Succinct Data Structures for Chordal Graphs

J. Ian Munro
Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
imunro@uwaterloo.ca
https://orcid.org/0000-0002-7165-7988

Kaiyu Wu
Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
k29wu@uwaterloo.ca
https://orcid.org/0000-0001-7562-1336

Abstract

We study the problem of approximate shortest path queries in chordal graphs and give a $n \log n + o(n \log n)$ bit data structure to answer the approximate distance query to within an additive constant of 1 in $O(1)$ time.

We study the problem of succinctly storing a static chordal graph to answer adjacency, degree, neighbourhood and shortest path queries. Let G be a chordal graph with n vertices. We design a data structure using the information theoretic minimal $n^2/4 + o(n^2)$ bits of space to support the queries:

- whether two vertices u, v are adjacent in time $f(n)$ for any $f(n) \in \omega(1)$.
- the degree of a vertex in $O(1)$ time.
- the vertices adjacent to u in $(f(n))^2$ time per neighbour
- the length of the shortest path from u to v in $O(nf(n))$ time

2012 ACM Subject Classification Theory of computation → Shortest paths, Theory of computation → Data compression

Keywords and phrases Succinct Data Structure, Chordal Graph

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.67

Funding This work was supported by NSERC of Canada and the Canada Research Chairs Programme.

1 Introduction

Chordal graphs have a rich history of study. There were encountered in the study of Gaussian elimination of sparse matrices [15]. Chordal graphs have many equivalent characterizations including the absence of chordless cycles of length greater than 3, the existence of an perfect elimination order[16], the existence of a clique tree [4], and as the intersection graph of subtrees of a tree [18]. Tarjan et. al [16] gave a linear $O(n + m)$ algorithm for recognizing chordal graphs with n vertices and m edges by computing a perfect elimination order. The structure of chordal graphs allows the computation of many otherwise NP-Hard problems to be solved in polynomial time. These include finding the largest clique or computing the chromatic number. Chordal graphs have found applications in many fields, including compiler construction [14] and databases [6].

We consider the problem of creating a data structure for a chordal graph through the lens of *succinct* data structures. The goal of succinct data structures is to store a set X of objects in the information theoretic minimal $\log(|X|) + o(\log(|X|))$ bits of space while still being able to efficiently support the relevant queries. Jacobson [10] is the first to consider

space efficient data structures in this sense and he gave representations of bit vectors, trees and planar graphs. Further work in this area gave space minimal representations of dynamic trees [11], arbitrary graphs [8] and partial k-trees [7].

1.1 Related Work

Graphs are a fundamental combinatorical structure and it is no surprise that there are a lot of work in constructing space efficient data structure for different classes of graphs. Many classes of graphs have been considered, such as arbitrary graphs [8], partial k-trees [7], planar graphs [10] and separable graphs [2]. For chordal graphs, there has been work in the dynamic setting, focusing mainly on whether certain edge insertions/deletions preserve chordality [1, 9]. Banerjee et. al showed that insertions/deletions can be done in $O(\deg(u) + \deg(v))$ time where (u, v) is the edge that is inserted/deleted. They also show a lower bound that $O(\log n)$ amortized time is required.

Singh et. al [17] gave an $O(n \log n)$ bit data structure for the problem of approximate distance queries in chordal graphs. Their result is a $2d + 8$ approximation, that is, the result of the query is anywhere between d the actual distance and $2d + 8$.

1.2 Our Results

Our representation of a chordal graph is based on the clique tree [4]. We store a slight variation of the clique tree in the information theoretic minimal $n^2/4 + o(n^2)$ bits of space. We then augment this structure to support degree in $O(1)$, adjacency and neighbourhood in $O(f(n)), O(f(n)^2)$ respectively for any $f \in \omega(1))$ and distance queries in $O(nf(n))$. We then consider the problem of approximating the distance query and identify the necessary portions of the previous data structure required to answer this approximation to obtain a $n \log n + o(n \log n)$ bit data structure with $O(1)$ query time. The approximation is within 1 of the actual distance.

Finally we explore the close relationship between the distance query and the set intersection oracle problem, and show that heuristically, it is difficult to construct a data structure in the exact distance scenario.

2 Preliminaries

2.1 Graph Terminology

We will assume basic terminology from graph theory such as vertex, edge, tree, undirected graph, etc. We will denote an undirected graph as $G = (V, E)$ with vertex set V and edge set E. We will denote an edge between vertices u, v by (u, v). The number of vertices as $n = |V|$ and the number of edges as $m = |E|$. As we will be dealing with multiple graph-like structures at the same time, we will use V to denote the vertex set when the underlying graph is clear and $V(G)$ to denote the vertex set of graph G. To avoid confusion in discussing mapping a graph onto a tree, we will refer to vertices of trees as *nodes*. A clique of G is a complete subgraph of G. Unless otherwise stated, our log are base 2.

