Space-Efficient Depth-First Search via Augmented Succinct Graph Encodings

Michael Elberfeld
THM, University of Applied Sciences Mittelhessen, Gießen, Germany

Frank Kammer
THM, University of Applied Sciences Mittelhessen, Gießen, Germany

Johannes Meintrup
THM, University of Applied Sciences Mittelhessen, Gießen, Germany

Abstract

We call a graph G separable if a balanced separator can be computed for G of size $O(n^\epsilon)$ with $\epsilon < 1$. Many real-world graphs are separable such as graphs of bounded genus, graphs of constant treewidth, and graphs excluding a fixed minor. In particular, the well-known planar graphs are separable. We present a succinct encoding of separable graphs G such that, after the encoding is computed, any number of depth-first searches (DFS) can be performed from any given start vertex, each in $o(n)$ time and $o(n)$ bits in the word RAM model. After the execution of a DFS, the succinct encoding of G is augmented such that the DFS tree is encoded inside the encoding while maintaining succinctness. Afterward, the encoding provides common DFS-related queries in constant time. These queries include queries such as lowest-common ancestor of two given vertices in the DFS tree or queries that output the lowpoint of a given vertex in the DFS tree. Furthermore, for planar graphs, we show that the succinct encoding can be computed in $O(n)$ bits and expected linear time, and a compact variant can be constructed in $O(n)$ time and bits. For other separable graph classes $\mathcal{G}$ the runtime and space usage depends on the specific algorithms used to find balanced separators in graphs of $\mathcal{G}$.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Graph algorithms analysis

Keywords and phrases Depth-First Search, Succinct, Space Efficient, Separable Graphs, Planar Graphs, Table Lookup, r-Division

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2025.29

Related Version *Extended Version*: `https://arxiv.org/abs/2504.19547`

1 Introduction

Depth-first search (DFS) in graphs forms the backbone of algorithms for a number of applications like finding vertex or edge cuts. A depth-first search implicitly computes a tree (or forest), over the vertices of the graph that is called a *DFS tree* (or forest), which has crucial structural properties that are commonly used for applications. DFS that runs in linear time in the number of vertices and edges utilizes two folklore data structures: a stack to keep track of the current paths into the graph and an adjacency list to efficiently iterate over the neighbors of a given vertex. Standard implementations of this approach use $\Theta(n \log n)$ bits of space for the stack and $O(\log n)$ bits for each vertex identifier, where n refers to the number of vertices in a given graph. Lowering the space requirements for depth-first search to $O(n)$ bits while still maintaining a (nearly) linear runtime was the aim of a series of works during the last years: one of the first algorithms is due to Asano et al. [3] who reduced the space to $O(n)$ bits, but increased the runtime to $O(m \log n)$, where m refers to the number of edges in a given graph. After several improvements [4, 7, 9] to this result, Hagerup [13, Theorem 5.4] presented the current state-of-the-art algorithm using $O(n)$ bits and $O(n + m \log^* n)$ time.

In many circumstances, we do not need to be able to handle input graphs of any kind. Instead, the input graphs have a special structure that we can utilize to devise algorithms that are more efficient than the ones handling the general case. Typical examples are planar graphs, which can be drawn in the plane without edge crossings, or generalizations of it like graphs of bounded genus or graphs excluding a fixed minor. What these graphs have in common is that they are sparse, meaning $m = O(n)$. As there exists a DFS that uses $O(n + m)$ time and bits on general graphs, as stated by Hagerup [13, Theorem 4.1] and Banerjee et al. [4, Lemma 2], we can execute a DFS with $O(n)$ time and bits on sparse graphs. If we shift our attention to $o(n)$-bit DFS algorithms, we can observe some strong indications from complexity that no sublinear space polynomial time algorithm can exist for a special variant of DFS, called *lexicographical DFS* [3, 9]. A concrete sublinear space algorithm for this problem restricted to planar graphs has an unwieldy massive polynomial runtime [18], due to the reliance on the logspace reachability result of Reingold [27].

As mentioned above, computing a DFS is often just the beginning or part of an algorithm solving more involved graph problems like computing biconnected and strongly-connected components. Typically, these applications of DFS store certain meta information that is computed while traversing the graph. Examples are *pre-order numbering* that numbers the vertices in the order of their exploration [28], and *lowpoints* that help to identify vertex and edge cuts [17]. If we want to extend existing approaches for computing DFS that are efficient in both space and time to this more general setting, we can not easily store the meta information of a vertex using any standard encoding. Again, this would use $\Theta(\log n)$ bits for each vertex and, hence, creates a total memory footprint of $\Theta(n \log n)$ bits. While there exists space-efficient techniques for, e.g., computing pre-order numbers or lowpoints [7, 13, 20], they require significantly more resources than the space-efficient DFS variants, e.g., the algorithm Chakraborty et al. [7] for computing lowpoints uses $\Theta(n)$ bits and $\Theta(m \log n \log \log n)$ time.

In this paper we present both (1) a data structure that is able to keep meta information we compute for the vertices of a graph space-efficiently and (2) an approach for computing DFS traversals space-efficiently as well as extensions to some common applications. Our results are applicable to classes of separable graphs which have balanced separator size $O(n^\epsilon)$ for some $\epsilon < 1$. This covers, in particular, planar graphs, graphs of bounded genus and graphs excluding a fixed minor.

Data Structure: Nested Divisions and Augmentations. An encoding of an element from a universe U is called *succinct* (*compact*) if it uses $Z + o(Z)$ ($Z + O(Z)$) bits where $Z = \log_2 |U|$. In this paper we present a succinct encoding for separable graphs (precisely defined in Section 2.1), such as planar graphs, that provides the following functionalities. After the encoding is initialized, a DFS can be executed (from any given start vertex) in $o(n)$ time and $o(n)$ bits and afterward information of the executed DFS can be queried in constant time. If needed, a new DFS can be computed at any given time. Such information includes the pre-order numbering and lowpoints of a vertex, and lowest-common ancestor of two vertices. Furthermore, for planar graphs, we show that the succinct encoding can be computed in expected linear time with $O(n)$ bits used during the construction step, and a compact variant can be constructed in $O(n)$ time and bits.

The data structure we present, called *succinct nested division*, extends a succinct encoding of separable graphs by Blelloch and Farzan [6]. The encoding of Blelloch and Farzan is built on dividing the input graph into smaller "mini pieces". Mini pieces are, in turn, divided into even smaller "micro pieces" that are small enough such that relevant structural information about them can be pre-computed and kept in a lookup table. The key property of this

approach is that there are only few "boundary vertices" in each piece that are contained in multiple pieces. This suggests algorithms that, like the data structure itself, switch between these three levels. In fact, different algorithmic approaches are needed for different levels to maintain our target space and time bounds.

