# Compact Navigation and Distance Oracles for Graphs with Small Treewidth ⋆

**Technical Report CS-2011-15**

Arash Farzan[1] and Shahin Kamali[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] Cheriton School of Computer Science, University of Waterloo, Ontario, Canada

**Abstract.** Given an unlabeled, unweighted, and undirected graph with $n$ vertices and small (but not necessarily constant) treewidth $k$, we consider the problem of preprocessing the graph to build space-efficient encodings (oracles) to perform various queries efficiently. We assume the word RAM model where the size of a word is $\Omega(\log n)$ bits.
The first oracle, we present, is the navigation oracle which facilitates primitive navigation operations of adjacency, neighborhood, and degree queries. By way of an enumerate argument, which is of independent interest, we show the space requirement of the oracle is optimal to within lower order terms for all treewidths. The oracle supports the mentioned queries all in constant worst-case time. The second oracle, we present, is an exact distance oracle which facilitates distance queries between any pair of vertices (i.e., an all-pair shortest-path oracle). The space requirement of the oracle is also optimal to within lower order terms. Moreover, the distance queries perform in $O\left(k^2 \log^3 k\right)$ time. Particularly, for the class of graphs of our interest, graphs of bounded treewidth (where $k$ is constant), the distances are reported in constant worst-case time.

## 1 Introduction

Graphs are arguably one of the most prolific structures to model relationships among entities. With the ever-growing size of objects to model, the corresponding graphs increase in size. As a result compact representation of graphs has always been of interest. In this paper, we consider the problem of representing graphs compactly while allowing efficient access and utilization of the graph by showing fast support of navigation and distance queries.

Random graphs are highly incompressible [2]. Fortunately, graphs that arise in practice are not random and turn out to have some combinatorial structural property. Therefore, researchers have considered graphs with various combinatorial structures for the purpose of space-efficient representation (see [9] for a review of the exiting results). In this paper, we are interested in compact representation of graphs with a small treewidth (to be defined in Section 2). Graphs

⋆ A preliminary version of this paper is to appear in ICALP 2011

of bounded treewidth are of interest since many NP-hard problems on general graphs are solvable in polynomial time on these graphs. In addition, graphs with small treewidth occur in many more real-world applications [5,6]. We assume the standard word RAM model where a word is at least $\lg n$ bits wide and $n$ is the number of vertices (lg denotes $\log_2$).

### 1.1 Contribution

In the first part of the paper (Section 4), we describe a data structure that encodes a given undirected and unlabeled graph with $n$ vertices and treewidth at most $k$ in $k(n + o(n) - k/2) + O(n)$ bits and supports degree, adjacency, and neighborhood queries in constant time. Degree query is to report the degree of a vertex. Adjacency query is given two vertices $u, w$ to determine if edge $(u, v)$ exists. Neighborhood query is to report all neighbors of a given vertex in constant time per neighbor. These three queries constitute the set of primitive navigational queries often required in a graph [2,10,3].

In [12] an implicit representation of graphs with treewidth $k$ in $n(\lg n + O(k \lg \lg(n/k)))$ bits is presented, which is not compact for all values of $k$. Although it is not explicitly mentioned, the succinct representation of separable graphs given in [3] yields an optimal navigation oracle for graphs of bounded treewidth for any treewidth $k = O(1)$. This is since graphs with a constant treewidth are also separable (a graph is separable if it and its subgraphs can be partitioned into two approximately equally sized parts by removing a relatively small number of vertices [2]). The storage requirement of the oracle for separable graphs is optimal to within lower order terms, and previously mentioned navigation queries perform in constant time. In this paper, we extend the result to graphs with treewidth $k$, where $k = \Omega(1)$.

Moreover, we show that the storage requirement of the oracle is optimal for all values of $k$ by proving that $k(n - o(n) - k/2) + \delta n$ bits are required to encode graphs of treewidth $k$ and $n$ vertices ($\delta$ is a positive constant). Our proof is a counting argument which is of independent interest as to best of our knowledge, there existed no such enumerative result for graphs with a given treewidth (though a lower bound of $kn - o(kn)$ is known [11] for graphs with pagenumber $k$, which are a larger family of graphs which include graphs with treewidth smaller than $k$ [8]). The desired oracle of this paper adopts the encoding of [3] for values $k = O(1)$ and the encoding outlined in this paper for $k = \Omega(1)$. Since the storage requirement of the oracle matches the entropy bound for constant values of $k$ and our lower bound for non-constant values of $k$, both the space of the oracle and our lower bound are tight.