2.2 Chordal Graph Structure

A graph G is chordal if it does not contain any C_k, a cycle on k vertices as an induced subgraph for any $k \geq 4$. We will assume that all our chordal graphs are connected, or if not we could treat each component separately. The well known result of Rose et al. [16]

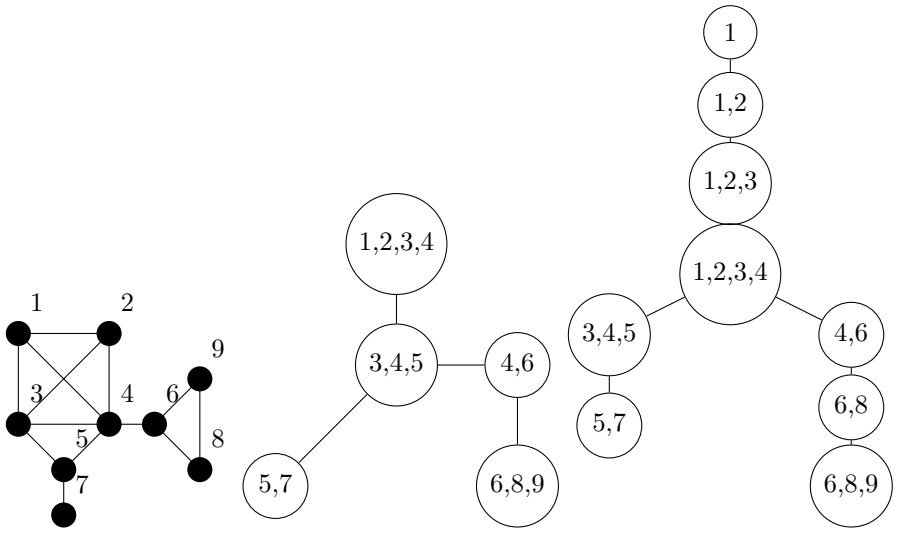

Figure 1 A chordal graph and a PEO of it labelled. A clique tree and the tree decomposition.

states that this is equivalent to the existence of a perfect elimination order (PEO) of the vertices of G. A PEO of a chordal graph G is an ordering $v_1, v_2, \ldots, v_n$ of V such that the predecessor set $\text{pred}(v_i) = \{v_j; j < i, (v_i, v_j) \in E\}$ is a clique for every vertex $v_i \in V$. For simplicity, we will denote v_i simply as i. Furthermore, one can construct a clique tree of G using the maximal cliques of G. Here every node of the tree is assigned a maximal clique and the tree has the property that for every pair of cliques K, K', $K \cap K'$ is contained in every clique along the path between the nodes corresponding to K, K'. This is equivalent to for every vertex $v \in V$, the set of cliques v belongs to forms a contiguous subtree.

We will use a variant of the clique tree that has n nodes constructed from the PEO, which we will denote as a *tree decomposition* of G. Let T be a tree and $X : V(T) \to 2^V$ a function that assigns to each node of T a subset of the vertices of G such that:

- For every $v \in V$, the set of nodes $X^{-1}(v)$ is non-empty and is contiguous. We will call this the contiguous subtree property.
- For every pair of vertices $u, v \in V$, (u, v) is an edge if and only if there is a tree node T_w such that $u, v \in X(T_w)$.

Note clique trees satisfies these properties.

Define $B(i) = \text{pred}(i) \cup \{i\}$ which we will call the *bag* of i. Define the functions $s(i) = \min(\text{pred}(i))$ and $l(i) = \max(\text{pred}(i))$. It is easily seen that $\text{pred}(i) \subseteq B(l(i))$ since $l(i) \in \text{pred}(i)$ so it is adjacent to every element of $\text{pred}(i)$.

We will construct a tree decomposition T from a PEO of G inductively. The initial node is T_1 with $X(T_1) = \{1\} = \text{pred}(1) \cup \{1\} = B(1)$. Given a tree decomposition of $1, \ldots, i$, construct a tree decomposition of $1, \ldots, i+1$ by creating a node T_{i+1} with $X(T_{i+1}) = B(i)$ and connect T_{i+1} to $T_{l(i+1)}$.

▶ **Lemma 1.** *This construction is a tree decomposition of G.*

Proof. The second condition is easily seen as for every edge (i, j) with $i < j$ is in bag $X(T_j) = B(j)$. Conversely, every bag is a clique. For the first condition, each $T_i \in X^{-1}(i)$, so it is non-empty. Furthermore, since $\text{pred}(i) \subseteq B(l(i))$, it follows by induction that the set $X^{-1}(i)$ is contiguous for every i. ◀

We will abuse notation and refer to both the tree node T_i and the vertex i as i when the context is clear. We will naturally refer to $l(i)$ as the parent of i and denote the tree decomposition constructed by T_l. We will build a second tree (not a tree decomposition) by setting the parent of i as $s(i)$ and call this tree T_s.

