliques appears in $O(n)$ time. We find the single maximal clique containing x and y , and look for edges from this node which correspond to separators containing $1 + |N(x) \cap N(y)|$ vertices; we then test whether x or y is the vertex missing in such a separator.

- Case 1. Two maximal cliques appear: $x + (N(x) \cap N(y))$ and $y + (N(x) \cap N(y))$. In this case, the tree node corresponding to $x + y + (N(x) \cap N(y))$ is split into 2 adjacent nodes $x + (N(x) \cap N(y))$ and $y + (N(x) \cap N(y))$. The neighboring cliques Y_* containing y are made neighbors of $y + (N(x) \cap N(y))$, other cliques which were neighbors of $x + y + (N(x) \cap N(y))$ become neighbors of $x + (N(x) \cap N(y))$ in the new tree, as in Figure 4.
- Case 2. If one maximal clique, which we will w.l.o.g. assume is $x + (N(x) \cap N(y))$ appears, the following changes are made.

Let K_y be the neighbor of $x + y + (N(x) \cap N(y))$ which is separated by an edge separator with $1 + |N(x) \cap N(y)|$ vertices. We remove y from the node $x + y + (N(x) \cap N(y))$ and for every neighbor Y_* of $x + y + (N(x) \cap N(y))$ containing y in the clique tree except for K_y , we remove the connection from Y_* to $x + y + (N(x) \cap N(y))$ and add an edge from Y_* to K_y , as in Figure 5. Note that since none of these Y_* correspond to cliques containing x , the separator between Y_* and K_y is the same as the old separator between K_y and $x + y + (N(x) \cap N(y))$.

- Case 3. In the remaining case, no new maximal clique appears; we remove $x + y + (N(x) \cap N(y))$ from the clique tree, then we find K_y and an analogous K_x as in the previous case; these are cliques which contain $y + N(x) \cap N(y)$ and $x + N(x) \cap N(y)$ respectively. We add an edge between K_x and K_y . All former neighbors of $x + y + (N(x) \cap N(y))$ containing y are given edges to K_y , while other neighbors are given edges to K_x , as in Figure 6. Since no remaining clique contains both x and y , the separators remain the same in the new clique tree, except for the edge separating K_x and K_y .

We now describe how to modify the variables maintained after the modification of the clique tree.

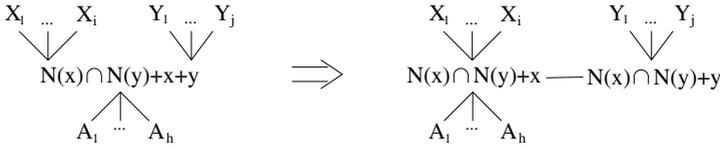


Fig. 4. Delete — case 1

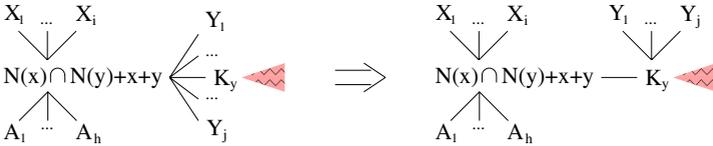


Fig. 5. Delete — case 2

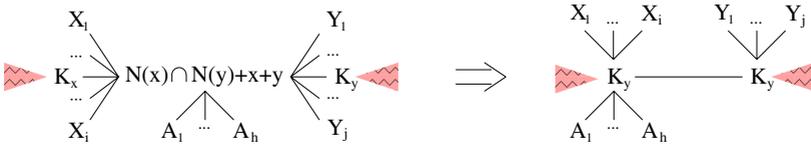


Fig. 6. Delete — case 3

The only new maximal cliques to appear are either $x + (N(x) \cap N(y))$ or $y + (N(x) \cap N(y))$. Since we know the number of neighbors in each vertex in $x + y + (N(x) \cap N(y))$, it is easy to compute the number of neighbors in the new clique in constant time per vertex. The only new separator which could have been created was $N(x) \cap N(y)$. Again, it is easy to compute the number of neighbors of each vertex w.r.t. the new separator $N(x) \cap N(y)$, since we know the number of its neighbors in $x + y + N(x) \cap N(y)$.

We now come to the updating of the array *Insertable* after deletion or addition of the edge xy :

Theorem 5. *Let G be a connected chordal graph. Let x, y, v and w be vertices such that neither v nor w is equal to x or y .*

1. *If $G+xy$ is chordal: vw can be inserted into $G+xy$ while preserving chordality iff vw could be added to G while preserving chordality.*
2. *If $G-xy$ is chordal: vw can be inserted into $G-xy$ while preserving chordality iff vw could be added to G while preserving chordality.*

Proof.

