Succinct Representation of Labeled Graphs

Jérémy Barbay[1], Luca Castelli Aleardi[2], Meng He[1], and J. Ian Munro[1]

[1] Cheriton School of Computer Science, University of Waterloo, Canada, {jbarbay, mhe, imunro}@uwaterloo.ca
[2] Laboratoire d'Informatique, Ecole Polytechnique and University of Marne-la-Vallée, France, amturing@lix.polytechnique.fr

Technical Report CS-2007-11
Cheriton School of Computer Science
University of Waterloo

Abstract. In many applications, properties of an object being modeled are stored as labels on vertices or edges of a graph. In this paper, we consider succinct representation of labeled graphs. Our main results are the succinct representations of labeled and multi-labeled graphs (we consider vertex labeled planar triangulations, as well as edge labeled planar graphs and the more general k-book embedded graphs) to support various label queries efficiently. The additional space cost to store the labels is essentially the information-theoretic minimum. As far as we know, our representations are the first succinct representations of labeled graphs.
We also have two preliminary results to achieve the main results. First, we design a succinct representation of unlabeled planar triangulations to support the rank/select of edges in ccw (counter clockwise) order in addition to the other operations supported in previous work. Second, we design a succinct representation for a k-book graph when k is large to support various navigational operations more efficiently. In particular, we can test the adjacency of two vertices in $O(\lg k \lg\lg k)$ time, while previous work uses $O(k)$ time [9, 15].

1 Introduction

Graphs are fundamental combinatorial objects in mathematics and in computer science. They are widely used to represent various types of data, such as the link structure of the web, geographic maps, and surface meshes in computer graphics. As modern applications often process large graphs, the problem of designing space-efficient data structures to represent graphs has attracted much attention. Researchers have designed succinct graphs to address this problem by applying the idea of *succinct data structures*, i.e. data structures that occupy space close to the information-theoretic lower bound to represent them, while supporting efficient navigational operations [4, 5, 6, 7, 12, 15].

Previous work focused on succinct graph representations which support efficiently testing the adjacency between two vertices and listing the edges incident to a vertex [4, 5, 15]. However in many practical applications this connectivity information is associated with labels on the edges or vertices of the graph, and the space required to encode those labels largely dominates the space used to encode the connectivity information, even when the encoding of the labels is compressed [11]. For example, when surface meshes are associated with properties such as color and texture information, more bits per vertex are required to encode those labels than to encode the graph itself. We address this problem by designing succinct representations of *labeled graphs*, where labels from an alphabet $[\sigma]$ [3] are associated with edges or vertices, which support efficiently label-based connectivity queries, such as navigating

[3] We use $[\sigma]$ to denote the set $\{1, 2, \ldots, \sigma\}$ of references to arbitrary labels.

among the subset of neighbors of (resp. edges connected to) a vertex v which are associated with a label α. Our results are on the word RAM model with word size $\Theta(\lg n)$ bits[4]. We assume that all the graphs are simple graphs for simplicity, though most of our techniques can be extended to support multigraphs.

The rest of the paper is organized as follows. We describe previous work that we either use or improve upon in Section 2. We present a succinct index for triangulated planar graphs with labels associated with their vertices in Section 3. To achieve this result, we describe a succinct representation of unlabeled planar triangulations which supports the rank/select of edges in ccw (counter clockwise) order in addition to the other operations supported in previous work [4, 5, 6, 7]. We present a succinct encoding for k-book graphs and with labels associated with their edges in Section 4. To achieve this result, we design a succinct representation for a k-book graph when k is large which supports various navigational operations more efficiently. For example, we can test the adjacency of two vertices in $O(\lg k \lg\lg k)$ time, while previous work uses $O(k)$ time [9, 15]. We conclude with a discussion of our results in Section 5.

2 Preliminaries

2.1 Related Work

Here we briefly review related work on succinct unlabeled graphs. As most graphs in practice have particular combinatorial properties, researchers usually exploit these properties to design succinct representations.

Jacobson [12] first proposed the problem of representing unlabeled graphs succinctly. His approach is based on the concept of *book embedding* by Bernhart and Kainen [3]. He showed how to represent a k-book embedded graph on the bit probe model using $O(kn)$ bits to support adjacency test in $O(\lg n)$ time, and the listing of all neighbors of a given vertex x in $O(d(x)\lg n + k)$ time, where $d(x)$ is the degree of x. Munro and Raman [15] improved his results on the word RAM model by showing how to represent a graph using $2kn + 2m + o(nk + m)$ bits to support adjacency test and the computation of the degree of a vertex in $O(k)$ time, and the listing of all the neighbors of a given vertex x in $O(d(x) + k)$ time. Gavoille and Hanusse [9] proposed a different tradeoff. They proposed an encoding in $2(m + i)\lg k + 4(m + i) + o(km)$ bits, where i is the number of isolated vertices, to support adjacency test in $O(k)$ time. A very common type of graphs are planar graphs, and any planar graph can be embedded in at most 4 pages [18]. Thus the above results can be applied directly to planar graphs. In particular, a planar graph can be represented using $8n + 2m + o(n)$ bits to support adjacency test and the computation of the degree of a vertex in constant time, and the listing of all the neighbors of a given vertex x in $O(d(x))$ time [15].