2.3 Chordal Graph Enumeration

Wormald [19] showed that the number of connected labelled chordal graphs on n vertices is asymptotic to $\sum_r \binom{n}{r} 2^{r(n-r)} > \binom{n}{n/2} 2^{n^2/4}$. To bound the number of unlabelled chordal graphs, we take into account the number of automorphisms and obtain a lower bound of $\binom{n}{n/2} 2^{n^2/4}/n!$ unlabelled chordal graphs. Thus the information theoretic lower bound gives $\log(\binom{n}{n/2} 2^{n^2/4}/n!) = n^2/4 - \Theta(n \log n)$ bits.

2.4 Succinct Structures Used

In this paper we will use both succinct trees and succinct bit vectors. While there have been work on further compressing bit vectors to zeroth order entropy [13], we only require the most basic form of bit vectors.

▶ **Lemma 2.** *There is a succinct data structure for a bit vector B of length n using $n + o(n)$ bits of space that supports following operations in $O(1)$ time. [10]*

- *B[i]: returns the bit at position i of B.*
- *rank(i) $= \sum_{k=1}^{i} B[i]$ the number of 1s at or before position i*
- *select$(i) = j$ such that $B[j] = 1$ and rank$(j) = i$, is the position of the i-th 1.*

▶ **Lemma 3.** *There is a succinct data structure for a tree T on n nodes using $2n + o(n)$ bits of space that supports the following operations in $O(1)$ time. [11]*

- *parent, k-th child*
- *depth(i), the depth of node i*
- *level-ancestor(i,d), the ancestor of node i at depth d in the tree*
- *LCA(i,j), the lowest common ancestor of nodes i, j*

3 Representation, Adjacency and Neighbourhood

We will store the chordal graph as follows: for each vertex i, store a bit vector $W(i)$ of length $|B(l(i))|$ indicating which subset pred(i) is of $B(l(i))$ equipped with rank and select operations. We also store 1 bit indicating whether this bit vector is the all 1s vector, and if so, store the length in $\log |B(l(i))|$ bits instead. We also store the trees T_l, T_s. We identify each vertex with the corresponding node in these trees. Unless otherwise stated, the tree relations such as parent, are in T_l.

▶ **Theorem 4.** *This representation uses at most $n^2/4 + o(n^2)$ bits.*

Proof. The main fact we will use is that $pred(i) \subseteq B(l(i))$ so that $|B(i)| \leq |B(l(i))| + 1$ and equality occurs only when $pred(i) = B(l(i))$. In this equality case, we need only $\log |B(l(i))| \leq \log n$ bits.

Consider the index i such that bag size $|B(i)| = b$ is maximized. Since at each vertex, the bag size can only increase by 1 from its parent $l(i)$, there must be at least b indices such that the above inequality is an equality, and we only need $\log n$ bits each in these indices, for a total of $b \log n \leq n \log n$ bits. In all other vertices j, we need to store at most $|B(l(j))| \leq |B(l(i))| = b$ bits ($+o(b)$ for the rank and select structures). Thus in total we need to store $(b + o(b))(n - b) + n \log n + O(n) \leq n^2/4 + o(n^2)$ bits. ◀

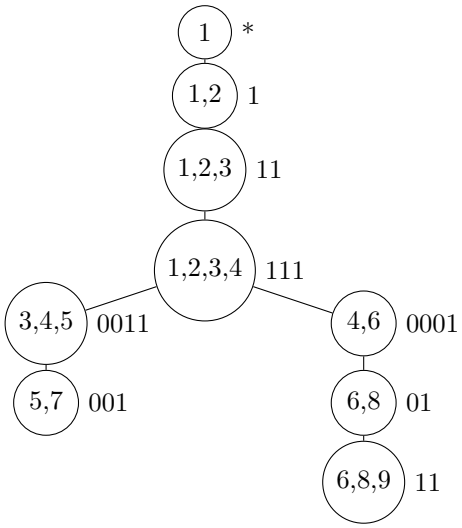

Figure 2 The label on each node is the bit vector $W(i)$. The ones that are all 1s are identified and only their lengths are stored, but for clarity, they are drawn out explicitly.

3.1 Adjacency Queries

This structure is enough to answer the following queries in $O(n)$ time:

- Given a vertex i and an integer k, find the k-th smallest predecessor of i. We will call this $decode(i, k)$.
- Given two vertices $j < i$, determine whether $(i, j) \in E$ or equivalently, $j \in pred(i)$. We will call this $adj(i, j)$

Proof. We will handle these queries recursively up the tree.