1. xy is not an edge of G . G and $G + xy$ are chordal.
 $\implies (G+xy)+vw$ is chordal. Suppose $G + vw$ is not chordal: there exists a chordless cycle $C = v \sim w \sim v$. In $G + xy + vw$, which is chordal, C

does not remain chordless and this must be due to chord xy . Then x and y are in C which is w.l.o.g. $v-s-w\sim x\sim y\sim v$. As a consequence, cycle $v-s-w\sim x-y\sim v$ is chordless in $G + xy + vw$ — a contradiction.

\Leftarrow $G+vw$ is chordal. By Theorem 3 there exists a minimal vw -separator $S = N(v) \cap N(w)$. The addition of edge xy does not change the common neighborhood of v and w ; then there exists in $G + xy$ a minimal vw -separator $S' \supseteq S$. Suppose $S' \neq S$; then there exists a new path between v and w ; this path must use edge xy , and is w.l.o.g. $v\sim x-y\sim w$. As there is also a path $v-s-w$ for some $s \in S$, chordal graph $G + xy$ contains a chordless cycle of length ≥ 4 — a contradiction. Then $S' = S = N(v) \cap N(w)$ remains a minimal vw -separator in $G + xy$ and, by Theorem 3, vw is insertable in $G + xy$.

2. Apply part 1 of this theorem with $G' = G + xy$ and thus $G = G' - xy$. \square

Given the above theorem, we only need to find which vertex pairs $\{x, z\}$ and $\{y, z\}$ can be inserted to preserve chordality, and update these values in the array *Insertable*.

We give an algorithm which takes an arbitrary single vertex v and finds all non-neighbors w of v such that vw can be inserted and preserve chordality in $O(n)$ time given the information maintained on the clique tree. By Theorem 5, we can simply run this for x and y when xy is inserted or deleted, and we will have updated our *Insertable* list correctly.

Theorem 6. *Given a vertex v , we can find all vertices w such that vw can be inserted while preserving chordality in $O(n)$ time.*

Proof. The following algorithm takes a vertex v and finds all w such that adding vw to G will preserve chordality.

For each non-neighbor w of v , place w on the clique tree at any clique which contains w . We select any clique containing v , and traverse the clique tree in a depth-first fashion. When a non-neighbor w is reached on the clique tree, we will decide whether adding vw to the tree would preserve chordality.

We keep one extra data structure during our traversal. Recall that vw can be added iff w and v are both completely adjacent to some minimal vw -separator. The extra structure is an array *posssep* of size n , which holds pointers to possible separators S such that S might be a minimal vw -separator meeting the criteria of Theorem 2 for some w we may encounter on the path. Initially, *posssep* is empty.

Suppose that we are at clique node K in our traversal of T , and our DFS traversal of T leaves K by an edge corresponding to separator S to a new node K' of T .

We test using $neighnum(v, S)$ whether v is adjacent to all vertices of S . If $neighnum(v, S) = |S|$, then S could be a possible vw -separator meeting the conditions of Theorem 2. We look at position $|S|$ of *posseps*. If this is already non-empty, then there is already a separator S' with the same vertices as S encountered earlier than S on the path from x . In this case, if S meets the conditions of Theorem 2, S' also meets the conditions of Theorem 2 for any w

encountered on the path, and S is not stored in *posseps*. If position $|S|$ is empty, we mark this edge e as a candidate separator, and put a pointer in *posseps*($|S|$) to edge e . We add 1 to a count of the number of separators in *posseps*, and if this number of separators in *posseps* becomes 1, we call this separator *startsep*.

Suppose that we encounter a vertex w which was placed on the clique tree. We want to test whether v and w are separated by any vw -separator S such that both v and w are completely adjacent to S .

It is not hard to see that all minimal vw -separators are on the path from v to the current node, though (since w may be in many cliques on this path) not all edges on the path correspond to minimal vw -separators. We test how many neighbors of w there are in separator *startsep*; call this number *vwneighbors*. Clearly, any separator with fewer than *vwneighbors* vertices cannot separate v from w . In addition, any separator with more than *vwneighbors* vertices cannot be completely adjacent to w , since neighbors of v can only disappear as we traverse the path from v to w . Thus, we only need to check if the edge pointed to by *posseps*(*vwneighbors*) is a minimal vw -separator satisfying the conditions of Theorem 2. If the array points to edge $K_i - S - K_j$, with K_i closer to v , we check that w is not in K_i (or this would not be a vw -separator), and that $\mathit{neighnum}(w, S) = |S|$.

We make $\mathit{Insertable}(v, w) = 1$ if these conditions hold, and 0 otherwise.