A different line of research is based on the canonical ordering of planar graphs. Chuang *et al.* [7] designed a succinct representation of planar graphs of n vertices and m edges in $2m + (5 + \epsilon)n + o(m + n)$ bits, for any constant $\epsilon > 0$, to support the operations on planar graphs in asymptotically the same amount of time as the approach described in the previous paragraph. Chiang *et al.* [6] have further reduced the space cost to $2m + 3n + o(m + n)$ bits. When a planar graph is triangulated, Chuang *et al.* [7] showed how to represent it using $2m + 2n + o(m + n)$ bits.

[4] We use $\log_2 x$ to denote the logarithmic base 2 and $\lg x$ to denote $\lceil \lg x \rceil$. Occasionally this matters.

Based on a partition algorithm, Castelli Aleardi *et al.* [4] proposed a succinct representation of planar triangulations with a boundary. Their data structure uses 2.175 bits per triangle to support various operations efficiently. Castelli Aleardi *et al.* [5] further combined this approach with some recent optimal encodings of planar maps to design succinct representations of 3-connected planar graphs and triangulations using 2 bits per edge and 1.62 bits per triangle respectively, which asymptotically match the respective entropy of these two types of graphs.

2.2 Multiple Parentheses

Chuang *et al.* [7] proposed to succinctly represent *multiple parentheses*, which is a string of $O(1)$ types of parentheses that may be unbalanced. Thus a multiple parenthesis sequence of p types of parentheses is a sequence over the alphabet $\{(_1, (_2, ..., (_p,)_1,)_2, ...,)_p\}$. We call $(_i$ and $)_i$ *type-i open parenthesis* and *type-i closing parenthesis*, respectively. Chuang *et al.* [7] proved the following lemma:

Lemma 1 ([7]). *Let S be a string consisting of $O(1)$ types of parentheses. Then it is possible to construct in $O(|S|)$ time a succinct representation of S using $o(|S|)$ auxiliary bits, which supports the following queries in $O(1)$ time:*

- $\texttt{m\_rank}(S, i, \alpha)$*: the number of parentheses α in $S[1..i]$;*
- $\texttt{m\_select}(S, i, \alpha)$*: the position of the i^{th} parenthesis α;*
- $\texttt{m\_first}_\alpha(S, i)$*: the position of the first occurrence of parenthesis α after $S[i]$;*
- $\texttt{m\_last}_\alpha(S, i)$*: the position of the last occurrence of parenthesis α before $S[i]$;*
- $\texttt{m\_match}(S, i)$*: the position of the parenthesis matching $S[i]$;*
- $\texttt{m\_enclose}_k(S, i_1, i_2)$*: the position of the closest matching parenthesis pair of type k which encloses $S[i_1]$ and $S[i_2]$.*

We propose an encoding for the case when the number of types of parentheses is non-constant in Theorem 3.

2.3 Succinct Indexes for Strings and Binary Relations

Barbay et al. [1] showed how to achieve data abstraction in the design of succinct data structures by designing *succinct indexes*. Given an abstract data type, or ADT, to access the given data, the goal is to design auxiliary data structures (i.e. succinct indexes) that occupy asymptotically less space than the information-theoretic lower bound on the space required to encode the given data, and support an extended set of operations using the basic operators defined in the ADT. They considered a string S of length n over an alphabet of arbitrary size σ, and the operations include: $\texttt{string\_rank}(\alpha, x)$, the number of occurrences of α in $S[1 \dots x]$; $\texttt{string\_select}(\alpha, r)$, the position of the r^{th} occurrence of α in the string; and $\texttt{string\_access}(x)$, the character at position x in the string. They defined the interface of the ADT of a string through the operator `string_access` and achieved:

Lemma 2 (Theorem 3.1 in [1]). *Given support for* `string_access` *in $f(n, \sigma)$ time on a string $S \in [\sigma]^n$, there is a succinct index using $n \cdot o(\lg \sigma)$ bits that supports* `string_rank` *in $O((\lg\lg\lg\sigma)^2(f(n,\sigma) + \lg\lg\sigma))$ time, and* `string_select` *in $O(\lg\lg\lg\sigma(f(n,\sigma) + \lg\lg\sigma))$ time.*