We augment the encoding of Blelloch and Farzan with additional data structures and show that this can be used to design more complex queries than the standard graph access queries provided by them. Intuitively, the boundary vertices act like relays since any interaction between two non-boundary vertices in different pieces must necessarily pass through them. Consequently, long-range dependencies (e.g., long paths), are efficiently mediated by the sparse set of boundary vertices. Hence, pieces can often be considered in isolation with few interactions between them, and all interactions between vertices of different pieces are communicated via boundary vertices. To capture the idea we intuitively call a property π *strongly local* if, for all non-boundary vertices u, $\pi(u)$ can be evaluated only with information stored with the boundary vertices together with a small amount of information that is directly encoded in P, typically for us there exists a boundary vertex v of P such that $|\pi(u) - \pi(v)| = O(|P|)$.

For circumventing the trivial lower bound of $\Omega(n \log n)$ on the number of bits required for encoding members of a family of labeled graphs with n vertices, we work with so-called unlabeled graphs. This means that we are able to choose our own vertex labels for the encoding so that we can construct a succinct encoding for separable graphs that uses $\Theta(n)$ bits [6]. Such a setting is very common, e.g., used by Blelloch and Farzan's original encoding [6] as well as other works [1, 2, 8, 11, 19, 22, 25].

Algorithms: Depths-First Search and Applications. Our succinct encoding for separable graphs is constructed such that a DFS can be performed directly on the encoding from any given starting vertex. Afterward, various queries regarding the DFS traversal are available. In particular, we directly provide the necessary queries that, e.g., Hopcroft and Tarjan's biconnected-component algorithm [17], or Schmidt's algorithm for chain decompositions [28] require. We effectively provide an interface that allows us to implement typical standard algorithms without the need to design specialized techniques. The following theorem summarizes our main results. Note that the runtime and bits are sublinear once the encoding is computed. The operations presented in it are only a set of examples of queries that have the previously outlined intuition of strongly local, chosen such that the previously mentioned standard algorithms can be executed directly. Without loss of generality, we assume that all graphs we work with are connected. If a given graph is not connected, one can apply all our results to each connected component separately.

▶ **Theorem 1.** *Let G be a separable graph. There exists a succinct encoding $\mathcal{D}$ of G that provides neighborhood iteration, adjacency and degree queries in constant time per element output. Additionally, it provides the operation* $\mathrm{dfs}(u)$ *that executes a DFS in $O(n/\operatorname{poly}(\log\log n)) = o(n)$ time and $o(n)$ bits from a given start vertex u such that at all times the encoding remains succinct. After the execution of a DFS the following queries are available in constant time for the last computed DFS tree.*

- $\mathrm{children}(u)$ *iteration over all children of u.*
- $\mathrm{parent}(u)$*: output the parent of u.*
- $\mathrm{pre}(u)/\mathrm{post}(u)$*: return the pre-/post-order number of vertex u.*
- $\mathrm{num\text{-}descendants}(u)$*: return the number of descendants of u.*
- $\mathrm{depth}(u)$*: return the depth of u.*
- $\mathrm{lowpoint}(u)$*: return the lowpoint of u.*
- $\mathrm{lca}(u, v)$*: return the lowest-common ancestor of vertices u and v.*

Replacing succinct with compact in Theorem 1, and additionally restricting G to be a planar graph, the encoding can be constructed with $O(n)$ bits and in $O(n)$ time. The change from succinct to compact is due to our wish to avoid the use of a certain succinct dictionary data structure of Raman et al. [26] relying on hashing functions that can only be computed in expected linear time. When we replace the dictionary with a simpler variant of Baumann and Hagerup [5], we obtain a compact encoding that can be computed much more easily. When we are fine with expected linear time, we can use the dictionary of Raman et al., and thus can construct the succinct encoding in $O(n)$ bits and expected $O(n)$ time. Subsection 3.3 presents the details regarding the construction steps. For other graph classes the construction time and bits depend on the runtime of the respective separator algorithms used as subroutines to divide the input graph.

▶ **Corollary 2.** *Let G be a planar graph. There exists a compact (succinct) encoding of G that can be constructed in $O(n)$ time (expected time) and $O(n)$ bits that provides the same functionalities as the encoding of Theorem 1.*

Related Work on Graph Divisions. Recursive divisions have been used as the basis for algorithms and data structures for decades, commonly based on the so-called r-division for planar graphs, defined precisely in the next section. These divisions build on a recursive application of the linear-time separator algorithm of Lipton and Tarjan [24], and the improvement of Goodrich [12] who showed that the entire recursively application can be executed in linear time – a standard approach would use $O(n \log n)$ time. Recursive divisions have been successfully applied to algorithmic problems such as maximum flow and minimum cut and all-pairs shortest path [10]. Applications in the field of data structures include decremental data structures for connectivity [15, 16] where decremental refers to modifications of a graph that remove vertices or edges. None of the mentioned results have a focus on space-efficiency, and internally they use other techniques compared to our approach.

Structure of the Paper. Section 2 details our first contribution, the nested-division data structure and augmented variants of it. Subsequently, in Section 3, we present the sublinear-space and time execution of DFS and related applications, executed directly on our data structure.

2 Nested Divisions

The present section contains an overview of what we view as a nested division, beginning with the graph-theoretic properties they provide us with (Subsection 2.1), and continuing how Blelloch and Farzan used nested divisions as a data structure to encode separable graphs (Subsection 2.2). Finally, we extend the section with our ideas of abstract augmentations in nested divisions we provide to allow complex queries (Subsection 2.3). We use the notation $[i]$ for any natural number i to refer to the set $\{1, \ldots, i\}$.

2.1 Graph-Theoretic Properties of Nested Divisions

We describe a slightly generalized variant of the well-known concept of r-divisions sketched in the introduction. We start with a small set of definitions. A *balanced separator* of a graph $G = (V, E)$ is a set of vertices $S \subset V$ such that each connected component of $G[V \setminus S]$ contains at most a constant fraction of the vertices. The exact constant does not matter to us. We call a class of graphs $\mathcal{G}$ that is closed under taking vertex induced subgraphs

$O(n^\epsilon)$-*separable* if there exists a constant ϵ with $0 < \epsilon < 1$, such that each $G = (V, E) \in \mathcal{G}$ has a balanced separator $S \subset V$ of size $O(n^\epsilon)$. We simply say that a graph or class of graphs is *separable* if the size of the balanced separator is of no particular concern to us.