- First find the index of the k-th predecessor in the parent $l(i)$ be $k' = select(W(i), k)$. The vertex we are looking for is thus the k'-th predecessor of $l(i)$. Note that $l(i)$ is a predecessor of i and it will be at index $|B(l(i))|$. This is the only predecessor that we know exactly, all others are relative. Hence if $k' = |B(l(i))|$ we report the answer being $l(i)$, otherwise we recursively call $decode(l(i), k')$. In the worst case, this will recurse $depth(i)$ times with $O(1)$ work per recursion, which could be as bad as $\Theta(n)$.
- First note that every predecessor j of i, their tree node must be an ancestor of the tree node corresponding to i (in T_l). This is because predecessor set are taken as subsets of our ancestor's predecessor sets. Thus it is necessary that j is an ancestor of i in T_l and we can do this by $LCA(T_l, j, i) = j$, which if fails, we return false.
  Next consider the path from j to i, $j = p_0, p_1, \ldots, p_h = i$. We wish to calculate the index k_{h-1} of j in $B(l(i)) = B(p_{h-1})$ if it exists, at which point, we may determine whether it survived in the subset $pred(i)$ by checking the value of $W(i)[k_{h-1}]$. To do this, we know that the index of j in $B(p_0)$ is simply $k_0 = |B(p_0)|$. Thus j exists in $B(p_1)$ if $W(p_1)[k_0] = 1$, and its index in $B(p_1)$ is simply $k_1 = rank(W(p_1), k_0)$. We return false if $W(p_1)[k_0] = 0$. Thus we create the helper query $adj(i, j, k)$ which determines whether the k-th predecessor of j is adjacent to i, with $adj(i, j) = adj(i, j, |B(j)|)$ and in the recursive case above, call $adj(i, p_1, k_1)$. We determine p_1 in $O(1)$ time by calling level-ancestor$(i, depth(j) + 1)$. This is $O(1)$ per recursive call and the number of calls is at most $depth(i)$ which could be as bad as $\Theta(n)$. ◀

To speed up the query times, we would need to store some additional information. For certain nodes, rather than storing its predecessors relative to its parents, we store them explicitly with a bit vector using n bits (that is we store the corresponding row of the adjacency matrix) along with a rank and select structure on this. To use $o(n^2)$ bits, we can only store this information in $o(n)$ of these nodes. Furthermore, we would like to select these nodes in an uniform manner, such that for the paths above, we will encounter these *shortcut* nodes with regularity. Formally, we would like to find a set of $(o(n))$ nodes such that every path of length k in T_l intersects one of these nodes. This is exactly the problem of k-path vertex cover. Bresar et al. [3] showed that while in general it is NP-hard, it is solvable on trees in linear time.

▶ **Lemma 5.** *There is an algorithm that computes an optimal k-path vertex cover of a tree T, of size at most $\frac{|V(T)|}{k}$ in linear time.*

Let $f = \omega(1)$ be any non-constant increasing function, for example, the inverse Ackermann function. Then by Lemma 5, we can find a set of at most $\frac{n}{f(n)} = o(n)$ *shortcut* nodes such that every path in T_l contains one of these nodes. We may thus modify the above queries to cap the recursion depth.

- If i is a shortcut node, then the k-th predecessor of i is $select(W(i), k)$. Thus the recursion depth is at most $f(n)$. The time is thus $O(f(n))$.
- We follow the path to the root from i until we hit either j or a *shortcut* node. If it hit j first, then we continue as above, but with the recursion depth guaranteed to be less than $f(n)$. If we hit a *shortcut* node p_0 first, check that j is a predecessor of p_0 by $W(p_0)[j] = 1$. If not, return false, otherwise call $adj(i, p_0, rank(W(p_0), j))$ since j is the $rank(W(p_0), j)$-th predecessor of p_0. Again the recursion depth is at most $f(n)$ so the time is $O(f(n))$.

Thus we have the following result:

▶ **Theorem 6.** *There is a data structure for chordal graphs on n vertices that can answer adjacency queries in $f(n)$ time using $n^2/4 + n^2/f(n) + o(n^2)$ bits of space.*

3.2 Degree, Neighbourhood queries

Degree queries are simple, since we may write the down the degree of every vertex in $n \log n$ bits of space. For neighbourhood queries at vertex i, we split it into two parts, those neighbours that are smaller than i and those that are greater.

- For the smaller neighbours, we simply query: at vertex i, find the k-th predecessor of i for $1 \leq k \leq |pred(i)|$ - in other words, applying $decode(i, k)$. This takes $f(n)$ time per neighbour.
- For the larger neighbours at vertex i, we store a bit vector of length $(n - i)/f(n)$, with entry j being a 1 if there is a neighbour in range of vertices $[i + (j - 1)f(n), i + jf(n)]$. Total space is $n^2/f(n) = o(n^2)$. We select each 1 from this bit vector and check adjacency for every vertex in the given range. Therefore, each neighbour will need at most $f(n)$ adjacency queries, and thus we need $f(n)^2$ time per neighbour.

Thus we have the following:

▶ **Theorem 7.** *There is a data structure for chordal graphs on n vertices that can answer adjacency queries in $f(n)$ time and neighbourhood queries in $(f(n))^2$ time per neighbour using $n^2/4 + O(n^2/f(n)) + o(n^2)$ bits of space.*

4 Shortest Paths

Let $d(i,j)$ denote the distance between vertices i,j.

We would like to answer queries of the form:

- $sp(i,j) = i = p_0, p_1, \ldots, p_k = j$ a path from i to j of minimal length
- $dist(i,j) = k$ the length of the shortest path

We will show that these queries are difficult to answer, since they are a superset of adjacency queries. Thus we will look at approximate forms of these queries. We will use $d(i,j)$ to denote the actual distance and $dist(i,j)$ to denote the result of our query. We would like $dist(i,j) = d(i,j)$ but in general this is difficult. Define the approximate forms of these queries as $asp, adist$.