As we back up across an edge $e = K_i - S - K_j$ in the DFS, if e is marked as a separator, we delete the pointer in *posseps*($|S|$), and decrement the number of current possible separators. \square

Combining the theorems above, we get the desired result.

Theorem 7. *The algorithm maintains a chordal graph under the operations insert, delete, delete query, and insert query, taking $O(n)$ time for the first three operations and $O(1)$ time for insert query.*

If a connected graph G is given as input, it is necessary to construct a clique tree, and compute $\mathit{neighnum}(v, S)$ and $\mathit{neighnumN}(v, K)$ for each vertex v , each minimal separator S , and each clique K of the clique tree.

As a clique tree of a chordal graph can be found in $O(m+n)$ time and has an $O(m)$ overall number of vertices in the nodes, we can step through all maximal cliques K , and for each v in K add 1 to $\mathit{neighnum}(w, K)$ for all neighbors w of v , thus finding all these variables in $O(mn)$ time.

Although there are also $O(m)$ vertices over all minimal separators of the clique tree (so the argument above could also be used to count all variables in $O(mn)$ time), the algorithms for finding a clique tree of a chordal graph usually do not explicitly label the separators. Therefore, we describe briefly how these separators could be labeled in $O(m+n)$ time. Choose an arbitrary root for the clique tree. For each node K of the clique tree, perform the following operation. Mark all positions of an array which are vertices of K . For each child K' of K , step through all vertices of K' , putting them on the separator between K and K' if these are marked in the array. Since vertices of each maximal clique K are

traversed at most twice (once when K is a parent, and once when K is a child), the total time spent constructing separators is $O(m+n)$.

Thus, all initial variables can be computed in $O(mn)$ time. Alternatively, we can use matrix multiplication to get the variables. Construct a graph G' with a vertex for each minimal separator S , a vertex for each maximal clique K , and two copies v_1 and v_2 of each vertex v of the graph. Add an edge from v_1 to v_2 if and only if v and w are adjacent in G , and an edge from v_2 to K or S if and only if v is in this maximal clique or minimal separator. The number of neighbors of v in K (or S) in G is the number of paths of length two from v_1 to K (or S) in G' . It is well known that the number of paths of length two from i to j is equal to $M^2[i, j]$, where M is the adjacency matrix of the graph. Since G' has $O(n)$ vertices, all variables can be constructed in $O(n^\alpha)$ time, where α is the coefficient of n in a matrix multiplication algorithm. The best known bound for α is 2.376 [4], and the variables can be computed in this time bound if one is willing to allow the (complex) algorithms used for matrix multiplication in that paper.

Recall that our algorithm assumes that our initial graph is edgeless. We thus need to deal with a chordal graph which is not connected. We begin with a forest of elementary clique trees. While the graph is not connected, the insertability of xy is determined by first testing whether x and y are in the same connected component of G . If not, xy is insertable, and the corresponding cliques trees will be merged (process similar to Case 1 of insertion, but without initial edge $N(x) \cap N(y)$); clearly, the only changes we have then to perform on array *Insertable* are w.l.o.g. $Insertable(x, z) = 1$ for each neighbor z of y . For operation Delete, when the graph is disconnected by an edge deletion, its clique tree is split (by deleting edge $K_x - K_y$ in Case 1 or 3 of deletion) and the child trees are managed separately.

5 Phylogeny

We now discuss the problem of efficient construction of a chordal graph by repeatedly adding a minimum weight edge xy to a current chordal graph G such that $G + xy$ is chordal. The fastest known algorithm for this problem runs in $O(n^4)$ time ([2]); we will reduce the time complexity to $O(n^3)$. The previous algorithm relied on Characterization 1 and used the algorithm ([11]) for maintaining all 2-pairs in a general graph as edges are added in $O(n^4)$ overall time. In this paper, we show that new 2-pairs can be found more efficiently if the graph is known to be chordal.

The problem discussed in this section arose in the context of computational biology. The problem, called phylogeny, involves reconstructing an evolutionary tree, given genetic information of modern species. A correspondence between the phylogeny problem and chordal graphs was first noted in [7].

To summarize very briefly the work most relevant to this paper, one can construct a matrix computing phylogenetic dissimilarity between different pairs of species. If we assume that this matrix is an ‘additive tree distance’ (which

corresponds to the notion that if species A branches off from B and C, and then later B branches off from C, the phylogenetic difference between A and C is equal to the difference between A and B plus the difference between B and C), then the graph G_i formed by including all edges with difference less than each threshold i will be a chordal graph ([7]). In practice, [7] found that the data tends to give graphs which are not chordal, but are "almost" chordal. We want to take this data, and modify it as little as possible to get a chordal graph. [2] proposes an edge addition scheme, starting from an edgeless graph, and repeatedly adding the smallest weight edge which preserves chordality, as an effective way of processing the phylogenetic data.