In the second part of the paper, we give distance oracles for undirected, unlabeled, and unweighted graphs with $n$ vertices and treewidth $k$ that for all values of $k$ requires the entropy bound number of bits to within lower order terms. These are *exact* oracles that report the distance of two given vertices precisely. The distance queries perform in $O(k^2 \lg^3 k)$ in which $k$ is the graph treewidth. We emphasize that for graphs of bounded treewidth where $k$ is constant (the family of graphs of our interest), the queries are supported in constant time.

Exact distance oracles for unweighted undirected graph require $\Omega\left(n^2\right)$ bits and there exists an oracle with about $0.79n^2$ bits [16]. Hence, we show that for graphs with a bounded treewidth, these results can be significantly improved as there is a linear size exact distance oracle (the number of edges can be $\theta(kn)$).

The time to construct the oracles depends on the time to compute the treewidth of the given graph and compute the tree decomposition correspondingly. Determining the treewidth of a graph is NP-hard [5]. Fortunately however, for graphs with constant treewidth, the treewidth and the corresponding tree decomposition can be determined in linear time [6]. Moreover, for graphs with treewidth $k = \omega(1)$, there exists a polynomial time algorithm that approximates the treewidth within $O(\log k)$ factor and generates the corresponding tree decomposition [5]. All other aspects of navigation oracles can be constructed in $O(kn)$ time where $k$ is the determined treewidth and $n$ is the number of vertices. For the distance oracle, we pre-compute distances between every pairs of vertices at the initial stage, and this can be accomplished in $o(kn^2)$ time [7].

## 2 Tree Decompositions and variations

We use the notion of tree decompositions of graphs to design the oracles:

**Definition 1 ([5]).** *A tree decomposition of a graph $G = (V, E)$ of width $k$ is a pair $(\{X_i \| i \in I\}, T)$ where $\{X_i \| i \in I\}$ is a family of subsets of $V$ (bags), and $T$ is a rooted tree whose nodes are the subsets $X_i$ such that*

- $\bigcup_{i \in I} X_i = V$ *and* $\max_{i \in I} |X_i| = k+1$.
- *for all edges $(v, w) \in E$, there exists an $i \in I$ with $v \in X_i$ and $w \in X_i$.*
- *for all $i, j, k \in I$: if $X_j$ is on the path from $X_i$ to $X_k$ in $T$, then $X_i \cap X_k \subseteq X_j$.*

*We say a vertex $v \in V$ is* introduced *in node $X_i (i \in I)$ of the tree, if $v$ is in the bag $X_i$ ($v \in X_i$) but not in that of the parent of $X_i$. All vertices at the bag of the root node are introduced by definition.*

For each of the oracles, we use a specially adapted version of tree decomposition. For the navigation oracle, we use a *standard* tree decomposition, in which each bag contains exactly $k+1$ vertices, and two neighboring nodes share exactly $k$ vertex, i.e., each node introduces one vertex. It is known that a tree decomposition can be changed to form a standard tree decomposition in linear time [4]. To design distance oracles, we use the *height-restricted* tree decomposition $T$, whose height is logarithmic in the number of vertices, i.e., $height(T) = O(\log n)$. A tree decomposition can be transformed into a height-restricted tree decomposition by the following lemma:

**Lemma 1.** *Given a tree decomposition with treewidth $k$ for a graph with $n$ vertices, one can obtain, in linear time, a height-restricted tree decomposition with $n$ nodes and width at most $3k + 2$ (proof in the long version of the paper).*

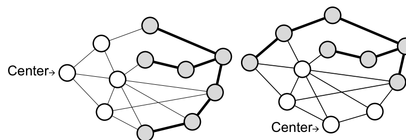



**Fig. 1.** Two different understanding of tree vertices in an asymmetric 4-graph. Dark vertices are tree vertices.

## 3 Lower bound

To best of our knowledge, there exists no enumerative result for the number of unlabeled graphs with a given treewidth $k$. We prove one in this section. We use the concept of *asymmetric trees* (also known as *identity* trees), which are trees in which the only automorphism is the identity, i.e., each vertex can be uniquely distinguished from others. Harary et al, showed the total number of asymmetric trees on $n$ nodes is $u(n) \sim cn^{-5/2}\mu^{-n}$ in which $c$ and $\mu$ are positive constants roughly equal to 0.299 and 0.397, respectively [14]. We use this result to count *asymmetric $k$-graphs* as a family of graphs with treewidth $k$.