4.1 Ancestor Case

We will first study the easy case, where $j < i$ is an ancestor of i.

▶ **Lemma 8.** *The following algorithm:*

- *repeatedly apply $s(.)$ to i to obtain the sequence $i = p_0, p_1, \ldots, p_k$. This is equivalent to traverse the node to root path from i in T_s.*
- *stop when $p_k > j$ but $p_{k+1} = s(p_k) \leq j$.*
- *if $adj(p_k, j)$ then $dist(i,j) = k+1$ and the path is $i = p_0, p_1, \ldots, p_k, p_{k+1} = j$. Otherwise, $dist(i,j) = k+2$ and the path is $i = p_0, p_1, \ldots, p_k, p_{k+1}, j$*

correctly computes the distance (that is $dist(i,j) = d(i,j)$) and a shortest path between i and j given that j is an ancestor of i.

Proof. We induct on the distance between i and j.

If $d(i,j) = 1$, then i and j are adjacent. Furthermore, since $j \in pred(i)$, $s(i) \leq j$. Thus $i = p_0 = p_k$ and algorithm correctly gives $dist(i,j) = 1$.

Suppose that $d(i,j) = 2$, and let i, h, j be a path from i to j with minimal h. Note that if $h > i$ then both $i, j \in pred(h)$ so they are adjacent, contradicts $d(i,j) = 2$. In all other cases, the algorithm will return the path $i, s(i), j$. We need to show that $s(i)$ and j are adjacent. First note that $h > s(i)$ and they are adjacent by definition of $s(.)$. Thus the ordering must be either $i > h > s(i) > j$ or $i > h > j > s(i)$ or $i > j > h > s(i)$. In the first two orderings, $s(i), j \in pred(h)$. In the third ordering, since $s(i) \in pred(i)$, by the contiguous subtree property, it must exist in the bag along the entire path between $s(i), i$ which contains j. Thus $s(i) \in pred(j)$. In all these cases $s(i)$ is adjacent to j.

Now suppose that our algorithm is correct for distances $< k$. Let $d(i,j) = k$ and a shortest path be $i = p_0, p_1, \ldots, p_k = j$. We will show that there is a shortest path that begins with the step $i, s(i)$. Thus, $d(s(i), j) = k - 1$ and a path for it can be found using the above algorithm. But the step $i, s(i)$ is the first step in the algorithm for distances > 2, so the combination of the two is exactly the output of the algorithm.

Essentially, we will replace p_1 by $s(i)$ and argue that the resulting sequence is still a path. Let p_α be the node such that $p_\alpha < i$. We claim that $\alpha = 1$ since otherwise, $p_{\alpha-1}$ is a descendant of i and by the contiguous subtree property, i is adjacent to p_α. Thus we may replace the entire path $i, \ldots, p_\alpha$ by i, p_α, contradicting minimality. Thus at each step of the shortest path, we must go to an ancestor. Let p_β be the first node in the path such that $p_\beta < s(i)$. Note that $p_{\beta-1} > s(i) > p_\beta$ is a path on the tree, and thus by the contiguous subtree property, $s(i)$ is adjacent to p_β hence we may replace the path $i = p_0, p_1, \ldots, p_\beta$ with $i, s(i), p_\beta$. ◀

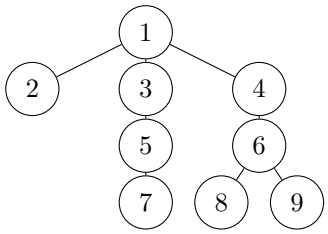

Figure 3 The tree T_s.

To answer $sp(i,j)$, we simply follow the algorithm, and traverse T_s, so we can output the path in $O(1)$ per vertex in the path. To answer $dist(i,j)$ we would like to compute k efficiently. Denote $i' = p_k$ in the algorithm. That is, p_k is the ancestor such that $p_k > j$ but $s(p_k) \leq j$. The only candidates are level-ancestor$_{T_s}(i, depth(j))$ and level-ancestor$_{T_s}(i, depth(j)+1)$. Thus we may find i' in constant time. In both queries, we require 1 adjacency check in the final step. Finally, if we do not perform this check, we are able to answer the queries within 1. Thus we obtain:

▶ **Lemma 9.** *Using the data structure as before, and suppose that j is an ancestor of i in T_l, then we can answer $sp(i,j)$ in $O(d(i,j)+f(n))$ time and $dist(i,j)$ in $O(f(n))$ time. We can answer $asp(i,j)$ in $O(d(i,j))$ time and $adist(i,j)$ in $O(1)$ time such that $d(i,j) \leq |asp(i,j)| = adist(i,j) \leq d(i,j)+1$.*

Furthermore, since we only need to traverse through T_l, T_s in the approximate queries, the space required is the two trees plus a table to identify the nodes that correspond to the same vertex. Thus the space required is $n\lceil \log n \rceil + 4n + o(n)$ bits.

