Succinct Data Structures for Chordal Graphs

by

Kaiyu Wu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I would like to acknowledge the names of my co-authors who contributed to the research described in this dissertation, these include:

1. Prof. J. Ian Munro

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 $O(f(n)^2)$ time per neighbour
- the length of the shortest path from u to v in $O(nf(n))$ time

Acknowledgements

This thesis is the result of my work while I was a master's student at the University of Waterloo and would have been possible without the support of my supervisor, Ian Munro. I have learned a lot under his supervision. I would like to also thank him for his feedback on drafts of this thesis.

I would like to thank my readers Therese Biedl, and Lap Chi Lau. Therese especially had a strong influence on my research. Thank you both for your helpful suggestions and improvements.

I would like to thank Hicham, Yakov, Sebastian, Bryce, and Corey for useful discussion during our group meetings.

Lastly, I would like to thank my friends and family for their support, and motivation during my master's.

Table of Contents

List of Figures

Chapter 1

Introduction

In this thesis, we construct a succinct data structure for chordal graphs. We also consider the problem of constructing a data structure to answer approximate distance queries.
Chordal graphs have a rich history of study. They were encountered in the study of Gaussian elimination of sparse matrices [21]. Chordal graphs have many equivalent characterizations including the absence of chordless cycles of length greater than 3, the existence of an perfect elimination order [22], the existence of a clique tree [5], and as the intersection graph of subtrees of a tree [25]. Rose et. al [22] 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 [20] and databases [8].
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 an element of 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 [12] was 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 [17], arbitrary graphs [10] and partial k-trees [9].

Outline of the Thesis

In subsection 1.1 we give the preliminaries that will help readers understand the thesis better. We then in subsection 1.2 present a survey of some of the related problems that

has appeared in the literature. In chapter 2 we construct our succinct data structure for chordal graphs. In chapter 3 we consider the problem of the shortest path (distance) query and after hitting a wall, relax the query to allow small errors in the answer. Finally in chapter 4, we show that at least heuristically, the problem shortest path is hard and thus the wall we hit was inevitable. To do this we show the close relationship between the distance query and the Set Intersection Oracle (SIO) problem, which is very similar to the relationship between the distance query in bipartite graphs and the SIO problem.

1.1 Preliminaries

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). We will also denote 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 logarithms are base 2.

1.1.1 Chordal Graphs

A chordal graph G is a graph that does not contain any cycle C_k for $k \geq 4$ as an induced subgraph. This condition means that every cycle in the graph must have a chord, an edge joining two non-consecutive vertices in the cycle (thus the name chordal graph). It is not surprising that chordal graphs are sometimes called *triangulated* graphs since adding chords in all cycles creates triangles.

A perfect elimination order or PEO of a graph G is an ordering $v_1, v_2, \ldots, v_n$ of the vertices of G such that the predecessor set $\text{pred}(v_i) = \{v_j; j < i, (v_i, v_j) \in E\}$ forms a clique for every vertex v_i.

A graph G is a chordal graph if and only if G has PEO. Furthermore, a linear time chordal graph recognition algorithm [22] using lexicographic-BFS outputs a PEO if and only if the input graph is chordal.

A third equivalent definition of chordal graphs is the existence of a *clique tree* that satisfies a few conditions. A *clique* tree is a tree T such that every node corresponds bijectively to an (inclusion) maximal clique of G. The tree has the property that for every pair of

cliques K, K' that correspond to nodes of T, $K \cap K'$ is contained in every clique along the path between the nodes corresponding to K, K'. This property is equivalent to for every vertex $v \in V$, the set of cliques v belongs to forms a contiguous subtree. If we identified each vertex v_i with this contiguous subtree, we see that G is the intersection graph of these subtrees of T. That is $(v_i, v_j) \in E$, if and only if the subtrees corresponding to v_i and v_j intersect in a node.

One important subclass of chordal graphs are *split graphs*. A split graph consists of a size k clique and a size $n - k$ independent set. There are no restrictions on the edges connecting a vertex in the clique to a vertex in the independent set. Of course, if we want the split graph to be connected, each vertex in the independent set must be adjacent to some vertex in the clique. To see that such a graph is chordal, consider any length ≥ 4 cycle. If two non-consecutive vertices of the cycle are in the clique, then we have a chord. But any such cycle must contain two non-consecutive vertices in the clique since vertices of the independent set cannot be consecutive in the cycle. Bender, Richmond and Wormald [2] showed that furthermore, almost all chordal graphs are split graphs (in the sense that in the limit, the probability that a randomly sampled chordal graph is a split graph is 1) , thus they are essentially the typical representative of chordal graphs.

Chordal graphs are important computationally as many otherwise NP-hard problems are solvable in polynomial time in chordal graphs. For example, computing the chromatic number, the minimal number of colours require to colour a graph. Another is computing the maximum clique of a graph. Since chordal graphs have at most n maximal cliques, and they are in the nodes of the clique tree, it is simple to find which is the largest.