**Definition 2.** *An asymmetric $k$-graph on $n$ vertices is a graph which has an asymmetric tree of size $n-k$ as an induced subgraph. The involved vertices in this subgraph are called* tree vertices*. Among the other $k$ vertices, there is one vertex, called* center*, which is connected to all other $k-1$ non-tree vertices and is not connected to any of the $n-k$ tree vertices (Figure 1).*

**Lemma 2.** *Any asymmetric $k$-graph has treewidth at most $k$.*

*Proof.* Consider a tree decomposition of the asymmetric tree, which has at most two vertices in each bag. Copy all other vertices except the center to all bags. Now each bag contains $2+(k-1)$ vertices. Create a new bag of size $k$ involving all non-tree vertices (including center) and attach it to an arbitrary position in the tree decomposition. The result is a legitimate tree decomposition of width $k$ (at most $k+1$ vertices in each bag).

Therefore, to get a lower bound on the number of graphs with of treewidth $k$, we just count asymmetric $k$-graphs.

**Theorem 1.** *The number of asymmetric $k$-graphs with $n$ vertices is at least $x_n \sim c2^{(k+\delta)n-(k^2+(3+2\delta)k-2)/2} \times (n–k)^{-5/2}/(n(k–1)!)$ where $c$ and $\delta$ are constants roughly equal to 0.299 and 0.332 (proof in the long version of the paper).*

Since we match the bound with an encoding, the exponent is tight within lower order terms.

**Corollary 1.** *At least $k(n-o(n)-k/2)+\delta n$ bits are required to represent a graph of treewidth $k$ with $n$ vertices, where $\delta$ is a constant roughly equal to* 0.332.

## 4 Navigation Oracles

In this section, we provide a compact representation of graphs of treewidth $k$ which supports adjacency, neighborhood, and degree queries in constant time in

the $\lg n$-bit word RAM model. The representation requires $k(n + o(n) - k/2) + O(n)$, which is tight given corollary 1. First we mention some existing results for auxiliary data structures we use in our representation.

**succinct rank/select structures:** For a binary sequence $S$, we define $access_S(i)$ as the content of the $i$'th index of $S$, $rank_S(i, c)$ as the number of occurrences of $c$ before index $i$, and $select_S(i, c)$ as the index of the $i$'th occurrence of $c$ in $S$ $(c \in \{0, 1\})$. There are data structures which represent a binary sequence of length $n$ using $n + o(n)$ bits which supports *access*, *rank*, *select* in constant time. Moreover, for sequences with $m$ ones $(m \ll n)$, the space can be reduced to $\lg \binom{n}{m} + O(n \lg\lg n/ \lg n)$ to support queries in constant time [17].

**Balanced Parenthesis and Multiple Parenthesis:** A balanced parenthesis sequence of size $2n$, which is equivalent to an ordered tree of size $n$, can be represented in $2n+o(n)$ bits, with support of $access(v)$, $rank(v, '(')$, $select(v, '(')$, $findmatch(v)$, and $child(i, v)$ in constant time [15]; *access*, *rank*, and *select* are defined as before, $findmatch(v)$ finds the position of parenthesis matching the parenthesis at position $v$, and $child(v, q)$ finds the position of the $q$'th child of node $v$. A multiple parenthesis sequence, is an extension of balanced parenthesis to $k$ types of parenthesis, where an open parenthesis of *type* $i$, denoted by $(_i$, can be matched by a closed parenthesis of the same type, denoted by $)_i$.

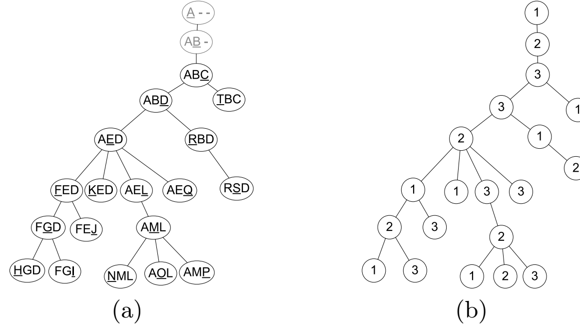
**Lemma 3.** *[1] A multiple parenthesis sequence with $2n$ parentheses of $k$ types, in which the parentheses of any given type are balanced, can be represented using $(2+\epsilon)n \lg k + o(n \lg k)$ bits to support m_access, m_rank, m_select, m_findmatch and m_enclose in $O(1)$ time; all operations are defined as before, $m_enclose(v, i)$ gives the position of the tightest open parenthesis of type $i$ which encloses $v$.*

**Compact Tables:** Given a binary matrix $M$ of size $k \times n$, we are interested in a compact representation of $M$ which supports the following queries: $access(i, j)$ which gives the content of $M[i, j]$, $r_successor(i, j)$ which gives the entry one in row $i$ that comes after index $j$, and $c_successor(i, j)$ which is defined identically on columns. For our purpose, we need to represent matrices in which $k \leq n$, and the first $k$ columns form a triangular submatrix.