We note that $\Theta(n \log n)$ bits is best possible for our idea of representing these two trees and the mapping between them. The mapping between them is equivalent to computing the function $s(.)$. Since the order of the children in the trees does not matter, they are free trees. Consider the split graph with a size $n/2$ clique $\{v_1, \ldots, v_{n/2}\}$, size $n/4$ independent set $\{u_1, \ldots, u_{n/4}\}$ together with one child of each of the $n/4$ vertices in the independent set $\{w_1, \ldots, w_{n/4}\}$. Furthermore, we have the freedom to allow $s(u_i)$ to be any permutation of $\{v_1, \ldots, v_{n/4}\}$ and also $s(w_i)$ to be any permutation of $\{v_{n/4+1}, \ldots, v_{n/2}\}$. Thus for any ordering of the children in the tree, we would have to store a permutation on $n/4$ elements. This requires $\Theta(n \log n)$ bits.

4.2 General Case

Now we study the general case when i, j do not have the ancestor relation. We will reduce to the ancestor case by the following lemma:

▶ **Lemma 10.** *Consider the shortest node-path P_T in T_l $T_i, \ldots, T_h, \ldots, T_j$. For every shortest path P_G from i to j in G, and every node T_w in P_T, $B(T_w)$ contains a vertex of P_G.*

Proof. Note that if $(u,v) \in E$ then $X^{-1}(u) \cap X^{-1}(v) \neq \emptyset$ since there must be a bag that both u, v belong to. Thus the set $\bigcup_{v \in P_G} X^{-1}(v)$ is a contiguous subtree of T_l that contains both T_i and T_j. So in particular it contains the path P_T. ◀

Let $h = LCA_{T_l}(i,j)$. Then $B(h)$ contains a vertex x on a shortest path between i, j. Thus, $d(i,j) = d(i,x) + d(x,j)$. Furthermore, x is an ancestor of both i and j.

▶ **Lemma 11.** *The algorithm $dist(i,j)$ ($sp(i,j)$):*

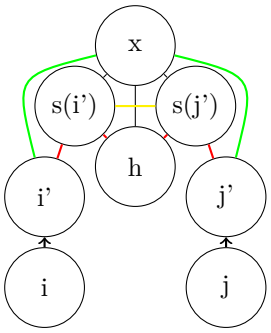

Figure 4 Green is the optimal path between i' and j'. Red is the naive path returned by *adist* and yellow is the fix from an error of 2 to an error of 1.

- *For each vertex $x \in B(h)$ compute $dist(x,i) + dist(x,j)$ (or $sp_(x,i) \cup sp_(x,j)$).*
- *return the minimum sum (resp. path) among those calculated above.*

Computes the distance (resp. a shortest path) between i, j. The time cost for $d(i,j)$ is $O(|B(h)| \cdot f(n)) = O(n \cdot f(n))$ and the time cost for $sp(i,j)$ is $O(|B(h)| \cdot (d(i,j) + f(n))) = O(n \cdot (d(i,j) + f(n))$

The time cost is dominated by the term $|B(h)|$ which is as bad as $O(n)$. It seems difficult to avoid performing the entire loop, so we will turn to approximation again.

▶ **Lemma 12.** *The algorithm $adist(i,j)$ ($asp(i,j)$):*

- *Compute $dist(h,i) + dist(h,j)$ (or $sp(h,i) \cup sp(h,j)$)*

Gives an error of at most 2 in the distance between i, j. That is $d(i,j) \leq dist(i,j) \leq d(i,j)+2$.

Proof. Let $x \in B(h)$ be the vertex that is in a shortest path between i, j. Consider the paths $h, sp(x,i)$ and $h, sp(x,j)$. These are paths between h and i, j. Thus $d(h,i) \leq 1 + d(x,i)$ and $d(h,j) \leq 1 + d(x,j)$. Finally, we have $adist(i,j) = d(h,i) + d(h,j) \leq 2 + d(x,i) + d(x,j) = 2 + d(i,j)$. ◀

4.3 Improved Bounds

We would like to improve the approximate distance algorithm in two ways: first, reduce the error to 1 and second, to use *adist* as the subroutine rather than *dist* as the subroutine. To do this, we would need to compare the computation steps that are done by both the approximate and the exact versions.

Let $x \in B(h)$ be the optimal vertex. We consider the computation of $dist(x,i)$ and $dist(h,i)$. In $dist(h,i)$, we compute i'_h and depending on $(i'_h, h) \in E$ we return $depth(i) - depth(i'_h) + 1$ or $+2$. In $adist(h,i)$ we always return $+2$ skipping the adjacency check. Now consider $dist(x,i)$. We compute i'_x which is either i'_h or an ancestor of i'_h. In the case that $i'_x = i'_h$, the worst case is that $(i'_x, x) \in E$, thus $adist(h,i) - dist(x,i) = 1$. If i'_x is an ancestor of i'_h then $depth(i'_x) < depth(i'_h)$ and $adist(h,i) - dist(x,i) \leq 0$ and we occur no error at all.