1.1.2 Succinct Data Structures

A succinct data structure for a set X of objects aims to store each element (in the worst case) using the information theoretic minimal $\log(|X|) + o(log(|X|))$ bits, while still supporting relevant queries efficiently. Without this query condition (in other words, having a *data structure*), we could simply number the objects from 1 to $|X|$, but this does not support any queries.
Jacobson was the first to consider data structures in this space efficient manner. He gave succinct data structures for bit vectors, trees, and planar graphs [12].

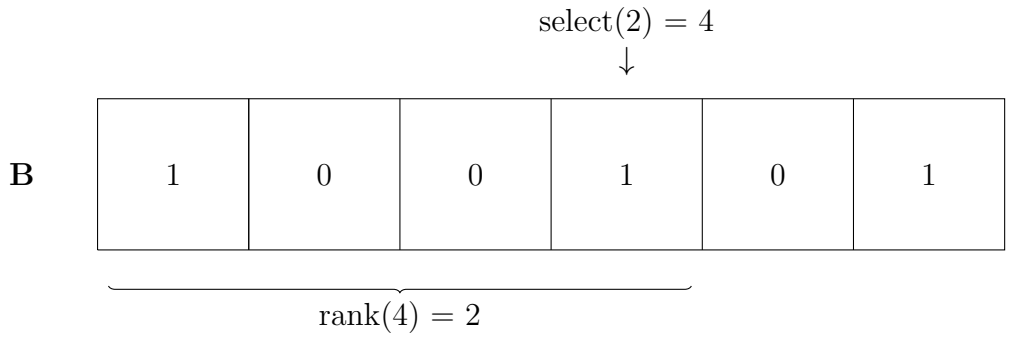

Figure 1.1: A length 6 bit vector, we illustrate how rank/select are inverse operations

Bit Vectors

A bit vector B of length n is a sequence of n bits. Since there are 2^n such sequences, the information theoretic minimum is $\log(2^n) = n$ bits. The bit vector problem asks us to support the following queries:

- *access* - return the i-th bit of B, $B[i]$.
- *rank* - given an index i, return the number of 1 bits before or at i. In other words, return $\sum_{j=1}^{i} B[j]$.
- *select* - given i, return the index of the i-th 1.

rank and *select* are in some sense inverse operations. $select(rank(i)) = i$ if $B[i] = 1$ and $rank(select(i)) = i$. Furthermore, we can define the same operations for 0s, that is $rank_0(i)$ return the number of 0s before or at i. These can be implemented with the 1 based rank/select. For example, $rank_0(i) = i - rank_1(i)$. Jacobson showed that there is a succinct data structure using $n + o(n)$ bits that support these operations using $O(\log n)$ bit probes. Munro [14] improved this result to support these operations in $O(1)$ time.
There is further work in compressing bit vectors even more. If we consider only those bit vectors with k 1s, then there are only $\binom{n}{k}$ such bit vectors, and thus the space lower bound is lower. Patrascu [19] compressed bit vectors to zeroth order entropy, that is to account for the number of 1s in the input, but we will not need the more advanced version in this thesis.

Trees

Trees are a fundamental object in computer science. From the most basic binary tree, to binary search trees, to self balancing binary search trees such as AVL trees, red-black trees, and B-trees. Despite its rich history, research in different kinds of trees continues, for example Tarjan's zip trees [24].
In the context of succinct data structures, the number of rooted, binary trees with n nodes is $C_n = \frac{1}{n+1}\binom{2n}{n}$ the n-th Catalan number. The information theoretic lower bound for binary trees is therefore $\log(C_n) + o(\log(C_n)) = 2n + o(n)$. This connection with the Catalan numbers suggests a way to represent trees succinctly using balanced parentheses. Indeed some of the first succinct representations of binary trees were based on balanced parentheses [15].
Further work in this area gave succinct representations of static trees (in particular, trees with arbitrary degree) that supported a wide range of queries, with time complexity $O(1)$ [17]. In this thesis, we will not need to use the full set of queries available. The ones we will use are:

- 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

1.1.3 Chordal Graph Enumeration

To lower bound the number of bits required, we would need a count of how many chordal graphs there are. Wormald [26] 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.

1.2 Related Work

Graphs are a fundamental combinatorical structure and it is no surprise that there is 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 [10], partial k-trees [9], planar graphs [12] and separable graphs [3].

The graphs that are most similar to chordal graphs are partial k-trees. A k-tree is a chordal graph, with the condition that in the clique tree representation, every clique has size exactly $k+1$. A partial k-tree is a subgraph of a k-tree. Note that every graph is a subgraph of the complete graph K_n, so every graph is a partial $n-1$-tree. The *treewidth* of a graph is the minimal k such that the graph is a partial k-tree.

Farzan and Kamali [9] showed that $k(n-o(n)-k/2)+\delta n$ bits are necessary to represent a graph of treewidth k. They construct an exact distance oracle (which answers distance queries) in the minimal space with query time $O(k^3 \log^3(k))$.

For chordal graphs, there has been work in the dynamic setting, focusing mainly on whether the given edge insertions/deletions preserve chordality [1, 11]. 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. [23] gave an $O(n \log n)$ bit data structure for the problem of approximate distance queries (given two vertices, output the distance between them) in chordal graphs. Their query returns a value that is anywhere between d, the actual distance and $2d+8$ in constant time.