▶ **Definition 3** (Relaxed Division). *Let G be a graph belonging to an $O(n^\epsilon)$-separable class of graphs for some $0 < \epsilon < 1$. An* (α, r)-relaxed division *is a decomposition of G into a collection $\mathcal{P}$ of subgraphs, called* pieces*, satisfying:*

1. *Each piece is a subgraph with at most r vertices such that there are $\Theta(n/r)$ pieces in total.*
2. *Every edge is assigned to exactly one piece (containing both endpoints of the edge).*
3. *Each piece has $O(\alpha r^\epsilon)$* boundary vertices*, which are vertices shared between different pieces.*

We call α the relaxation *and r the* piece size*. Edges between two boundary vertices are called* boundary edges*, edges between two non-boundary vertices* non-boundary edges *and all other edges* transitional edges*.*

See Figure 1 for a sketch of such a division. The reason we introduce the relaxation α, is due to our desire to achieve space-efficiency when algorithmically constructing a division. There exists an algorithm that computes a $(\log n, r)$-relaxed division via recursive separator searches that uses only $O(n)$ bits and, for planar graphs, $O(n)$ time [21]. If the construction step of our encoding must not be space-efficient, one can of course use a standard non-space efficient algorithm [12, 23], and therefore have a relaxation factor of $\alpha = 1$; for our application a factor of $\alpha = \log n$ allows for a more (space-)efficient computation of our final encoding, buy when using the encoding makes no difference. Common applications of divisions use them in a recursive fashion, and so do we. In particular, we construct for each piece of some (α, r)-relaxed division a $(1, \tilde{r})$-relaxed division where $\tilde{r} \ll r$. For this, we define a *nested division* for a separable graph G as follows.

▶ **Definition 4** (Nested Division). *Let G be a graph belonging to an $O(n^\epsilon)$-separable class of graphs for some $0 < \epsilon < 1$. An* $(\alpha, r, \tilde{r})$-nested division *of G is an (α, r)-relaxed division of G into pieces $\mathcal{P} = \{P_1, P_2, \ldots\}$ such that for each piece $P_i \in \mathcal{P}$ we have a $(1, \tilde{r})$-relaxed division $\mathcal{P}_i$. We call pieces of $\mathcal{P}$* mini pieces *and pieces of some $\mathcal{P}_i$* micro pieces*.*

If we have an $(\alpha, r, \tilde{r})$-nested division for a graph G each vertex can be viewed as being part of three *levels*:

- *graph*: Each vertex is part of V.
- *mini*: Each vertex of V is assigned to one or more mini pieces.
- *micro*: Each vertex of V is assigned to one or more micro pieces.

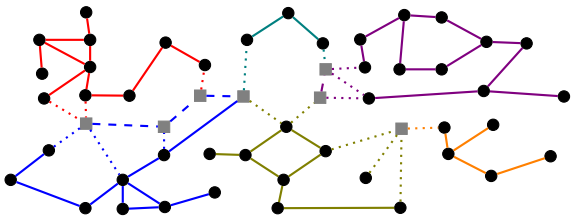

Figure 1 A sketch of a division of graph into 6 pieces of at most 13 vertices. The edges of each piece are colored with a distinct color. Boundary vertices are gray squares, and non-boundary vertices black circles. Non-boundary edges are solid, transitional edges dotted, and boundary edges dashed.

Given an $(\alpha, r, \tilde{r})$-nested division for a graph G we can categorize all vertices as follows. We call a vertex *mini (micro) boundary* if it is a boundary vertex in a mini (micro) piece, and *mini (micro) non-boundary* if it is not a boundary vertex in a mini (micro) piece. Edges are categorized similarly: an edge is called *mini (micro) boundary edge* if both endpoints are mini (micro) boundary vertices, *mini* (micro) non-boundary edge if both endpoints are mini (micro) non-boundary vertices, and *mini (micro) transitional edge* otherwise. We routinely drop the specifier mini or micro if we make statements that apply to both mini and micro levels. We are interested in bounding the number of total occurrences of mini (micro) boundary vertices among all mini (micro) pieces, referred to as *mini (micro) duplicates*. Let $\mathcal{P}$ be an (α, r)-relaxed division constructed for an $O(n^\epsilon)$-separable graph G. As each piece has $O(r^\epsilon)$ boundary vertices, the total number of duplicates is $O((n/r)(\alpha r^\epsilon))$. Thus, for a nested division we have $O(\alpha n/r^{1-\epsilon})$ (duplicates of) mini boundary vertices, and $O((n/r)(r/\tilde{r})(\tilde{r}^\epsilon)) = O(n/\tilde{r}^{1-\epsilon})$ (duplicates of) micro boundary vertices. We refer to a more detailed description of the reasoning behind these bounds to Blelloch and Farzan [6]. We summarize it in the following lemma.

▶ **Lemma 5** ([6]). *Let $\mathcal{D}$ be a $(\alpha, r, \tilde{r})$-nested division constructed for an $O(n^\epsilon)$-separable graph G with relaxation parameters α, and piece sizes r and $\tilde{r}$. Then there are $O(\alpha n/r^{1-\epsilon})$ (duplicates of) mini boundary vertices and $O(n/\tilde{r}^{1-\epsilon})$ (duplicates of) micro boundary vertices.*

For all of our use cases of $(\alpha, r, \tilde{r})$-nested divisions we have $\alpha = O(\log n)$, $r = \text{poly}(\log n)$ and $\tilde{r} = \text{poly}(\log \log n)$, with the exact polynomial chosen such that we have $O(n/\text{poly}(\log n))$ (duplicates of) mini boundary vertices, and $O(n/\text{poly}(\log \log n))$ (duplicates of) micro boundary vertices. For the remainder of our paper, when we refer to a *nested division* we refer to a $(\log n, \text{poly}(\log n), \text{poly}(\log \log n))$-nested division.

2.2 Nested Divisions as Data Structures

As we now describe concrete data structures, we fix our model of computation to the word RAM model with word size $\Omega(\log n)$. For the rest of the section, let G be a separable graph and assume that a nested division is given for G. We begin by outlining the most basic functionality we require for a nested division when used as a data structure to encode a graph. Blelloch and Farzan presented a succinct encoding for unlabeled separable graphs that effectively uses a nested division at its core, and the framework we present in this subsection is based on their work [6], defined slightly more general to allow for our later augmentations. While they do not use an α-relaxation factor for the initial (α, r)-relaxed division of the nested division (they use their own variation of a standard recursively constructed r-division), their framework works with $\alpha = \log n$ without any modifications. Their general idea is to succinctly encode the nested division, and encode each piece at the micro level as an index into a lookup table. Boundary vertices are encoded via additional data structures that use standard non-space efficient techniques, e.g., using arrays and lists. Combined with a succinct bidirectional mapping that maps a vertex to its different occurrences in each level (graph, mini and micro) they realize neighborhood, adjacency and degree queries.

To describe the functionality of these bidirectional mappings, we introduce a labeling of pieces and vertices. Each mini piece $P_i \in \mathcal{P}$ is uniquely identified by an id i. Analogously, the relaxed division constructed for each P_i is identified as $\mathcal{P}_i$. Each micro piece $P_{i,j}$ is identified by a tuple (i, j) with j indicating it is the piece with id j, i.e., piece $P_{i,j} \in \mathcal{P}_i$. Each vertex $u \in V$ has a *graph label* assigned from $[n]$, a *mini label* u_i assigned from $[r]$ in each mini piece P_i it is contained, and a *micro label* assigned from $[\tilde{r}]$ in each micro piece $P_{i,j}$ it is contained. When we talk about a vertex $u \in V$ we always assume that u is identified by its graph label.