**Lemma 4.** *A $k \times n$ matrix, in which the first $k$ columns form a triangular submatrix $(k \leq n)$ can be presented using $kn - k^2/2 + o(kn)$ bits to support access and successor queries in constant time (proof in the long version of the paper).*

### 4.1 Representing the tree decomposition

Assume for a given graph $G = (V, E)$, a tree decomposition $\tau$ of width $k$ is given in standard form. We assign *types* to all vertices in a top-down manner: for the vertices in the root, fix an arbitrary ordering $1 .. k+1$ and give a vertex type $i$ iff it has index $i$ in this ordering. For a vertex $v$ introduced in a bag $X$, define $type(x) = j$, where $j$ is the type of the vertex in the parent of $X$ which has been replaced by $v$. Note that the only information associated with each bag is the type of the vertex it introduces. So we can present the tree decomposition $\tau$ as an ordered tree with a single label, not larger than $k$+1, on each bag. We assume




$(_1\, (_2\, (_3\, (_3\, (_2\, (_1\, (_2\, (_1\, )_1\, (_3\, )_3\, )_2\, (_3\, )_3\, )_1\, (_1\, )_1\, (_3\, (_2\, (_1\, )_1\, (_2\, )_2\, (_3\, )_3\, )_2\, )_3\, (_3\, )_3\, )_2\, (_1\, (_2\, )_2\, )_1\, (_1\, )_1\, )_3\, )_2\, )_1$

**Fig. 2.** A standard tree decomposition, an ordered labeled tree, and a multiple parenthesis are all equivalent.

the root introduces $k+1$ vertices of different types, and to make representation easier separate them in a path of bags, with labels 1 to $k+1$ (See Figure 2).

To represent $\tau$ efficiently, we use multiple parenthesis structure of Lemma 3. Assume a preorder traversal of $\tau$; we open a parenthesis of type $i$ $(i \leq k)$ whenever we enter a bag with label $i$, and close it when we leave the bag. The result would be a balanced sequence of $2n$ parenthesis of $k+1$ types, and using Lemma 3, this can be presented using $(2+\epsilon)n \lg k + o(n \lg k)$ bits. We call this sequence the *MP sequence* and represent every vertex in the graph by the index of its opening parenthesis in this sequence.

To represent a graph of treewidth $k$, beside the tree decomposition, we need to store which edges are indeed present in each bag. In fact, the tree decomposition represents a (full) $k$-tree, which is the graph with maximal edges to respect the tree decomposition. In a graph of treewidth $k$ when a new vertex is introduced in a bag $X$, it can be connected to any subset of $k$ vertices present in $X$. These vertices all have distinct types as they all appear in bag $X$. For each vertex $v$, let $l_v$ be a bitmap of size $k+1$, such that $l_v(j)$ denotes if there is an edge between $v$ and $u_j$, where $u_j$ is the unique vertex of type $j$ present in bag $X$. Observe that $u_j$ is introduced in the closest ancestor of $v$ which has type $j$. Let all $l_v$s form the columns of a table $M$, referred as 'big table', where the vertices are arranged in preorder. So $M$ is a matrix of size $(k+1) \times n$ and $M[j,v] = l_v(j)$ (see Figure 3). Since two vertices of the same type cannot be connected, for any vertex $v$ we have $M[type(v), v] = 0$ (in the figure these entries are distinct by '*'). Also the first $k$ columns of $M$ are associated with the vertices introduced in the root and form a triangular submatrix. We apply Lemma 4 to store $M$ in $kn - k^2/2 + o(kn)$ bits to support *access* and *successor* queries in constant time.

Assume we are given a vertex $v$ (its index in the MP sequence), and we need to access its column in the big table, i.e., the index of $v$ in the preorder walk. We use a *map structure* as follows: create a binary sequence $S$ of size $2n$

|   | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | * | 1 | 1 | 0 | 1 | * | 0 | * | 1 | 1 | 1 | 0 | 0 | * | 0 | 1 | 1 | * | 1 | * |
| 2 | - | * | 0 | 1 | * | 0 | * | 1 | 0 | 1 | 0 | 1 | * | 0 | * | 1 | 1 | 0 | * | 1 |
| 3 | - | - | * | * | 1 | 0 | 1 | 1 | * | * | 1 | * | 0 | 1 | 1 | * | * | 1 | 1 | 1 |