Chapter 2

Data Structure

In this chapter, we will build a data structure that will answer adjacency, degree and neighbourhood queries. An adjacency query answers whether the edge (i, j) is in the graph. A degree query gives the degree of a given vertex. Neighbourhood queries will return a list of vertices adjacent to a given vertex. The results in this thesis are based on the ISAAC 2018 paper [16].

2.1 Preliminaries

The basis of our data structure will be a slight variant of the clique tree that has n nodes constructed from the perfect elimination order, which we will denote as a *strong 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 $u, v \in X(T_w)$.

We will say that any decomposition of the graph into a tree T and the function X satisfying the above conditions a strong tree decomposition. Note that clique trees satisfy these properties.

We start with a PEO of the graph G. Let i be a vertex, and recall that $\text{pred}(i) = \{j < i; (i,j) \in E\}$, are the set of vertices preceding i in the PEO that is also adjacent to i. 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 2.1.1. *This construction is a strong tree decomposition of G.* [1]

Proof. The second condition is easily seen as 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. □

The main difference between our strong tree decomposition and other tree decomposition such as the clique tree is that every vertex is associated with exactly one node. Furthermore, we guarantee that for any node i, the parent $l(i)$ is an element of $B(i)$. Furthermore, the structure is nicer since only one vertex is introduced at each node. This makes the proof for the space usage cleaner.
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 use T_l for strong tree decomposition constructed as in figure 2.1.

2.2 Representation, Adjacency and Neighbourhood

Since we may reconstruct the graph from the tree decomposition, all the information content of the graph exists in it. We will store the chordal graph as follows: for each vertex i, store a bit vector $W(i)$ of length $|B(l(i))|$. $W(i)$ will indicate which subset $\text{pred}(i)$ is of $B(l(i))$. Specifically, $W(i)[j] = 1$ when the j-th element of $B(l(i))$ also belongs to $\text{pred}(i)$. We also store 1 bit indicating whether this bit vector is the all 1s vector, and if so, store the length in $\lceil \log |B(l(i))| \rceil$ bits instead. This second part is important as the bit vector we store depend on the size of the cliques. If the cliques are too large, simply storing the

[1] This construction is from a personal communication with Therese Biedl

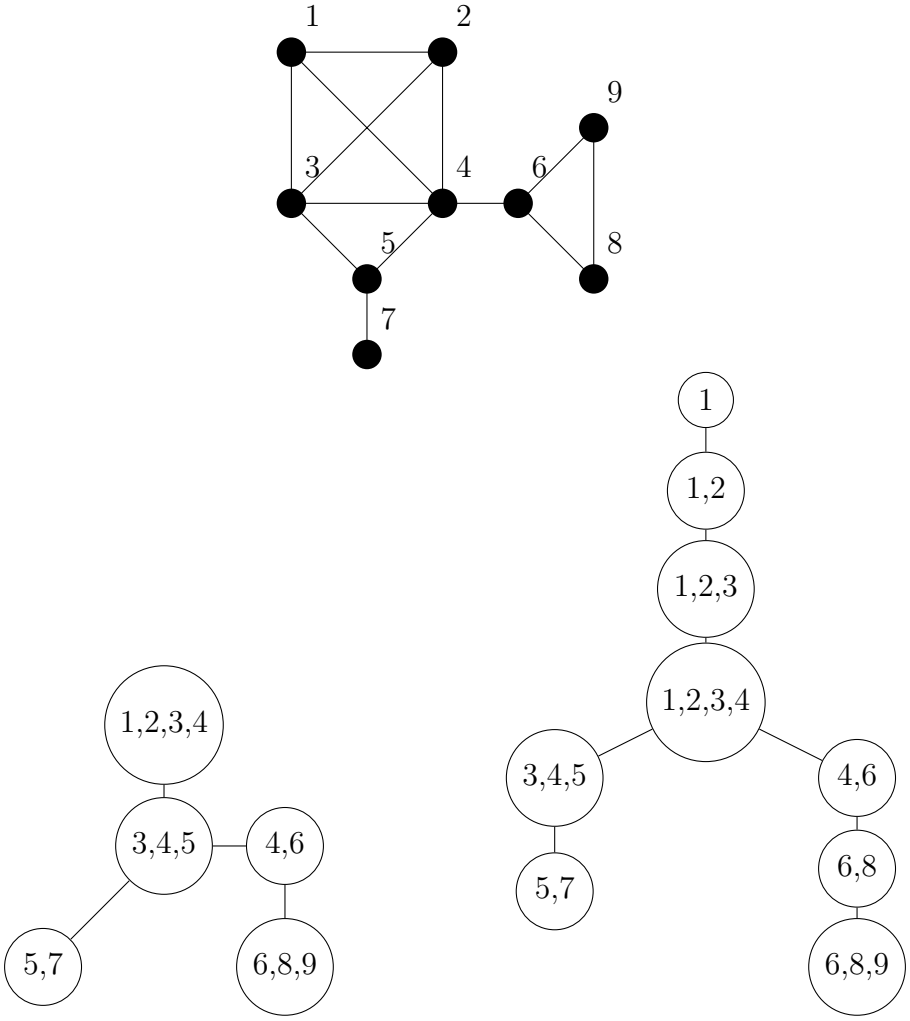

Figure 2.1: A chordal graph with a PEO of it labelled. A clique tree and the strong tree decomposition obtained from the PEO.

bit vectors would take too much space. However, there are many such simple bit vectors (consisting of all 1s) when cliques are large, which helps bring the space usage down as we will see.