The mappings that are provided are as follows: given a vertex $u \in V$ output the tuples (i, u_i) such u_i is the mini label of u in the piece P_i. Analogously for a given tuple (i, u_i) where i refers to a mini piece P_i and u_i is the label of some vertex of P_i, output the tuple $(j, u_{i,j})$ where $u_{i,j}$ is the micro label of vertex u_i in a micro piece $P_{i,j}$. Note that each of these queries can output more than one element if the vertex is a boundary vertex at the respective level. For less verbose writing we implicitly assume that we always have access to all mappings outlined in this paragraph and only make distinctions between the different types of labeling if necessary. We refer to the set of all outlined mappings as *translation mappings*.

▶ **Lemma 6** (Succinct Nested Division [6]). *Let $G \in \mathcal{G}$ be a graph and $\mathcal{G}$ a separable class of graphs. There exists a succinct data structure $\mathcal{D}$ of G that represents a nested division of G such that level mapping queries and level graph queries are provided in constant time.*

2.3 Augmenting Nested Divisions

In the following we want to augment the nested division of Lemma 6 with various capabilities. For this we analyze the runtime and memory budget we have when our goal is to run some algorithms in $o(n)$ time and with $o(n)$ bits, followed by outlining necessary techniques. As stated, we construct our nested divisions with $r = \text{poly}(\log n)$, $\tilde{r} = \text{poly}(\log \log n)$ and $\alpha = \log n$. This means that for each of the $O(n/\text{poly}(\log n))$ mini boundary vertices we have a memory budget of $\text{poly}(\log n)$ bits. Analogously, we have a budget of $\text{poly}(\log \log n)$ bits for each of the $O(n/\text{poly}(\log \log n))$ micro-boundary vertices. This means, we have a budget of $\text{poly}(\log n)$ bits for each mini piece and $\text{poly}(\log \log n)$ for each micro piece. These bounds allow us to use more space than what is needed for our applications, as we only require $O(\log n)$ bits per mini boundary vertex (and mini piece), and $O(\log \log n)$ bits per micro boundary vertex (and micro piece) for our applications. We have the same budget for the runtime as we do for memory, but for our application only require (amortized) constant time per boundary vertex.

For micro pieces we use a table lookup technique for constant time computations to subproblems. Micro pieces are so tiny, that any query we require can be described by a constant number of words in the word RAM model. We assume that we have a lookup table available that lists all graphs with at most $\tilde{r}$ vertices of the separable graph class $\mathcal{G}$ we work with. The table lists all graphs modulo isomorphism, i.e., no two entries are isomorphic. This is not enough for us, as we require additional data for the boundary vertices. Instead, we require the graphs of the lookup table to be *partially augmented*, meaning for at most $b = O(\tilde{r}^\epsilon)$ vertices of each graph G' of the lookup table we store a bit string (called a *partial vertex augmentation*) of some fixed length $\ell = O(\log \tilde{r})$, and an additional bitstring of length $O(b\ell)$ (called a *partial graph augmentation*).

Using a lookup table that lists all partially augmented graphs with at most $\tilde{r}$ vertices, we can realize what we refer to as *swapping indices*. Each micro piece is encoded as an index into the lookup table referencing some graph G' together with its partial augmentations. When we want to update values stored for (the micro boundary vertices of) the micro piece we concretely do this by changing the index we store. Let $P_{i,j}$ be some micro piece encoded as an index x into the lookup table. To update an augmented value in the micro piece, e.g., color a boundary vertex, we can replace the index x with a different index y. See Figure 2 for an example. We have to take care while constructing the table that any changes of the partial augmentation (i.e., the values stored for boundary vertices) does not change the internal labels of the graph, i.e., the names of the vertices stay the same independent of the changes we make to the data stored for boundary vertices. For the rest of our paper we always assume that we have access to such a lookup table.

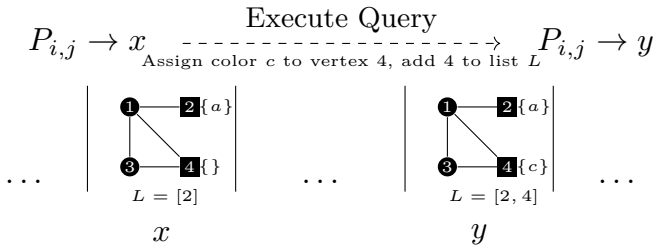

Figure 2 Initially the micro piece $P_{i,j}$ is stored as an index x of the table lookup, sketched below with two entries explicitly shown. The graph shown in the table has regular vertices shown as circles, and the vertices for which we can store binary strings shown as rectangles. We have a list L for each graph of the table that has enough capacity to store some values. We execute a query that takes as input the index x, the vertex label 4 and color c to assign to vertex 4. It returns the index y of the lookup table such that y stores the state that represents the execution of this query when applied to the graph stored at index x. We then swap the index x stored for $P_{i,j}$ with the index y in our succinct encoding.

3 Depths-First Search in Sublinear Time and Space

We present our techniques for executing a DFS directly on the succinct encoding. First, we show how the DFS is computed such that a so-called palm tree is encoded inside the nested division (Subsection 3.1), followed by our description of how to provide queries for so-called *strongly local* values (Subsection 3.2). As examples for strongly local values we describe the technical realization of providing queries regarding the meta-information of a DFS, e.g., pre-order numbering, lowpoints, lowest-common ancestors, and more. We end the section with a description of how our main results can be realized.

For this section, let $G = (V, E)$ be a separable graph given as a succinct nested division $\mathcal{D}$ (Lemma 6). We begin with the description of how to execute a DFS to compute a *palm tree* T that is a directed version of G such that the edges E are directed according to the DFS traversal, and by their direction partitioned into two sets, tree edges and back edges. Effectively, a palm tree is a DFS tree with additional back edges. Each vertex $u \in V(G)$ is assigned a *pre-ordering number* $\mathrm{pre}(u)$ indicating at which point in time it was traversed for the first time. Analogously, a *post-ordering number* $\mathrm{post}(u)$ is indicating at which point in the time DFS backtracked past the vertex u (i.e., all children of u have been fully explored). All edges (u, v) of the palm tree are directed such that, for a tree edge $\{u, v\}$ of the DFS tree, it holds that $\mathrm{pre}(u) < \mathrm{pre}(v)$, and for all other edges $\{u, v\}$ (the back edges), it holds that $\mathrm{pre}(v) < \mathrm{pre}(u)$. In the next subsection we compute some palm tree T directly on $\mathcal{D}$ and afterward provide constant time queries regarding T such as iteration over all children of a vertex u . If we are only interested in the directed tree edges of T, we refer to it as the DFS tree.