**Fig. 3.** A big matrix associated with tree decomposition of Figure 2

with one at position $i$ if the $i$'th element is an open parenthesis (of any type) and zero otherwise. We store this sequence using $2n + o(n)$ bits to support *rank* and *select* in constant time. Now $rank_S(v, 1)$ gives the index of $v$ in the preorder walk, and $select_S(i, 1)$ retrieves the $i$'th vertex in the preorder walk. This enables us to interchangeably represent a vertex by its position in the preorder (the big matrix index) or its position in the MP sequence. Moreover, assume we are given a range $R$ in the MP sequence which may start or end with a close parenthesis, and we need the preorder range which include the involved open parentheses. If both endpoints of $R$ are open parentheses, we simply map them into preorder indices as discussed. If $R$ starts (ends) with a closed parenthesis, we need to find the next (previous) open parenthesis of any type. We can use $S$ to find the index of next (previous) open parenthesis as $select_S(rank_S(i, 1) \pm 1)$.

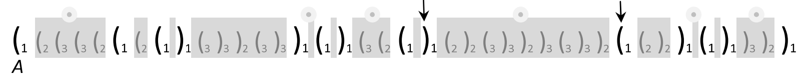
The MP sequence and the big table are sufficient for representing a graph of treewidth $k$. The other data structures used in the rest of this section are indices to support queries in constant time. Due to lack of space, we describe support for neighborhood queries here and support for degree and adjacency queries are presented in the long version of the paper.

### 4.2 Neighbor Report

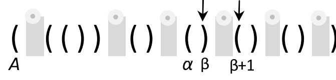
We are given a vertex $v$, and asked to report its neighbors in constant time per neighbor. We say a vertex $u$ is a *potential neighbors* of $v$ if there is a bag that contains both vertices. The column representing $v$ in the big table distinct the actual neighbors of $v$ among those potential neighbors which precede $v$ in preorder walk. To report these neighbors, we successively apply *c_successor* on the big table $M$ to visit all ones in the column of $v$. Assume we have $M[j, v] = 1$, then we report the parenthesis of type $j$ which encloses $v$ using $m_enclose(v, j)$. Note that these operations take constant time per neighbor.

Next, we show how report neighbors which come after $v$ in the preorder walk. Using the MP sequence, we can find the potential neighbors of $v$: we scan sequence from the position after $v$ and report every vertex (open parenthesis) until we observe the first open parenthesis of the same type as $v$. Let $w$ be such parenthesis, we jump to the matching parenthesis of $w$ using $m_findmatch_{MP}(w)$ in constant time, and continue this until we see the close parenthesis matching $v$. Therefore, in the tree decomposition, we skip the subtrees in which $v$ has been overwritten by $w$.

The potential neighbors of each vertex form *segments* of consequent vertices in the preorder walk. A segment of type $i$ is a range of elements in the MP sequence bordered by two parenthesis of type $i$. The bordering parenthesis can be




(a) The MP sequence in which the segments of type 1 are highlighted. The circles show those associated to vertex $A$




(b) The contracted parenthesis of type 1 ($C_1$)

**Fig. 4.** The ignore sequence for vertex $A$ ($Ig_A$) is 100100. To find the 4th segment associated to $A$ in the MP sequence, we find the 3rd child of $A$ in $C_i$ (the starred vertex in (b)), its matching parenthesis and the one after (arrowed ones) are the boundaries in the contracted sequence, which can be mapped to the MP sequence.

open or closed, and their segment can be empty if they are adjacent in the MP sequence (Figure 4(a)). Note that any segment is associated to exactly one vertex, which is a vertex of the same type which encloses it. To report actual neighbors in a given segment associated to vertex $v$, we successively apply $r_successor_M$ on the row of the same type of $v$ in the big table to find all ones in the range of the segment. So, we can report the neighbors inside a segment in constant time per neighbor. We also need to address how to select the appropriate segments. We say the segment is *good* if it includes at least one actual neighbor of $v$, and it is *bad* otherwise. Note that there may be a non-constant number of bad segments associated to a vertex, and we cannot probe all of them. For each vertex $v$, we define a bitmap $Ig_v$ where $Ig_v(i)$ determines whether the $i$th segment associated to $v$ is good ('1') or bad ('0'). We store an *ignore sequence IG* as follows: read vertices in preorder, for each vertex $v$ write down a '2' followed by the sequence $Ig_v$. The result would be a sequence of size $3n–k$ on alphabet $\{0, 1, 2\}$, which can be stored using $O(n)$ bits to support *select* in constant time [13]. To see why the size of $IG$ is $3n-k-1$, note that there $2n_i - 1$ segments of type $i$ where $n_i$ is the number of vertices with type $i$, so there are totally $2n-(k+1)$ segments. Since each segment is associated to exactly one vertex, the size of $IG$ is $2n-(k+1)+n$. Note that $Ig_v$ is the subsequent between $select_{IG}(i, 2)+1$ and $select_{IG}(i+1, 2)$, in which $i$ is the index of $v$ in preorder walk.