Theorem 2.2.1. *This representation uses at most $n^2/4+o(n^2)$ bits. So this representation matches the information theoretic lower bound in the leading term.*

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 at most $\log n$ 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. □

2.2.1 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 $adj(i, j)$

We first show how to answer these queries recursively with recursion depth equal to tree node depth in the worst case, then improve the query time by stopping the recursion early. The second query above is the adjacency query, while the first will be useful in listing out the neighbours of a vertex.

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

- For the *decode* operation, we find the index of the k-th predecessor in the parent $l(i)$ as $k' = select(W(i), k)$. 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

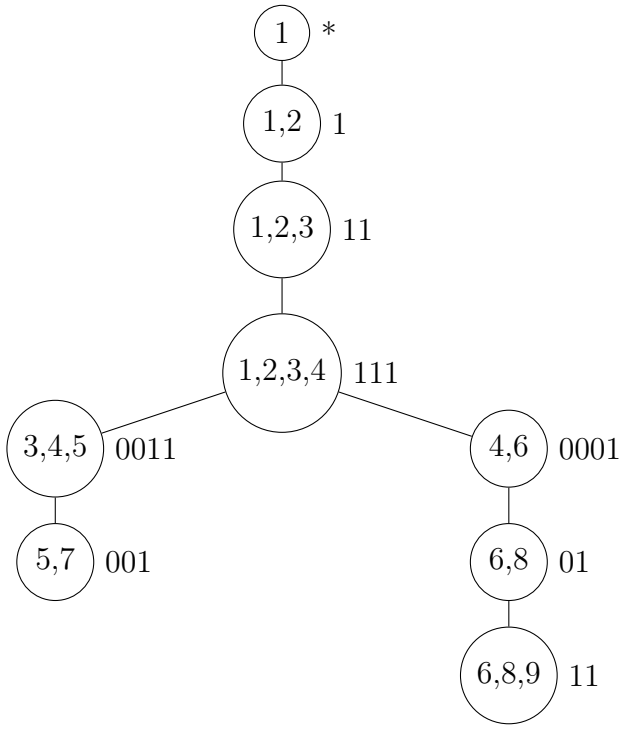

Figure 2.2: The label on each node is the bit vector $W(i)$. Those that are all 1s are identified and only their lengths are stored, but for clarity, they are drawn out explicitly.

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 $\Theta(n)$.

- For the *adj* operation, we recursively descend the tree. First note that for every predecessor j of i, the tree node of j must be an ancestor of the tree node corresponding to i (in T_l). This is because the predecessor set of i is produced by taking subsets of i's ancestor's predecessor sets. Thus it is necessary that j is an ancestor of i in T_l and we can check this by $LCA(T_l, j, i) = j$. If this 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 $adj(i, j) = adj(i, j, |B(j)|)$. In the recursive case above we will 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. Technically, this statement is stronger than what we require, as all our paths are node-to-root paths. However, the stronger statement seems easier to prove in trees whose degrees are arbitrary. This statement is the problem of k-path vertex cover. Bresar et al. [4] showed that while in general it is NP-hard, it is solvable on trees in linear time.

Lemma 2.2.2. *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 2.2.2, 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 we 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 $O(f(n))$.