3.1 Iterator based DFS to compute a Palm Tree

In addition to the standard stack-based approach for implementing a DFS, there is also an iterator approach that stores for each vertex an iterator of the adjacency list. We implement the second variant. Let T be the palm tree constructed by some DFS traversal. See Figure 3 for an example of the following description. We say T *enters* a piece P if the there exists tree edges (u, v) and (v, w) with only the edge $\{v, w\}$ part of the piece P, or if edge $\{v, w\}$ is part of the piece P, (v, w) is a tree edge, and v is the root of T. We then call v an *entry vertex of*

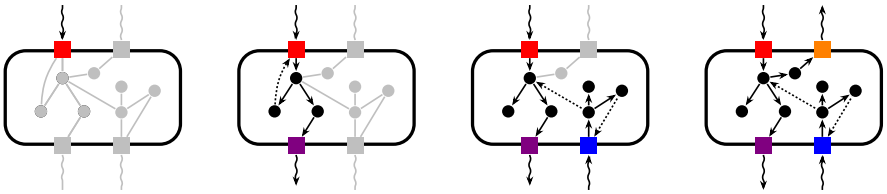

Figure 3 The figure shows four DFS states of some piece P. In the first state, no vertex inside P is visited yet, and we have just arrived at a boundary vertex of P (colored red). We then advance the DFS until the next boundary vertex is visited, colored violet. For micro pieces, we advance it via one query to the lookup table. We later arrive at the same piece again, this time via the blue boundary vertex. We enter the piece, and encounter the situation of the null-exit, thus we fully explore all vertices belonging to the subtree rooted at the blue vertex. Finally, we backtrack to the piece P via the violet vertex, and find a part of the piece not yet visited. We advance the state until the orange boundary vertex is visited and leave the piece again. The entry-exit pairs of the piece are (•, •), (•, `null`) and (•, •). Tree edges are shown as solid lines, and back edges as dotted lines.

P. Furthermore, we say T *exits* P if there exist tree edges (u, v) and (v, w) with only $\{u, v\}$ part of P and $\{v, w\}$ not part of P. We then call v an *exit vertex of* P. We say T *starts* at a piece P if the tree edge (u, v) exists with u the root of T, v is the first vertex visited after u and $\{u, v\}$ is part of P. Note that the entry of a piece is by definition a boundary vertex of some piece P, or the root of T. For us there is no difference between the two cases of entry vertex (i.e., boundary vertex or root). We refer to an exit vertex together with its matching entry vertex as an *entry-exit pair*. For easier description we refer to an entry vertex that has no matching exit vertex as part of an entry-exit pair where the entry is mapped to some special symbol, called the *null exit*. The following observation directly follows.

▶ **Observation 7.** *Let b be the number of (micro/mini) boundary vertices contained in a (micro/mini) piece P and T any palm tree. Then there are $O(b)$ entry-exit pairs associated with P.*

Let P be some micro piece stored as an index to the lookup table. We define a *partial DFS state of* P as a coloring of the vertices of P with the colors unvisited (white), currently visited (gray) and finished (black), and a constant number of (possibly empty) ordered sets of entry-exit pairs, ordered by, e.g., the pre-/and post-order numbering of their respective entry vertex, with ties being broken arbitrarily.

By the capabilities of the table lookup outlined in the previous section, an index of the lookup table is able to encode this information. The idea is that, when the DFS enters a micro piece P, categorized by some index x of the lookup table encoding P together with its partial DFS state, we obtain in $O(1)$ time an index y of the lookup table encoding P together with the next partial DFS state. We then swap out the index x with the index y. Note that this query depends on: the piece P, the coloring of the vertices of P, the ordered set of entry-exit pairs and the vertex we used to just enter the piece P (or the vertex we just backtracked to). This information is enough to obtain a new DFS state that correctly advances the DFS through P while considering the current partial DFS state. Since such an advancement results in the DFS traversing up to (or backtracking to) the next micro boundary vertex we obtain a new entry-exit pair in $O(1)$ time. There are multiple possible

viable partial DFS states that one could obtain by this operation, but we store one fixed state per entry of the lookup table, which is all that we require. See Figure 3 again for a visualization of how the DFS progresses through a piece. The change from one state of the figure to the next is done in $O(1)$ time for micro pieces.

A final note considers marking vertices as gray or black in micro pieces. When we visit a micro boundary vertex u, we must color u in all micro pieces $P_{i,j}$ that contain u (as micro label $u_{i,j}$). The translation mappings allow us to iterate over all micro pieces that contain $u_{i,j}$ in constant time per element output. For each such element output we must switch out the respective index of the micro pieces via the lookup table. This is done exactly twice (white to gray, and gray to black) per duplicate of a micro boundary vertices, and thus the time is linear in the number of total micro boundary vertices.

We now describe the data structures required for implementing the DFS. We begin with a description of how to implement an iterator over the neighborhood that can be paused and resumed. Note that while the nested division $\mathcal{D}$ provides neighborhood iteration (Lemma 6), it is not assumed that this iteration can be paused and continued at a later point in time without starting the process from the beginning. First, we describe this process for micro boundary vertices u_i in some piece P_i. For each such u_i store an array A containing the indices of each micro piece $P_{i,j}$ that contains u_i (as micro label $u_{i,j}$). An iterator for u_i now consists of an index j of A together with an index into the adjacency array of $u_{i,j}$ in $P_{i,j}$. Note that the array A contains many entries for each micro boundary vertex. By Lemma 5 this is asymptotically no problem, as the number of entries in all arrays A depends linearly on the number of micro boundary vertices. The arrays A can be built in sublinear time and space by iterating over all micro pieces, and inside each micro piece iterating over all micro-boundary vertices. We store the analogous data structure for all mini boundary vertices. We now must store the state of each iterator during the DFS. Construct an array storing for each boundary vertex the state of the iterator over u's neighborhood together with u's parent in the DFS tree and the color visited/unvisited. This uses $O(\log n)$ bits per mini boundary vertex. For each mini piece P_i we construct arrays storing the respective information of micro boundary vertices. As each vertex u (identified via mini label u_i) in P_i has a degree of $O(r)$ and must have its parent in P_i (identified by some mini label $v_i \in [r]$), this information can be stored with $O(\log r) = O(\log \log n)$ bits per micro boundary vertex. All other vertices are handled by the lookup table. All other data structures use $o(n)$ bits.