In that paper, an $O(n^4)$ algorithm was given to solve the problem. We use the results of the previous section to reduce the time complexity to $O(n^3)$. Recall from Theorem 5 that when an edge xy which maintains chordality is added, the only pairs which can change status as far as eligibility for addition are pairs containing x or y .

Therefore, we can maintain a list of edges eligible for addition to the structure, and there will be at most n^3 modifications of the list throughout the running of the algorithm. If the list of eligible edges is stored in increasing order of weight, we will simply choose the first eligible edge for addition at any step, and use the algorithm of the previous section to determine which changes to make in the list in $O(n)$ time.

If the list is stored as a balanced tree, additions and deletions can be made in $O(\log n)$ time, leading to an $O(n^3 \log n)$ algorithm for finding the order in which edges will be added. We will show how to accomplish the same task in $O(n^3)$ time.

Theorem 8. *We can find the complete sequence in which we will add minimum weight edges which preserve chordality in $O(n^3)$ time.*

Proof. As a first step, we sort all possible pairs by weight, and label each pair xy with the position of xy in the sorted list. At each step, we want to find the eligible pair with smallest label to add to our graph.

Instead of keeping the entire list of eligible pairs sorted, we group the eligible pairs as follows. We maintain unordered lists L_i of eligible pairs with thresholds $i n + 1$ through $(i + 1) n$ for each i from 0 to $n - 1$; i.e., we keep a list of eligible edges with thresholds in the ranges $[1..n]$, $[n + 1..2n]$, ... $[n(n - 1) + 1..n^2]$. Each eligible pair xy has a pointer to the position of xy in the appropriate list.

To choose the next eligible edge for addition, step through the lists until we find some non-empty list. Since there are $O(n)$ lists, we can find the next eligible pair in $O(n)$ time. Once the appropriate list is found, examine all pairs in the list to find the smallest eligible pair. Since each list contains at most n elements, the total time to find the next eligible pair is $O(n)$.

Using the algorithm of the previous section, we can find all pairs which must be added and deleted from the list of eligible edges in $O(n)$ time. Since each modification clearly takes constant time using our data structures, the total time for adding an edge is $O(n)$.

Since there are $O(n^2)$ edges added, the total time taken to find the sequence of edge additions is $O(n^3)$. \square

6 Conclusion

This paper shows that if we start with an edgeless graph, we can maintain chordality as edges are added and deleted using $O(n)$ time for insertions, deletions, and delete queries, and constant time for insert queries. As an application, we show that a triangulation problem arising out of phylogeny can be solved in $O(n^3)$ time.

Acknowledgement

The authors thank Jens Gustedt for drawing their attention to this possible complexity improvement.

References

1. Brandstädt A., Le V. B., Spinrad J. P.: *Graph Classes: A Survey*. Society for Industrial and Applied Mathematics, Philadelphia (USA), (1999).
2. Berry A., Sigayret A., Sinoquet C.: Maximal Sub-Triangulation as Improving Phylogenetic Data. *Proceedings of JIM'03. Soft Computing – Recent Advances in Knowledge Discovery*, G. Govaert, R. Haenle and M. Nadif (eds), **1900**:01 (2005).
3. Buneman P.: A characterization of rigid circuit graphs. *Discrete Mathematics*, **9** (1974) 205–212.
4. Coppersmith D., Winograd S.: On the Asymptotic Complexity of Matrix Multiplication. *SIAM J. Comput.*, **11**:3 (1982) 472–492.
5. Gåvril F.: The intersection graphs of subtrees of trees are exactly the chordal graphs. *Journal of Combinatorial Theory B*, **16** (1974) 47–56.
6. Golumbic M. C.: *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, (1980).
7. Huson D., Nettles S., T. Warnow T.: Obtaining highly accurate topology estimates of evolutionary trees from very short sequences. *Proc. RECOMB'99, Lyon (France)*, (1999) 198–207.
8. Ibarra L.: Fully Dynamic Algorithms for Chordal and Split Graphs. *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, (1999) 923–924.
9. Kearney P., Hayward R., Meijer H.: Inferring evolutionary trees from ordinal data. *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, (1997) 418–426.
10. Spinrad J. P.: *Efficient Graph Representation*. Fields Institute Monographs 19. American Mathematics Society, Providence (RI, USA), (2003) 324p.
11. Spinrad J., Sritharan R.: Algorithms for Weakly Triangulated Graphs. *Discrete Applied Mathematics*, **59** (1995) 181–191.
12. Walter J. R.: *Representations of Rigid Circuit Graphs*. PhD. Dissertation, Wayne State University, Detroit (USA), (1972).