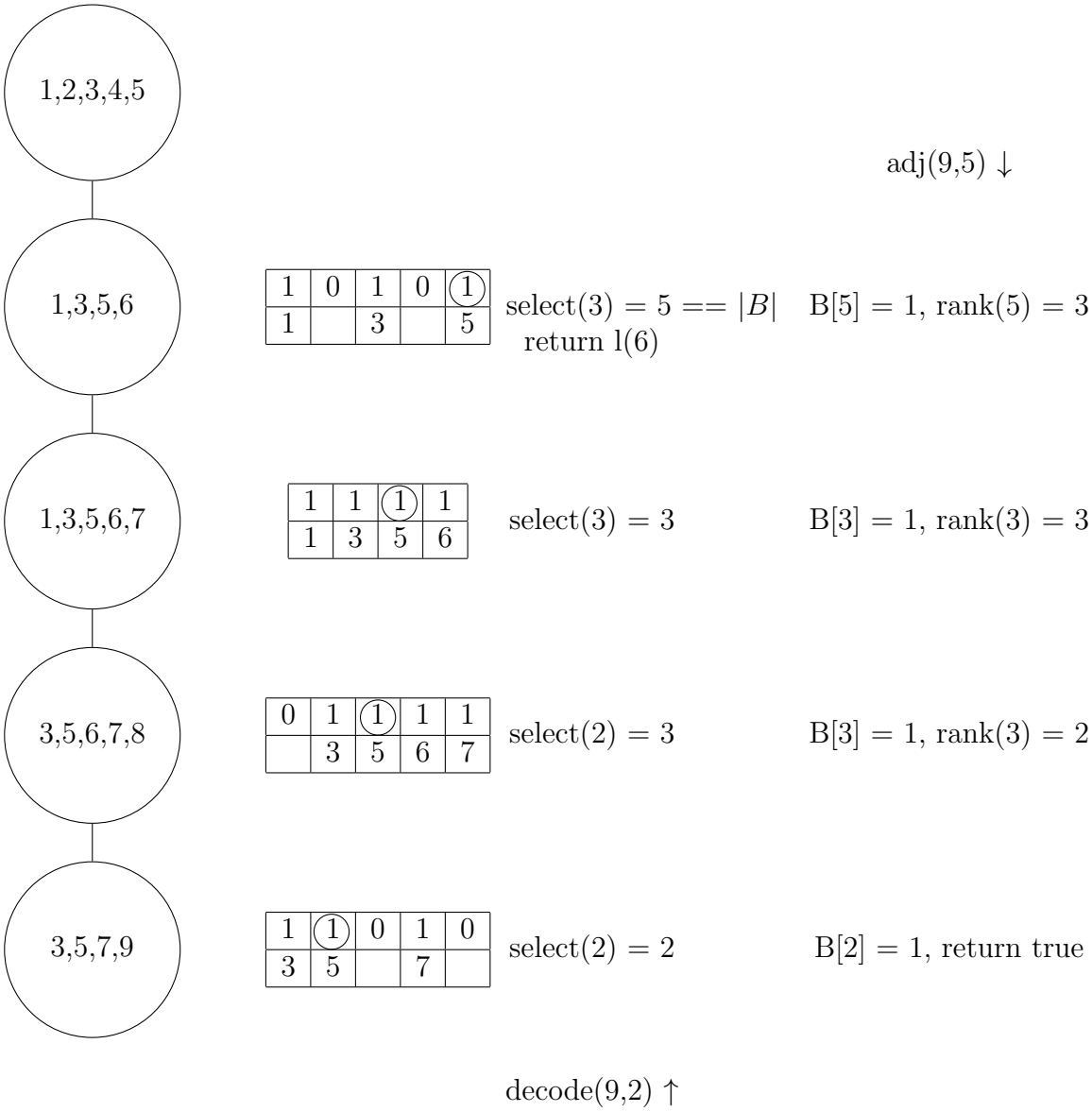

Figure 2.3: Example path in a tree decomposition, with how our algorithms works with the bit vectors

Thus we have the following result:

Theorem 2.2.3. *Let f be a function. 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. If we further assume that $f(n) = \omega(1)$, then the space usage is $n^2/4 + o(n^2)$ bits.*

2.2.2 Degree, Neighbourhood queries

Degree queries (given a vertex i, output $\deg(i)$) are simple, since we may write the down the degree of every vertex in $n \log n$ bits of space. A neighbourhood query asks that given a vertex i, output a list containing all vertices j adjacent to i. For neighbourhood queries at vertex i, we split it into two parts, those neighbours that are smaller than i (since we labelled vertices by their order in the PEO, these are those that precede 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 $O(n^2/f(n)) = o(n^2)$. 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 2.2.4. *Let f be a function. There is a data structure for chordal graphs on n vertices that can answer adjacency queries in $O(f(n))$ time, degree queries in $O(1)$ time and neighbourhood queries in $O(f(n)^2)$ time per neighbour using $n^2/4+O(n^2/f(n))+o(n^2)$ bits of space. If we further assume that $f(n) = \omega(1)$, then the space usage is $n^2/4 + o(n^2)$ bits.*

Chapter 3

Shortest Paths

In this chapter, we consider the problem of exact and approximate shortest path and distance queries. A shortest path query returns a path between two vertices i, j of minimal length. A distance query return the length of a shortest path between i, j. Let $d(i, j)$ denote the exact distance between vertices i, j. We will say our approximate distance query $adist(i, j)$ is an f-additive approximation if for all vertices i, j, $d(i, j) \leq adist(i, j) \leq d(i, j) + f(d(i, j))$. Similarly for the shortest path queries. In this chapter our goal will be a 1-additive approximation.
We will denote our queries by:

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

In the next chapter, we will show that the exact queries are difficult to answer. Specifically it is difficult to construct a data structure in minimal space with fast (for example $O(n^c)$ query time for some constant c) query time. For now, it is easily seen that these queries are at least as hard as adjacency queries, since they are a superset of them. For this reason, we will build our exact distance oracle on top of our navigational data structure. Note that we will use $d(i, j)$ to denote the actual distance and $dist(i, j)$ to denote the result of our query. We will split our investigation into two parts. First we investigate the easy case,

when i, j are in an ancestor relationship. We then investigate the general case, and see that the difficulty lies at the lower common ancestor of i and j.