The DFS naturally computes the palm tree, but it remains to show how to store it such that afterward we can execute common queries such as iteration over all children (i.e., incident tree edges), or iteration over all back edges that start/end at a given vertex. We describe this for iteration over all children of a vertex, the other queries are realized in the exact same fashion. For micro non-boundary vertices these queries can directly be provided by the table lookup, or by recursive structure. We thus focus on the boundary vertices. For each mini boundary vertex u we maintain a list L containing the ids of all mini pieces P such that u has a child in P. Iteration over the children of u can then be expressed as an iteration over L, and for each $i \in L$ we output all children of u in P_i. Concretely this is done by outputting all children of u_i (the mini label of u in P_i) as their respective mini labels, and then translating them to their global labels. For micro boundary vertices, analogous structures are built. The space analysis is analogous to the space analysis of the previous paragraph, i.e., it uses $o(n)$ bits total. We summarize the results in the next lemma.

▶ **Lemma 8.** *Let G be a separable graph given as a nested division $\mathcal{D}$. In $O(n/\operatorname{poly}(\log \log n)) = o(n)$ time and $o(n)$ bits we can execute a DFS on $\mathcal{D}$ and store the resulting palm tree T such that the following queries are available for any vertex u, in constant time (per element output).*

- *Iteration over the children v of u in T, ordered by* $\mathrm{pre}(v)$.
- *Iteration over all back edges starting/ending at u in T.*
- *Output the parent u.*

3.2 Meta Information of a Depth-First Search

During the computation of a DFS it is common to store various values that relate to the traversal such as the pre-order $\mathrm{pre}(u)$ of a vertex u, or more complex values such as the *lowpoint* of a vertex u, defined as the lowest numbered (by pre-order numbering) vertex reachable via a path starting at u that consists of zero or more tree edges and at most one back edge. We can augment $\mathcal{D}$ with this information as follows. We start to describe the computation of the pre-ordering number $\mathrm{pre}(u)$. Post-order can be implemented in an analogous way. Assume that we have constructed the palm tree of Lemma 8.

Pre-order numbering. We begin with an observation regarding the pre-order numbering $\mathrm{pre}(u)$ of some non-boundary vertex u assigned to some piece P (the following description applies both for micro and mini pieces), and denote with k the number of vertices of P. Note that any DFS must enter P via an entry vertex v, which is either the root or a boundary vertex. If the DFS leaves P via an exit vertex, the DFS can discover new vertices in P only if it enters via another entry vertex, or it backtracks over the exit vertex. By this we can observe two scenarios: vertex u is visited either (1) within the next k steps after the DFS enters via an entry vertex v, or (2) within the next k steps after a DFS backtracks over some exit vertex v. For (1) we have $|\mathrm{pre}(u) - \mathrm{pre}(v)| \leq k$ for some entry vertex v, and for (2) we have $|\mathrm{pre}(u) - \mathrm{post}(v)| \leq k$ for some exit vertex v. What we have shown is that the pre-order numbering of a non-boundary vertex u can be computed as a small *local offset* $k' \leq k$ plus the pre- or post-order numbering of some entry or exit vertex v, called the *reference boundary vertex*. It is crucial that the local offset is bounded by the number of vertices in P.

For a mini boundary vertex u, we store $\mathrm{pre}(u)$ and $\mathrm{post}(u)$ explicitly in an array with $\log n$ bits each. For a mini non-boundary vertex u, and its mini label u_i in a piece P_i, we store the local offset k' together with its reference boundary v (stored as its mini label v_i). Storing these values explicitly would use $O(n \log\log n)$ bits total, and as such we again employ a recursive strategy. We apply the mentioned strategy only for vertices being mini non-boundary vertices, that are also micro boundary vertices and we store the local offset and reference boundary explicitly. For non-boundary vertices $v_{i,j}$ of a micro piece $P_{i,j}$ we use the table lookup to compute both the local offset together with the reference boundary on-the-fly. This general strategy can be used for any strongly local value, possibly in combination with some additional structures required for more involved queries. Simple values such as the post-order, depth of a vertex, or the number of descendants, can be handled exactly the same way as the pre-order. For two of the more involved values, lowpoint and lowest-common ancestor, we describe the details next.

Lowpoints. Recall that $\mathrm{lowpoint}(u)$ of a vertex u is defined as the lowest pre-order number $\mathrm{pre}(v)$ such that v is reachable via a path that traverses the DFS subtree rooted at u together with at most one back edge of the palm tree. The lowpoint is a crucial value for typical DFS applications such as identifying biconnected components. We show that $\mathrm{lowpoint}(u)$ is strongly local. Assume that u is a non-boundary vertex in some piece P. Denote with k the number of vertices in P. First note that any back edge (u, v) must have v in P. Thus, $\mathrm{lowpoint}(u)$ either is $\mathrm{pre}(u)$, depends on a back edge that lies within P, or it depends on the lowpoint of a boundary vertex that is a descendant of u. Thus, we require the lowpoint of a boundary vertex of P, together with a local offset $k' \leq k$. See Figure 4a.

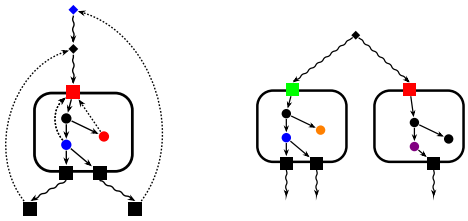

Figure 4 Lowpoints (a) The figure shows a piece and sketched parts of the DFS tree. Boundary vertices are drawn as squares, non-boundary vertices as circles, vertices of undefined type as diamonds. The lowpoint of the red non-boundary vertex is the red boundary vertex. The lowpoint of the blue non-boundary vertex is the blue diamond vertex. **LCAs (b)** The right figures shows two pieces. The LCA of the blue vertex and the violet vertex is the same as the LCA of the green and the red vertex. The LCA of the blue and the orange vertex is the green vertex. We can see that the LCA of two vertices in different pieces can be reduced to the LCA of two boundary vertices of the respective pieces.

The standard algorithm due to Hopcroft and Tarjan computes the lowpoints during a DFS traversal by updating it anytime the DFS backtracks to a vertex u and anytime a back edge (u, v) is discovered. For non-boundary vertices inside a micro piece, this update can be done via the table lookup using the augmentations provided by the lookup table for each boundary vertex of a micro piece $P_{i,j}$. Thus, anytime the DFS enters or leaves a micro piece $P_{i,j}$, we can update the current lowpoint values of each non-boundary vertex of $P_{i,j}$. As with the pre-order number, we store the required values for the boundary vertices explicitly in an array. The runtime of updating all values across the DFS is $O(n/\operatorname{poly}(\log\log n))$.