Using the ignore sequence, we can distinguish the index of good segments among all segments associated to a vertex $v$. Next, we need to locate these segments in the MP Sequence and use the map structure to locate the range of the segment in the big table. For each type $i$, we store a *contracted parenthesis* of type $i$, denoted by $C_i$, as a copy of the MP sequence in which all parenthesis except those of type $i$ are deleted. The result would be a balanced parenthesis sequence, equivalently an ordered tree, for each type. The total size of these trees is equal to $n$ and we need $2n + o(n)$ bits to represent them. Assume we need to locate the $t$'th segment of vertex $v$ in the MP sequence, and let $i$ be the type of $v$. If $t=1$, the desired segment starts with the parenthesis representing $v$ and ends

with the next parenthesis of the same type, which can be found in constant time. If $t > 1$, we locate the segment in the contracted parenthesis sequence and then map it into the MP sequence. First we locate $v$ in the contracted parenthesis, using $v_c = select_{C_i}(x, '(')$ where $x$ is the rank of $v$ among vertices of the same type, i.e., $x = rank_{MP}(v, '(_i')$. Observe the $t$'th segment of $v$ starts after the close parenthesis matching the open parenthesis representing $t$'th child of $v$ in the contracted parenthesis (Figure 4(b)). So we apply $\alpha = child_{C_i}(v_c, t)$ and $\beta = findmatch_{C_i}(\alpha)$ to find $\beta, \beta{+}1$ as the two neighboring parenthesis of type $i$ which bound segment $t$ in the contracted parenthesis. Using *rank* and *select*, respectively on $C_i$ and MP we can locate these parenthesis in the MP sequence.

To summarize, to report neighbors of vertex $v$ which succeed $v$ in the preorder walk, we use the ignore sequence to find the indices of good segments among all segments associated to $v$. We use contracted parenthesis to find the actual position of the good segments in the MP sequence, and use map structure to find the range of the segments in the big table. Using *r_successor* operation in the big table we can report neighbors in constant time per neighbor.

The additional space used for supporting neighbor report are ignore sequence and contracted parenthesis, which are both stored in $O(n)$ bits. The index used for degree request needs $n \lg k + o(nk)$ bits, and there is no additional index for adjacency queries (details are presented in the long version of the paper). Together with the main structures (the MP sequence and the big table), the size of the oracle would be $k(n + o(n) - k/2) + O(n)$.

**Theorem 2.** *Given a graph of size $n$ and treewidth $k$, an oracle is constructed to answer degree, adjacency, and neighborhood queries in constant time. The storage requirement of the oracle is optimal to within lower order terms.*

## 5 Distance Oracles

To give a distance oracle, we obtain the height-restricted tree decomposition $T$ of the input graph $G$ of treewidth $k$ using Lemma 1. Let $k'$ denote the maximum number of vertices in each node of $T$. Since treewidth of $T$ is at most $3k + 2$, we have $k' \leq 3k + 3$. We define the weight of a node as the number of vertices introduced in that node. Correspondingly, we define the weight of a subtree as the sum of the weights of nodes in the subtree. There are two recursive decompositions of $G$ into smaller subgraphs using its height-restricted tree decomposition $T$ using the following lemma (proof in the long version of the paper):

**Lemma 5.** *For any parameter $1 \leq L \leq n$, a tree with $n$ nodes such that the weight of nodes is at most $k'$ can be decomposed into $\Theta(n/L)$ subtrees of weight at most $2L + k'$ which are pairwise disjoint form their roots. Furthermore, aside from edges stemming from the component root nodes, there is at most one edge leaving a node of a component to its child in another component.*

In the first phase, tree $T$ is decomposed into smaller subtrees $T_1, T_2, \ldots$ using Lemma 5 with value $L = k' \lg^3(n)$ (the first phase is skipped if $L \geq n$). Let $V_i$

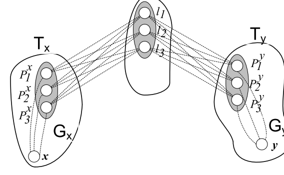



**Fig. 5.** Distance oracle: computing the distance between $x$ and $y$

be the set of graph vertices that occur in a node in subtree $T_i$. We define $G_i$ as the subgraph of $G$ induced on $V_i$.

Lemma 5 guarantees that there are at most two nodes of each subtree $T_i$ that are connected via a tree edge to other subtrees; we refer to these tree nodes as *portal* nodes. These nodes collectively contain $2k'$ graph vertices. We refer to these vertices as the *portal* vertices of $G_i$, and denote this set of vertices by $P_i$.