3.1 Ancestor Case

Recall that we built our strong tree decomposition based on a PEO. For each vertex i, we defined $\text{pred}(i) = \{j < i; (i, j) \in E\}$. We also defined the functions l (for largest) and s (for smallest) as $l(i) = \max(\text{pred}(i))$, $s(i) = \min(\text{pred}(i))$. We defined T_l as the tree built such that the parent of each node i was $l(i)$. We will now build a second tree which is not a tree decomposition as we are not associating each node with a set of vertices. We will denote this tree by T_s and set the parent of i as $s(i)$. For an example see Figure 3.1. T_s will be used in the computation of shortest paths and distances. The main way we will use T_s is to find an ancestor in the tree using tree operations rather than repeatedly applying $s(.)$. We will now store the tree T_s succinctly. This does not change the asymptotic space usage. We identify each vertex with the corresponding node in these trees T_l and T_s crudely by writing the mapping explicitly using an array taking $n\lceil \log n \rceil$ bits. Unless otherwise stated, all tree relations such as parent, are in T_l.
We will first study the easy case, where $j < i$ is an ancestor of i. In this case, a greedy algorithm successfully gives the exact distance between i and j.

Lemma 3.1.1. *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, while 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 $h > s(i)$ (both h and $s(i) \in \text{pred}(i)$ and by definition $s(i)$ is the smallest) 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 $i, s(i), p_\beta$. □

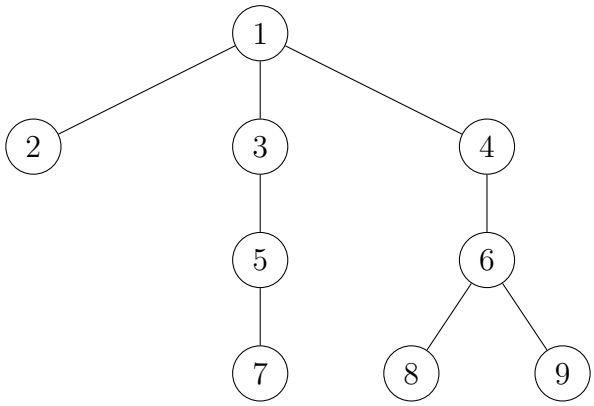

Figure 3.1: The tree T_s.

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 3.1.2. *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.

3.2 General Case

Now we study the general case when i, j do not have the ancestor relation. Since we are dealing with the same queries but in a more general setting, we will overload our query names. It should be clear when the vertices in question have the ancestor relation or not. We will reduce to the ancestor case by the following lemma:

Lemma 3.2.1. *Consider the shortest path P_T in T_l $T_i, \ldots, T_h, \ldots, T_j$ between T_i and T_j with LCA T_h. For every 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. Therefore, if we knew x, we would be able to convert the general case into two applications of the ancestor case functions. Unfortunately, it is difficult to find such an x. Indeed, the problem of finding this *minimal length ancestor* will inspire our reduction to the Set Intersection Oracle problem in the next section. In the mean time, we will crudely iterate over all such ancestors.

Lemma 3.2.2. *Consider the algorithm $dist(i, j)$ ($sp(i, j)$):*

- *Compute $h = LCA_{T_l}(i, j)$*
- *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 $O\left(|B(h)| \cdot (d(i, j) + f(n))\right) = O(n \cdot (d(i, j) + f(n))$

Our approximation algorithm in this case comes from the observation that the distances calculated in each iteration of the loop (that is using each of the vertices in the bag at h) do not change very much. The worst vertex in the bag do not perform vastly worse than the optimal vertex. Therefore, we may simply choose any vertex (in particular h) in the bag and use it as our approximation.

Lemma 3.2.3. *The algorithm $adist(i, j)$ ($asp(i, j)$):*

- *Compute $h = LCA_{T_l}(i, j)$*
- *Compute $dist(h, i) + dist(h, j)$ (or $sp(h, i) \cup sp(h, j)$)*

is a 2-additive approximation of the distance.

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 $adist(i, j) = d(h, i) + d(h, j) \leq 2 + d(x, i) + d(x, j) = 2 + d(i, j)$. □

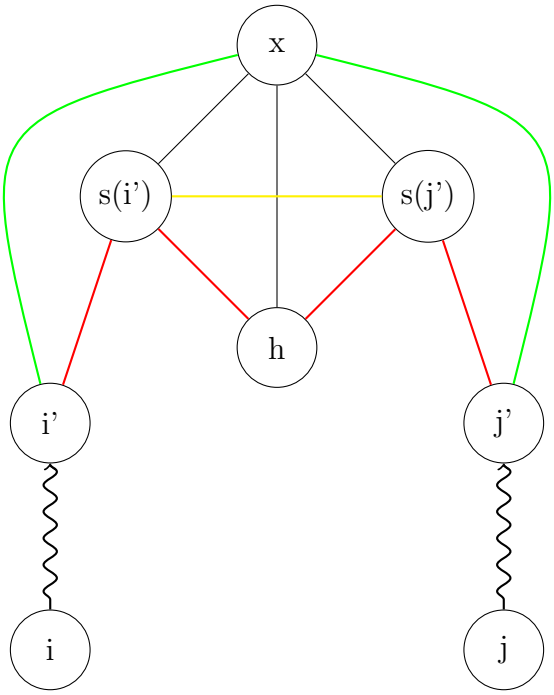

Figure 3.2: Green is the optimal path between i' and j'. Red is the naive path returned by *adist* and yellow is the fix from an error of 2 to an error of 1.