Lowest Common Ancestor. The operation $\operatorname{lca}(u, v)$ returns the root w of the smallest subtree T' of a tree T such that u and v are contained in T, i.e., it is the deepest vertex w in the tree such that u and v are descendants of w where deepest means it has the maximal distance from the root of T. We show that LCA queries for two vertices u, v can be reduced to strongly local values, one for each of the vertices. Let u, v be two non-boundary vertices of a piece P. If the LCA w of u, v lies within P, then the value is clearly local. If the LCA w does not lie within P, there must be a path from u to w and v to w such that each of these paths contains a boundary vertex u' and v', respectively. As there can be multiple such boundary vertices, let it be the first ones encountered by traversing the tree edges in reverse direction, i.e., the nearest such vertices. Then the LCA of u and v is equal to the LCA of u' and v'. Thus, for each pair of non-boundary vertices we only require to know the $\operatorname{lca}(u', v')$. The same is true, if u and v are non-boundary vertices in different pieces. Again, u' and v' are the nearest ancestors being boundary vertices in their respective pieces. Note that the choice of u' and v' is fixed for u and v, respectively, independent of the concrete query. See Figure 4b. To handle the LCA between mini boundary vertices, and for each mini piece, between micro boundary vertices, we use a known data structure to evaluate the LCA queries such as the data structure of Harel and Tarjan et al. [14]. For a tree with n' vertices, this data structure is initialized in $O(n')$ time and uses $O(n' \log n)$ bits such that LCA queries are available in linear time. We can not input our entire DFS tree to this data structure, and as such construct special smaller trees that capture the ancestry only between mini boundary vertices, and between micro boundary vertices for each mini piece. The time to construct the necessary structures is $O(n/\operatorname{poly}(\log\log n))$, and we use $o(n)$ bits.

We describe the concrete construction of the required smaller trees below. First observe the following. Let T' be some tree with vertex set V' and $S \subset V$ a set of *important vertices*, such that for pairs of vertices of S we want to know their LCAs. First, all leaves v in $V' \setminus S$ are irrelevant (can not be an LCA of a pair of vertices in S) and we recursively can remove them. Next consider vertices of $V' \setminus S$ of degree 2. These vertices are also irrelevant. We can replace such vertices v and its two edges $\{u, v\}\{v, w\}$ by a single edge $\{u, w\}$. This tree now contains the vertices of S and all vertices that are LCA's of pairs of vertices of S. Since we recursively removed all vertices of $V \setminus S$ with degree ≤ 2, the tree obtained has $O(|S|)$ vertices.

Now consider the DFS tree T and the set of mini boundary vertices. Taking the set of mini boundary vertices as the important vertices, we can see that we can shrink T such that it only contains $O(n/\operatorname{poly}(\log n))$ vertices without losing any information regarding the LCAs between boundary vertices. Analogously, for all micro boundary vertices inside a mini piece. The shrunken tree $T_{P_{i,j}}$ for a micro piece $P_{i,j}$ can be built via table lookup where we can support queries for every piece and every arbitrary set of important vertices S of the piece. Thus, we can obtain in $O(1)$ time a (set of) shrunken LCA tree(s) for piece $P_{i,j}$, which consists of the shrunken whole DFS tree with respect to the boundary vertices of the piece as important vertices. To build the shrunken tree for all mini boundary vertices, we iterate over all mini pieces $P_{i,j}$ and obtain shrunken $T_{P_{i,j}}$ with respect to the mini boundary vertices as important vertices, via table lookup. We so can build all required shrunken trees in $O(n/\operatorname{poly}(\log \log n))$ time using $o(n)$ bits.

▶ **Lemma 9.** *Let G be a separable graph given as a nested division $\mathcal{D}$ such that $\mathcal{D}$ encodes the palm tree of some DFS in $\mathcal{D}$. After $O(n/\operatorname{poly}(\log \log n))$ time preprocessing and using $o(n)$ bits we can provide the following queries.*

- children(u) *iteration over all children of u.*
- parent(u)*: output the parent of u.*
- pre(u)/post(u)*: return the pre-/post-order number of vertex u.*
- num-descendants(u)*: return the number of descendants of u.*
- depth(u)*: return the depth of u.*
- lowpoint(u)*: return the lowpoint of u.*
- lca(u, v)*: return the lowest-common ancestor of vertices u and v.*

Theorem 1 follows from first constructing the encoding of Lemma 6, and then constructing the palm tree of Lemma 8. Lemma 9 provides the queries. Next, we describe the details of constructing nested divisions efficiently.

3.3 Constructing Nested Divisions efficiently

For planar graphs we can construct the encoding of Lemma 6 efficiently, both in time and space. Kammer and Meintrup [21] presented a space-efficient algorithm for computing a $(\log n, r)$-relaxed divisions of minor closed graph classes that roughly works as follows. The input graph G is replaced with a minor F that contains $O(n/\log n)$ vertices such that each vertex of F represents $\Theta(\log n)$ vertices of G. Then, any algorithm $\mathcal{A}$ can be used for computing an $(1, r)$-relaxed division (a standard r-division) of F, which then can be turned into a $(\log n, r)$-relaxed division of G. As $\mathcal{A}$ is executed on a graph with $O(n/\log n)$ vertices, we achieve a speedup of a factor of $\Theta(\log n)$, and a space reduction by the same factor. Translation between the graph F and G uses linear time and bits. When recursively constructing divisions of mini pieces we can use standard algorithms as the graphs are small

enough. For planar graphs, linear time r-division algorithms [12, 23] exist, and thus the entire computation of the recursive division runs in linear time. For other graph classes the runtime depends on how efficiently separators can be found.

The final non-trivial part of constructing our encoding are related to the translation mappings between the different levels. The mapping is constructed using the powerful fully-indexable dictionary (FID) of Raman et al. [26] that, for a given universe $[\ell]$, uses $o(\ell)$ bits total when managing a set $S \subset [\ell]$ with $|S| = \Theta(\ell/\operatorname{poly}(\log \ell))$, which fits Blelloch and Farzan's use case. Effectively, this FID manages a compressed bit vector B of length ℓ where in B bits at index $i \in S$ are set to 1. Due to the use of Raman et al.'s FID their construction takes polynomial time $\omega(n)$ even for the simple case of encoding planar graphs and without regard to space-efficiency during the construction step. While Raman et al. mention their data structure can be constructed in expected linear time, a deterministic linear time construction is not known. This is due to the use of a certain hash function that is required. When succinctness is not required much simpler dictionaries suffices for the translation mappings. In this case, we effectively follow the same techniques outlined by Blelloch and Farzan, but use the indexable dictionary of Baumann and Hagerup, which can be constructed in $O(\ell/\log \ell)$ time and uses $\ell + o(\ell)$ bits for managing any set $S \subset [\ell]$:

▶ **Lemma 10** ([5]). *Let $B = (x_1, \ldots, x_\ell)$ be a bit string. In $O(\ell/\log \ell)$ time and $\ell + o(\ell)$ bits a data structure can be constructed such that afterward the following queries can be executed in constant time for $b \in \{0, 1\}$.*

- $\operatorname{rank}_b(i) = |\{j \in [i] | x_j = b\}|$
- $\operatorname{select}_b(i) = \min\{j \in [n] : \operatorname{rank}_B(j) = b\}$

Using the FID of Lemma 10 we are able to construct a compact variant of the encoding of Lemma 6 in $O(n)$ time and bits. If one is fine with expected linear time, the FID of Raman et al. [26] can be used and the encoding remains succinct. The functionalities of the different variants are the same.