To reduce the distance oracle to within $G_i$'s, we explicitly store the distance from each portal vertices to all vertices in an ancestor node of the corresponding portal nodes. Namely, for each vertex $v \in P_i$ in a portal tree node $t$ and vertex $u$ in a node an ancestor of $t$, we explicitly store the distance between $v$ and $u$. Since the height of the tree is $O(\log n)$, there are $O(k' \log n)$ such vertices as $u$. The storage requirement of this list in number of bits is $O\left(\frac{n}{k' \lg^3 n} k' \left(k' \log(n)\right) \log(n)\right) = o(kn)$.

We also take the projection of tree $T$ on portal nodes by adding an edge between two portal nodes if and only if the path in $T$ between them does not contain another portal node. The projected tree is a tree on $O\left(n/(k' \log^3(n))\right)$ nodes. We preprocess and store the tree (in $O\left(n/(k' \log^3(n))\right)$ bits) to be able to answer lowest common ancestor queries in constant time [15].

If we can internally in any $G_i$ determine the distance between any two vertices $s, t \in G_i$, then using the explicitly stored distances for portal vertices, we can determine the distances globally between any two vertices in $G$. Given two vertices $x, y$, we determine the subgraphs $G_x, G_y$ they belong in. We use a *rank/select* structure to accomplish this task in constant time and $o(n)$ storage. We compute the distances from $x$ to the portal vertices $p_1^x, \ldots, p_{2k'}^x$ in $G_x$ and analogously the distances from $y$ to the portal vertices $p_1^y, \ldots, p_{2k'}^y$ of $G_y$ (see Figure 5). Let $T_x$ and $T_y$ be the subtrees corresponding to $G_x, G_y$. We determine in constant time the lowest common ancestor $L$ of the roots of $T_x$ and $T_y$. Portal vertices have their distances to vertices introduced in their ancestors explicitly stored. Therefore, $p_1^x, \ldots, p_{2k'}^x$ and also $p_1^y, \ldots, p_{2k'}^y$ have their distances to vertices $l_1, \ldots, l_{k'}$ in the bag of node $L$ stored. Without loss of generality, we assume the harder case where roots of $T_x, T_y$ are not an ancestor of each other, the details of the other case is deferred to the full version of this paper.

We repeat the previous step for each $G_i$ by applying Lemma 1 to obtain a height-restricted tree decomposition and using Lemma 5 with value $L = k' \lg^2(k')(\lg\lg(n))^3$ to obtain smaller subgraphs $G_i'$. Additionally, we store the distance between each second-level portal vertex and all first-level portal vertices contained in the same subgraph $G_i$. This structure allows us to reduce the problem to within second-level subgraphs $G_i'$, without paying a factor of $k'$ for the

query time. The space requirement of this structure can be analyzed similarly to $o(kn)$. We repeat the step for a final time using value $L = k' \lg^2(k')(\lg\lg\lg(n))^3$ to obtain tiny subgraphs $G''_i$. Hence, the problem reduces to computing the distances of a vertex to third-level portal vertices confined to an individual third-level graph $G''_i$. As $G''_i$'s are subgraphs of the original graph, their treewidth is at most $k$. The corresponding tree decomposition for these graphs can be obtained trivially by projecting from the tree decomposition of the original graph. Hence, treewidths of $G''_i$'s are $k$ and not $k'$ any further.

We distinguish two cases according to the value of $k$. For smaller values of $k$ where $\lg k \leq (\lg\lg\lg n)^3$, the size of third-level subgraphs $G''_i$ is very small, therefore, we use a look-up table to catalog all graphs with $p$ vertices and treewidth $k-1$ such that $p < \lg(n)/(2k)$. We exhaustively list answers to all distance queries together with each graph. The representation of Section 4 bounds the number of such graphs and consequently the size of the table is $o(n)$. A third-level graph $G''_i = (V''_i, E''_i)$ is represented by an index to within the look-up table and therefore, the space requirement of each $G''_i$ matches the entropy of graphs with $|V''_i|$ vertices and treewidth $G''_i$ (we note that every subgraph of a graph with treewidth $k$ has treewidth at most $k$). Since $\sum_i |V''_i| = n + o(n)$, the distance oracle requires space which matches the entropy of graphs with treewidth $k$ to within lower order terms. Distances in $G''_i$ are read in constant time from the table and there is an additive overhead of $O(k^2)$ for each level of recursion. Thus, the total distance query time is $O\left(k^2\right)$. Therefore, the distance query performs in constant time when $k$ is constant.