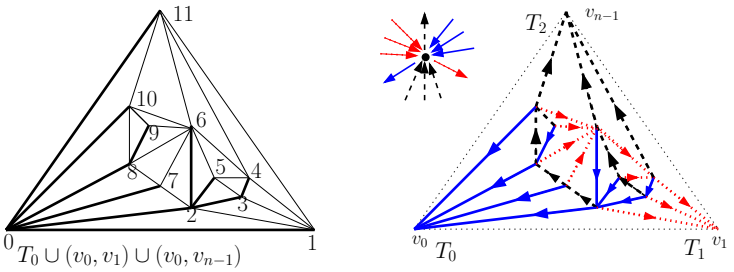

Fig. 1. A triangulated planar graph having 12 vertices with its canonical spanning tree $\overline{T}_0$ (on the left). On the right it is shown the triangulation induced with a realizer, as well as the local condition describing incident edges around each internal vertex.

Barbay *et al.* [1] extended the problem to the design of succinct indexes for sequences of n objects where each object can be associated with a subset of labels from $[\sigma]$, this association being defined by a binary relation of t pairs from $[n]\times[\sigma]$. The operations include: `object_access`(x, i), the i^{th} label associated with x in lexicographic order, and $+\infty$ if no such label exists. `label_rank`(α, x), the number of objects labeled α up to (and including) x; `label_select`(α, r), the position of the r^{th} object labeled α; and `label_access`(x, α), whether object x is associated with label α. They defined the interface of the ADT of a binary relation through the operator `object_access` and achieved:

Lemma 3 (Theorem 3.2 in [1]). *Given support for* `object_access` *in* $f(n, \sigma, t)$ *time on a binary relation formed by* t *pairs from an object set* $[n]$ *and a label set* $[\sigma]$*, there is a succinct index using* $t \cdot o(\lg \sigma)$ *bits that supports* `label_rank` *and* `label_access` *in* $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ *time, and* `label_select` *in* $O(\lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ *time.*

2.4 Realizers and Planar Triangulations

First, we give the definition of the realizers of planar triangulations:

Definition 1 ([17]). *A* ***realizer*** *of a planar triangulation* $\mathcal{T}$ *is a partition of the set of the internal edges into three sets* T_0*,* T_1 *and* T_2 *of directed edges, such that for each internal vertex* v *the following conditions are true:*

- *vertex* v *has exactly one outgoing edge in each of the three sets* T_0*,* T_1 *and* T_2*;*
- ***local condition:*** *the edges incident to* v *in ccw order are: one outgoing edge in* T_0*, zero or more incoming edges in* T_2*, one outgoing edge in* T_1*, zero or more incoming edges in* T_0*, one outgoing edge in* T_2*, and finally zero or more incoming edges in* T_1 *(see Figure 1).*

A fundamental property of the realizers of a planar triangulation is:

Lemma 4 ([17]). *Consider a planar triangulation* $\mathcal{T}$ *of* n *vertices, with exterior face* (v_0, v_1, v_{n-1})*. Then* $\mathcal{T}$ *always admits a realizer* $R = (T_0, T_1, T_2)$ *and each set of (undirected) edges in* T_i *is a spanning tree of all internal vertices. More precisely we have:*

- T_0 *is a spanning tree of* $\mathcal{T} \setminus \{v_1, v_{n-1}\}$*;*
- T_1 *is a spanning tree of* $\mathcal{T} \setminus \{v_0, v_{n-1}\}$*;*
- T_2 *is a spanning tree of* $\mathcal{T} \setminus \{v_0, v_1\}$*;*

2.5 The Book Embedding of a Graph

Bernhart and Kainen [3] proposed to embed a graph in a book. A *k-book embedding* of a graph is a topological embedding of it in a book of k pages that specifies the ordering of the vertices along the spine, and carries each edge into the interior of one page. Thus, a graph with one page is an *outerplanar graph*. The *pagenumber* (written as a single word) or *book thickness* [3] of a graph is the minimum number of pages that the graph can be embedded in. A very useful fact is that any planar graph can be embedded in at most 4 pages [18].

3 Vertex Labeled Planar Triangulations

3.1 Three New Traversal Orders on a Planar Triangulation

We first define three new traversal orders on planar triangulations based on realizers. Let $\mathcal{T}$ be a planar triangulation with exterior face (v_0, v_1, v_{n-1}). We denote its realizer by (T_0, T_1, T_2) following Definition 1. By Lemma 4, T_0, T_1 and T_2 are three spanning trees of the internal nodes of $\mathcal{T}$, rooted at v_0, v_1 and v_{n-1}, respectively. We add the edges (v_0, v_1) and (v_0, v_{n-1}) to T_0, and call the resulting tree, $\overline{T}_0$, the *canonical spanning tree* of $\mathcal{T}$ [7]. In this section, we denote each vertex by its number in *canonical ordering*, which corresponds to the ccw preorder number in $\overline{T}_0$.