Therefore, we may replace $dist(h,i)$ by $adist(h,i)$ and obtain the same guarantees.

Next consider the case that both $(i',h), (j',h) \notin E$. This is exactly when the algorithm can potentially give an error of 2, since we may obtain an error of 1 in both branches. In this case, both $s(i'), s(j') \in B(h)$ so they are adjacent. Therefore, instead of returning the path $i, \ldots, i', s(i'), h, s(j'), j', \ldots, j$, we may return the path $i, \ldots, i', s(i'), s(j'), j', \ldots, j$, and cut the error down to 1. Note that in $adist(i,h)$ we do not perform the adjacency check, so we will always contract the path. Thus we obtain the result:

▶ **Theorem 13.** *The algorithm adist(i,j):*
return $adist(h,i) + adist(h,j) - 1$
approximates $d(i,j)$ *within 1 in* $O(1)$ *time using* $n\lceil \log n \rceil + 4n + o(n)$ *bits of space.*

5 Relation to Set Intersection Oracle

We now consider the conditions in which our approximation algorithm is exact and when it incurs an error of 1. We argued above that we incur an error of 1 on both branches when there is $x \in B(h)$ such that $(x,i'),(x,j') \in E$. Equivalently, $(x,i') \in E \Leftrightarrow x \in B(i')$. Thus $x \in B(i') \cap B(j')$. Conversely, if no such x exists, we only incur an error of 1 on exactly one branch, and due to the adjustment our algorithm is exact.

▶ **Lemma 14.** *$adist(i,j)$ incurs an error of 1 if and only if $B(i') \cap B(j') \neq \emptyset$.*

5.1 Set Intersection Oracle Problem

The set intersection oracle (SIO) problem is the following:

Given n sets $S_i \subseteq U$, such that $\sum |S_i| = N$, preprocess the sets to answer queries of $S_i \cap S_j = \emptyset$? It is known that it can be done in $O(N)$ space and $O(\sqrt{N})$ time [5]. We may also view this as storing the intersection graph of the sets, where the vertex set is each set, and two sets are adjacent if they intersect. However, if we disregard N and focus on $|U|$, we see that the intersection graph can be any graph if $|U| = n^2/4$. Thus, $\Omega(n^2)$ space is required to answer these queries. Conversely, given a graph, we may ask, what is the minimum $|U|$ such that a set intersection representation exists. This is known as the intersection number of the graph. It is equivalent to the number of cliques required to cover the edges of the graph and is NP-hard to compute.

Now consider the case that $|U| = n$. Since every chordal graph can be covered by n maximal cliques, the number of graphs that can be represented by such a set intersection representation is at least $\binom{n}{n/2} 2^{n^2/4}/n!$. Therefore, again we require $\Omega(n^2)$ space for the data structure.

Furthermore it can be shown that $O(n|U|)$ bits of space is necessary and sufficient to answer these queries.

▶ **Theorem 15.** *Let $|U| = k$ and $|U| = \omega(\log n)$ and $|U| = O(n)$. Then $O(n|U|)$ bits is necessary and sufficient to answer set intersection queries.*

Proof. One direction is trivial. We may always represent a set with a length $|U|$ bit vector, with position $i = 1$ if i is in the set. To answer the queries, with compute the bit-wise-and of the bit vectors and check if it is the 0 vector. Therefore $n|U|$ bits is sufficient to answer the query.

Conversely, consider the split graphs where we have a size $n-k$ clique and a size k independent set. The neighbourhood of each of the vertices in the independent set is one of $2^{n-k} - 2$ subsets of the clique (we omit the empty set and the entire set). Since there are k such vertices in the independent set, there are $2^{k(n-k)}$ such graphs. Divide by $n!$ to account for isomorphisms and we obtain a lower bound of $k(n-k) - O(n \log n)$ bits required to represent these sets. Note that all of these split graphs have intersection number $k+1$. For $k = o(n)$ and $k \in \omega(\log n)$, $k(n-k) - O(n \log n) = nk - o(nk)$. For $k = cn$ for some constant c, we require $(1-c)nk = O(nk)$ bits. ◀

The above does not try to optimize the query time of the data structure. To obtain an query time of $O(1)$, it is not known whether there is any non-naive data structure (storing the entire incidence matrix using $n^2/2$ bits) to solve the problem, even when $|U| = O(\log^c(n))$ (see conjecture 3 in [12]).

Next we show the close relationship between SIO and an exact distance oracle for chordal graphs. As shown above, it seems very difficult to construct an exact distance oracle that has query time $O(1)$ succinctly, using $n^2/4$ bits of space.