For larger values of $k$, where $\lg k > (\lg\lg\lg n)^3$, we simply store third-level graphs $G''_i = (V''_i, E''_i)$ using the navigation oracle representation of Section 4 to store each $G''_i$ in $k(|V''_i| + o(|V''_i|) - k/2) + O(|V''_i|)$ bits. Since $\sum_i |V''_i| = n + o(n)$, the total storage requirement for distance oracle in this case is $k(n + o(n) - k/2) + O(n)$. In order to determine the distance of a vertex in $G''_i$ to the third-level portals of $G''_i$, we simply perform a breadth first search (BFS). The time of performing a BFS is $O\left(k^2 \lg^2(k)(\lg\lg\lg(n))^3\right)$ which in this case is $O\left(k^2 \log^3(k)\right)$. This dominates the overhead of $O\left(k^2\right)$ from recursion, and hence distance queries perform in $O\left(k^2 \log^3(k)\right)$ time.

**Theorem 3.** *Given an unlabeled, undirected, and unweighted graph with $n$ vertices and of treewidth $k$, an exact distance oracle is constructed to answer distance queries in time $O\left(k^2 \log^3 k\right)$. The storage requirement of the oracle is optimal to within lower order terms.*

## 6 Conclusion

We considered the problem of preprocessing a graph small treewidth to construct space-efficient oracles that answers a variety of queries efficiently. We gave a navigation oracle that answers navigation queries of adjacency, neighborhood, and degree queries in constant time. We also proposed a distance query which reports the distances of any pair of vertices in $O\left(k^2 \log^3 k\right)$ where $k$ is the (determined) treewidth. By way of an enumerative result, we showed the space requirements of the oracles are optimal to within lower order terms.

## References


1. Jérémy Barbay, Luca Castelli Aleardi, Meng He, and J. Ian Munro. Succinct representation of labeled graphs. In *Proc. 18th International Conference on Algorithms and Computation*, ISAAC'07, pages 316–328, 2007.
2. Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *Proceedings of 14th ACM-SIAM Symposium on Discrete Algorithms*, SODA'03, pages 679–688, 2003.
3. Guy E. Blelloch and Arash Farzan. Succinct representations of separable graphs. In *Proc. 21st Conference on Combinatorial Pattern Matching*, CPM'10, pages 138–150, 2010.
4. Hans L. Bodlaender. NC-algorithms for graphs with small treewidth. In *Proc. 14th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG'88, pages 1–10, 1989.
5. Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.
6. Hans L. Bodlaender. Treewidth: Characterizations, applications, and computations. In *Proc. 32nd International Workshop on Graph-Theoretic Concepts in Computer Science*, WG'06, pages 1–14, 2006.
7. Timothy M. Chan. All-pairs shortest paths for unweighted undirected graphs in *o(mn)* time. In *Proc. 17th ACM-SIAM Symposium on Discrete Algorithm*, SODA'06, pages 514–523, 2006.
8. Vida Dujmovic and David R. Wood. Graph treewidth and geometric thickness parameters. *Discrete Comput. Geom.*, 37:641–670, 2007.
9. Arash Farzan. *Succinct Representation of Trees and Graphs.* PhD thesis, School of Computer Science, University of Waterloo, 2009.
10. Arash Farzan and J. Ian Munro. Succinct representations of arbitrary graphs. In *Proc. 16th European Symposium on Algorithms*, ESA'08, pages 393–404, 2008.
11. Cyril Gavoille and Nicolas Hanusse. On Compact Encoding of Pagenumber $k$ Graphs. *Discrete Mathematics & Theoretical Computer Science*, 10(3):23–34, 2008.
12. Cyril Gavoille and Arnaud Labourel. Shorter implicit representation for planar graphs and bounded treewidth graphs. In *Proc. 15th European Symposium on Algorithms*, ESA'07, pages 582–593, 2007.
13. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th ACM-SIAM symposium on Discrete algorithms*, SODA'03, pages 841–850, 2003.
14. Frank Harary, Robert W Robinson, and Allen J. Schwenk. Twenty-step algorithm for determining the asymptotic number of trees of various species: Corrigenda. *Journal of the Australian Mathematical Society*, 41(A):325, 1986.
15. Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct ordinal trees based on tree covering. In *Proc. 34th international colloquium on Automata, Languages and Programming, Part I*, volume 4596 of *ICALP'07*, pages 509–520. 2007.
16. Igor Nitto and Rossano Venturini. On compact representations of all-pairs-shortest-path-distance matrices. In *Proc. 19th symposium on Combinatorial Pattern Matching*, CPM'08, pages 166–177, 2008.
17. Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007.