3.3 A Better Approximation

In this section, we will improve the approximation so that the error is 1 instead of 2. Furthermore, we would change the distance subroutine from *dist* to *adist*. Recall that the only difference between *adist* and *dist* in the ancestor case is that we skip the final adjacency check. We will see that this is exactly the same as not choosing minimal length ancestor in the bag at h instead of a minimal length ancestor. Therefore, since we do not use the minimal length ancestor, skipping the adjacency check does not impact our approximation. 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 a minimal length ancestor where i, j are the vertices we are interested in and $h = LCA(i, j)$. Let i'_α be the vertex right before $dist(\alpha, i)$ performs its adjacency check. Similarly for *adist*, as the only difference is that the adjacency check is omitted. Consider the computation of $adist(h, i)$ and $dist(x, i)$. They both proceed to find the vertices i'_h, i'_x, before *dist* performs its final adjacency check. Since x is an ancestor of h, we have $i'_h \geq i'_x$, that is i'_x is higher up in the tree. In the case that $i'_x \neq i'_h$, $adist(h, i)$ gives 2 steps between i'_h and h. Since i'_x is a step further than i'_h, $dist(x, i)$ gives that x is at least 2 steps away from i'_h by $i'_h \to i'_x \to x$. Thus $adist(h, i) \leq dist(x, i)$. In the case that $i'_x = i'_h$, $adist(h, i)$ gives 2 steps from i'_h to h and $dist(x, i)$ is 1 step from $i'_h = i'_x$ to x. Thus we have $adist(h, i) \leq dist(x, i) + 1$. In particular, we have an error exactly when $dist(x, i)$ performs the adjacency check, it is true.

Recall that our guarantees from before was that $dist(h, i) \leq dist(x, i) + 1$. Therefore, we may replace $dist(h, i)$ by $adist(h, i)$ and obtain the same guarantees.

Next consider the path we obtain. This path is $j \to^* j' \to s(j') \to h \to s(i') \to^* i$. In this case, both $s(i'), s(j') \in B(h)$ and thus are adjacent. We may therefore shorten the path by removing h from it. Therefore we obtain the path $j \to^* j' \to s(j') \to s(i') \to^* i$, which always exists. This show that in our choice of x, we may never have the situation that both $dist(x, i), dist(x, j)$ fails the adjacency check in the final step. Otherwise, our path adjustment above will give a better path than our optimal path. In particular, we could have chosen $x = s(i')$ which does not fail the adjacency check on the branch above i. Since we may always cut the length by 1, we see that in doing so, we reduce our error by 1 down to 1.

Lemma 3.3.1. *The algorithm adist(i,j):*

- *Compute* $h = LCA_{T_l}(i, j)$
- *return* $adist(h, i) + adist(h, j) - 1$

approximates $d(i, j)$ within 1 in $O(1)$ time.

Combining this result and 3.1.2, we obtain:

Theorem 3.3.2. *There is a data structure for Chordal graphs that gives a 1-additive approximation to distances in $O(1)$ time using $n\lceil \log n \rceil + 4n + o(n)$ bits of space.*

Chapter 4

Relation to Set Intersection Oracle

In this chapter, we explore the relationship between our distance queries and the Set Intersection Oracle problem. We first define the problem and explore some variations of it, then show the rough equivalence between it and distances in chordal graphs.

4.1 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 $S_i \cap S_j = \emptyset$? 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. It is known that it can be done in $O(N)$ space and $O(\sqrt{N})$ query time [6].
The above data structure uses N as the main parameter. In particular, if $N \in \Theta(n^2)$, the query time would be $\Theta(n)$. In our situation, we wish to use $|U|$ as the main parameter and view the sets as primarily the underlying intersection graph. This brings up the converse question: given a graph, what is the minimal $|U|$ such that we may find a set intersection representation of the graph. 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-complete to compute [13].
If we take $|U| = n^2/4$, the intersection graph can be any graph [7]. We are interested in the case that $|U| = n$ as it is in this case, that chordal graphs fall into. This is because every chordal graph can be covered by n maximal cliques. The number of graphs that can be represented by such a set intersection representation with $|U| = n$ is at least $\binom{n}{n/2} 2^{n^2/4}/n!$

by the counting argument of chordal graphs. Therefore, again we require $\Omega(n^2)$ space for the data structure.
Furthermore it can be shown that for a set system with universe U that is not too large or small, $O(n|U|)$ bits of space is necessary and sufficient to answer these queries (without any regard to the time complexity of the queries).

Theorem 4.1.1. *Let $|U| = k$ and $|U| = \omega(\log n)$ and $|U| = O(n)$. Then $O(n|U|)$ bits are 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 $(1 - c)nk - O(n \log n) = O(nk)$ bits. □

The above does not try to optimize the query time of the data structure. To obtain a 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 [18]).