▶ **Corollary 11** (Compact Nested Division). *For planar graphs, a compact (succinct) variant of the encoding of Lemma 6 can be constructed with $O(n)$ bits and $O(n)$ time (expected time).*

Combining Corollary 11 with Theorem 1 we are able to show Corollary 2. We give a more general version for arbitrary separable graphs next. In the following we denote with $f_{\mathcal{G}}(n)$ the runtime of an algorithm for graphs with n vertices of some separable graph class $\mathcal{G}$ that recursively computes a balanced separator until the graphs are small enough (of poly-logarithmic size), and with $f'_{\mathcal{G}}(n)$ the number of bits required by such an algorithm.

▶ **Corollary 12.** *Let G be a graph of a separable graph class $\mathcal{G}$. There exists a compact (succinct) encoding of G that can be constructed in $f_{\mathcal{G}}(n) + O(n)$ time (expected time) and $f'_{\mathcal{G}}(n) + O(n)$ bits that provides the same functionalities as the encoding of Theorem 1.*

References

1 Hüseyin Acan, Sankardeep Chakraborty, Seungbum Jo, and Srinivasa Rao Satti. Succinct data structures for families of interval graphs. In *16th International Symposium on Algorithms and Data Structures (WADS 2019)*, pages 1–13. Springer, 2019. doi:10.1007/978-3-030-24766-9_1.

2 Luca Castelli Aleardi, Olivier Devillers, and Gilles Schaeffer. Optimal succinct representations of planar maps. In *Proceedings of the Twenty-Second Annual Symposium on Computational Geometry (SCG 2006)*, pages 309–318. ACM, 2006. doi:10.1145/1137856.1137902.

3 Tetsuo Asano, Taisuke Izumi, Masashi Kiyomi, Matsuo Konagaya, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, Jun Tarui, and Ryuhei Uehara. Depth-first search using $O(n)$ bits. In Hee-Kap Ahn and Chan-Su Shin, editors, *25h International Symposium on Algorithms and Computation (ISAAC 2014)*, pages 553–564. Springer, 2014. doi:10.1007/978-3-319-13075-0_44.

4 Niranka Banerjee, Sankardeep Chakraborty, Venkatesh Raman, and Srinivasa Rao Satti. Space efficient linear time algorithms for bfs, DFS and applications. *Theory Comput. Syst.*, 62(8):1736–1762, 2018. doi:10.1007/S00224-017-9841-2.

5 Tim Baumann and Torben Hagerup. Rank-select indices without tears. In *16th International Symposium on Algorithms and Data Structures (WADS 2019)*, pages 85–98. Springer, 2019. doi:10.1007/978-3-030-24766-9_7.

6 Guy E. Blelloch and Arash Farzan. Succinct representations of separable graphs. In *21st Annual Symposium on Combinatorial Pattern Matching (CPM 2010)*, pages 138–150. Springer, 2010. doi:10.1007/978-3-642-13509-5_13.

7 Sankardeep Chakraborty, Venkatesh Raman, and Srinivasa Rao Satti. Biconnectivity, Chain Decomposition and st-Numbering Using O(n) Bits. In *27th International Symposium on Algorithms and Computation (ISAAC 2016)*, volume 64 of *LIPIcs*, pages 22:1–22:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ISAAC.2016.22.

8 Yi-Ting Chiang, Ching-Chi Lin, and Hsueh-I Lu. Orderly spanning trees with applications to graph encoding and graph drawing. In *12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 506–515. SIAM, 2001. URL: http://dl.acm.org/citation.cfm?id=365411.365518.

9 Amr Elmasry, Torben Hagerup, and Frank Kammer. Space-efficient Basic Graph Algorithms. In *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, volume 30 of *LIPIcs*, pages 288–301. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.STACS.2015.288.

10 Greg N. Federickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987. doi:10.1137/0216064.

11 Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Trans. Algorithms*, 2(4):510–534, 2006. doi:10.1145/1198513.1198516.

12 M.T. Goodrich. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences*, 51(3):374–389, 1995. doi:10.1006/jcss.1995.1076.

13 Torben Hagerup. Space-efficient DFS and applications to connectivity problems: Simpler, leaner, faster. *Algorithmica*, 82(4):1033–1056, 2020. doi:10.1007/S00453-019-00629-X.

14 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. doi:10.1137/0213024.

15 Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Lacki, and Eva Rotenberg. Decremental SPQR-trees for Planar Graphs. In *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *LIPIcs*, pages 46:1–46:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ESA.2018.46.

16 Jacob Holm and Eva Rotenberg. Good r-divisions imply optimal amortized decremental biconnectivity. *Theory Comput. Syst.*, 68(4):1014–1048, 2024. doi:10.1007/S00224-024-10181-Z.

17 John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973. doi:10.1145/362248.362272.

18 Taisuke Izumi and Yota Otachi. Sublinear-Space Lexicographic Depth-First Search for Bounded Treewidth Graphs and Planar Graphs. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, volume 168 of *LIPIcs*, pages 67:1–67:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2020.67.

19 G. Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science (FOCS 1989)*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.

20 Frank Kammer, Dieter Kratsch, and Moritz Laudahn. Space-efficient biconnected components and recognition of outerplanar graphs. *Algorithmica*, 81(3):1180–1204, 2019. doi:10.1007/S00453-018-0464-Z.

21 Frank Kammer and Johannes Meintrup. Space-Efficient Graph Coarsening with Applications to Succinct Planar Encodings. In *33rd International Symposium on Algorithms and Computation (ISAAC 2022)*, volume 248 of *LIPIcs*, pages 62:1–62:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ISAAC.2022.62.

22 Kenneth Keeler and Jeffery Westbrook. Short encodings of planar graphs and maps. *Discrete Appl. Math.*, 58(3):239–252, 1995. doi:10.1016/0166-218X(93)E0150-W.

23 Philip N. Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *45th Annual ACM Symposium on Theory of Computing (STOC 2013)*, pages 505–514. Association for Computing Machinery, 2013. doi:10.1145/2488608.2488672.

24 Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979. doi:10.1137/0136016.

25 J. Ian Munro, Patrick K. Nicholson, Louisa Seelbach Benkner, and Sebastian Wild. Hypersuccinct Trees - New Universal Tree Source Codes for Optimal Compressed Tree Data Structures and Range Minima. In *29th Annual European Symposium on Algorithms (ESA 2021)*, volume 204 of *LIPIcs*, pages 70:1–70:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.70.

26 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–es, 2007. doi:10.1145/1290672.1290680.

27 Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4), September 2008. doi:10.1145/1391289.1391291.

28 Jens M. Schmidt. A simple test on 2-vertex- and 2-edge-connectivity. *Information Processing Letters*, 113(7):241–244, 2013. doi:10.1016/j.ipl.2013.01.016.