▶ **Theorem 16.** *Consider the SIO problem, with n sets $S_i \subseteq U$ and $|U| = n$. Any solution using B bits of space and has query time t will yield an exact distance oracle for chordal graphs occupying $B + o(B)$ bits of space with query time $O(t)$. Conversely, any exact distance oracle for chordal graphs on n nodes using $B(n)$ bits of space with query time $t(n)$ will yield a solution to the SIO problem on n sets using $B(2n)$ bits of space and query time $t(2n)$.*

Proof. WLOG assume $U = [n]$ and $S_i \neq \emptyset$ Since if $S_i = \emptyset$ then $S_i \cap S_j = \emptyset$ for every j.

The lower bound implies that $B = \Omega(n^2)$. Suppose we have a SIO, then we simply store $B(i)$ for every vertex i. By lemma 14, we can detect when $adist(i, j)$ is wrong by applying the query $B(i') \cap B(j')$. Thus we have a chordal graph distance oracle using $B + o(B)$ bits of space with query time $t + O(1)$.

Conversely, consider the split graph on $2n$ vertices. Let the vertex set be $[n] \cup \{v_1, \ldots, v_n\}$ where $[n]$ is a clique and $\{v_1, \ldots, v_n\}$ is an independent set. It is easy to see that this graph is chordal. Let $N(v_i) = S_i$. Then $d(v_i, v_j) = 2$ or 3 and $d(v_i, v_j) = 2 \Leftrightarrow S_i \cap S_j \neq \emptyset$. Thus we have reduced the SIO query to a exact distance query in chordal graphs on $2n$ vertices. ◀

References

1. Niranka Banerjee, Venkatesh Raman, and Srinivasa Rao Satti. Maintaining Chordal Graphs Dynamically: Improved Upper and Lower Bounds. In Fedor V. Fomin and Vladimir V. Podolskii, editors, *Computer Science - Theory and Applications - 13th International Computer Science Symposium in Russia, CSR 2018, Moscow, Russia, June 6-10, 2018, Proceedings*, volume 10846 of *Lecture Notes in Computer Science*, pages 29–40. Springer, 2018. doi:10.1007/978-3-319-90530-3_4.
2. Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 679–688. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644219.
3. Bostjan Bresar, Frantisek Kardos, Ján Katrenic, and Gabriel Semanisin. Minimum k-path vertex cover. *Discrete Applied Mathematics*, 159(12):1189–1195, 2011. doi:10.1016/j.dam.2011.04.008.
4. Peter Buneman. A characterisation of rigid circuit graphs. *Discrete Mathematics*, 9(3):205–212, 1974. doi:10.1016/0012-365X(74)90002-8.
5. Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40):3795–3800, 2010. doi:10.1016/j.tcs.2010.06.002.
6. Ronald Fagin. Degrees of Acyclicity for Hypergraphs and Relational Database Schemes. *J. ACM*, 30(3):514–550, 1983. doi:10.1145/2402.322390.
7. Arash Farzan and Shahin Kamali. Compact Navigation and Distance Oracles for Graphs with Small Treewidth. *Algorithmica*, 69(1):92–116, May 2014. doi:10.1007/s00453-012-9712-9.
8. Arash Farzan and J. Ian Munro. Succinct encoding of arbitrary graphs. *Theoretical Computer Science*, 513:38–52, 2013. doi:10.1016/j.tcs.2013.09.031.

9 Louis Ibarra. Fully dynamic algorithms for chordal graphs and split graphs. *ACM Trans. Algorithms*, 4(4):40:1–40:20, 2008. doi:10.1145/1383369.1383371.

10 Guy Jacobson. Space-efficient Static Trees and Graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

11 Gonzalo Navarro and Kunihiko Sadakane. Fully Functional Static and Dynamic Succinct Trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014. doi:10.1145/2601073.

12 M. Patrascu and L. Roditty. Distance Oracles beyond the Thorup-Zwick Bound. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 815–823, October 2010. doi:10.1109/FOCS.2010.83.

13 Mihai Patrascu. Succincter. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.

14 Fernando Magno Quintão Pereira and Jens Palsberg. Register Allocation Via Coloring of Chordal Graphs. In Kwangkeun Yi, editor, *Programming Languages and Systems*, pages 315–329, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

15 Donald J Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970. doi:10.1016/0022-247X(70)90282-9.

16 Donald J. Rose, Robert Endre Tarjan, and George S. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs. *SIAM J. Comput.*, 5(2):266–283, 1976. doi:10.1137/0205021.

17 Gaurav Singh, N. S. Narayanaswamy, and G. Ramakrishna. Approximate Distance Oracle in O(n 2) Time and O(n) Space for Chordal Graphs. In M. Sohel Rahman and Etsuji Tomita, editors, *WALCOM: Algorithms and Computation - 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings*, volume 8973 of *Lecture Notes in Computer Science*, pages 89–100. Springer, 2015. doi:10.1007/978-3-319-15612-5_9.

18 James R. Walter. Representations of chordal graphs as subtrees of a tree. *Journal of Graph Theory*, 2(3):265–267, 1978. doi:10.1002/jgt.3190020311.

19 Nicholas C. Wormald. Counting labelled chordal graphs. *Graphs and Combinatorics*, 1(1):193–200, 1985. doi:10.1007/BF02582944.