4.2 Connection to Distances in Chordal Graphs

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

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 4.2.2. *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 $S_i \neq \emptyset$, for 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 4.2.1, 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, let $S_1, \ldots, S_n$ be an instance of the SIO problem and 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. 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. □

Chapter 5

Further Research

In this chapter, we ask some open questions and lines of potential research on topics related to those presented in the thesis.

- We gave an approximate distance query data structure using $n \log n + O(n)$ bits and showed that our approach cannot easily be optimized further by exploiting any structural properties between the mapping of the nodes of T_l and T_s. Furthermore, it would seem that given the correct enumeration model, the number of chordal graphs under a approximate distance "equivalence metric" is much less than the number of chordal graphs. How this "equivalence metric" would be defined is not attempted. However it should capture the notion that two graphs are equivalent if they cannot be distinguished by only approximate distance queries. Such a definition would have to deal with transitivity correctly. This is not surprising as our data structure gives the same queries for every split graph with a size k_1 clique and a size k_2 independent set, regardless of how the edges connect the clique to the independent set.

- This question of approximate distance queries and lower bounds can be extended past chordal graph. We could ask similar questions for other classes of graphs. For instance, how difficult is constructing an $d + 1$-approximate distance data structure for arbitrary graphs? What is the information theoretic lower bound in this case? We could replace arbitrary graphs with bipartite graph, or planar graphs and ask the same question.

- We discussed a conjecture from [18] that stated that $\Omega(n^2)$ space is required to answer set intersection oracle queries in $O(1)$ even when $|U| \in O(\log^c(n))$. We do not need

such a restriction as $|U| = n$ is our situation. In this case, it would require $\Omega(n^2)$ space due to the chordal graph lower bound, but a time bound of even $O(n^c)$ for $c < 1$ would be useful in our data structure.

- We could phrase the SIO problem in a different way. The set up is the same, but suppose that you have an oracle that answers $v \in S_i$. That is, you are allowed to query set elements. Can you build an intersection oracle in $o(n^2)$ additional space with query time $O(n^c)$ for some $c < 1$?

References

[1] Niranka Banerjee, Venkatesh Raman, and Srinivasa Rao Satti. Maintaining chordal graphs dynamically: Improved upper and lower bounds. In *Computer Science - Theory and Applications - 13th International Computer Science Symposium in Russia, CSR 2018, Moscow, Russia, June 6-10, 2018, Proceedings*, pages 29–40, 2018.

[2] E. A. Bender, L. B. Richmond, and N. C. Wormald. Almost all chordal graphs split. *Journal of the Australian Mathematical Society. Series A. Pure Mathematics and Statistics*, 38(2):214221, 1985.

[3] 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, 2003.

[4] Bostjan Bresar, Frantisek Kardos, Ján Katrenic, and Gabriel Semanisin. Minimum k-path vertex cover. *Discrete Applied Mathematics*, 159(12):1189–1195, 2011.

[5] Peter Buneman. A characterisation of rigid circuit graphs. *Discrete Mathematics*, 9(3):205–212, 1974.

[6] Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40):3795 – 3800, 2010.

[7] Paul Erds, A. W. Goodman, and Louis Psa. The representation of a graph by set intersections. *Canadian Journal of Mathematics*, 18:106112, 1966.

[8] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.

[9] Arash Farzan and Shahin Kamali. Compact navigation and distance oracles for graphs with small treewidth. *Algorithmica*, 69(1):92–116, May 2014.

[10] Arash Farzan and J. Ian Munro. Succinct encoding of arbitrary graphs. *Theoretical Computer Science*, 513:38 – 52, 2013.

[11] Louis Ibarra. Fully dynamic algorithms for chordal graphs and split graphs. *ACM Trans. Algorithms*, 4(4):40:1–40:20, 2008.

[12] 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, 1989.

[13] Lawrence T. Kou, Larry J. Stockmeyer, and C. K. Wong. Covering edges by cliques with regard to keyword conflicts and intersection graphs. *Commun. ACM*, 21(2):135–139, 1978.

[14] J. Ian Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, pages 37–42, 1996.

[15] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 118–126, 1997.

[16] J. Ian Munro and Kaiyu Wu. Succinct data structures for chordal graphs. In *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16-19, 2018, Jiaoxi, Yilan, Taiwan*, pages 67:1–67:12, 2018.

[17] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014.

[18] 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, Oct 2010.

[19] 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, 2008.

[20] 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.

[21] Donald J Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597 – 609, 1970.

[22] 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.

[23] Gaurav Singh, N. S. Narayanaswamy, and G. Ramakrishna. Approximate distance oracle in o(n 2) time and o(n) space for chordal graphs. In *WALCOM: Algorithms and Computation - 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings*, pages 89–100, 2015.

[24] Robert E. Tarjan and Caleb C. Levy. Zip trees. *CoRR*, abs/1806.06726, 2018.

[25] James R. Walter. Representations of chordal graphs as subtrees of a tree. *Journal of Graph Theory*, 2(3):265–267, 1978.

[26] Nicholas C. Wormald. Counting labelled chordal graphs. *Graphs and Combinatorics*, 1(1):193–200, 1985.