Definition 2. ***The zeroth order*** π_0 *is defined on all the vertices of* $\mathcal{T}$ *and is simply given by the preorder traversal of* $\overline{T_0}$ *starting at* v_0 *in* counter clockwise order *(ccw order).*

The first order π_1 *is defined on the vertices of* $\mathcal{T} \setminus v_0$ *and corresponds to a traversal of the edges of* T_1 *as follows. Perform a preorder traversal of the contour of* $\overline{T_0}$ *in a ccw manner. During this traversal, when visiting a vertex* v*, we enumerate consecutively its incident edges* $(v, u_1), \ldots, (v, u_i)$ *in* T_1*, where* v *appears before* u_i *in* π_0*. The traversal of the edges of* T_1 *naturally induces an order on the nodes of* T_1*: each node (different from* v_1*) is uniquely associated with its parent edge in* T_1*.*

The second order π_2 *is defined on the vertices of* $\mathcal{T} \setminus \{v_0, v_1\}$ *and can be computed in a similar manner by performing a preorder traversal of* T_0 *in* clockwise order *(cw order). When visiting in cw order the contour of* $\overline{T_0}$*, the edges in* T_2 *incident to a node* v *are listed consecutively to induce an order on the vertices of* T_2*.*

Note that the orders π_1 and π_2 do not correspond to any known traversal order on the trees T_1 and T_2, as they are dependent on $\overline{T_0}$ through π_0. Another important fact about π_1 and π_2 (but not π_0) is that, as DFUDS, they list consecutively the children of each node x in the corresponding tree, i.e., for any node x, its children in T_1 (resp. in T_2) have consecutive numbers in π_1 (resp. in π_2). Figure 2 illustrates the orders π_0, π_1 and π_2 defined above.

3.2 Planar Triangulations

We consider the following operations on unlabeled planar triangulations:
- `adjacency`(x, y), whether vertices x and y are adjacent;
- `degree`(x), the degree of vertex x;
- `select_neighbor_ccw`(x, y, r): the r^{th} neighbor of vertex x starting from vertex y in ccw order if x and y are adjacent, and ∞ otherwise;

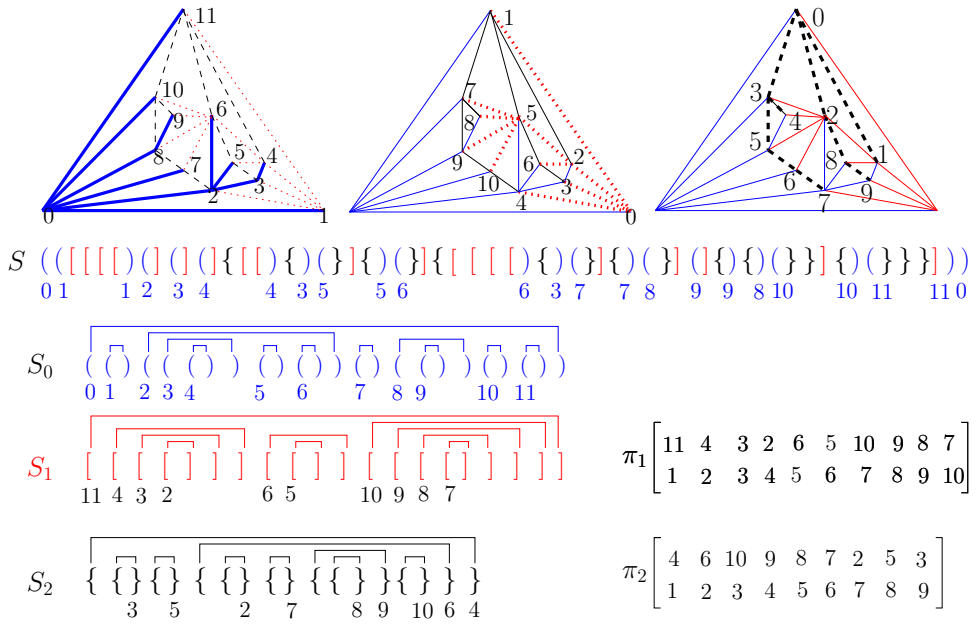

Fig. 2. A planar triangulation induced with one realizer: the three orders π_0, π_1 and π_2 on the set of its vertices are also shown.

– `rank_neighbor_ccw`(x, y, z): the number of neighbors of vertex x between (and including) the vertices y and z in ccw order if y and z are both neighbors of x, and ∞ otherwise.

To represent a planar triangulation $\mathcal{T}$, we compute a realizer (T_0, T_1, T_2) of $\mathcal{T}$ following Definition 1. We then encode the three trees T_0, T_1 and T_2 using a multiple parenthesis sequence S of length $2m$ consisting of three types of parenthesis. S is obtained by performing a preorder traversal of the canonical spanning tree $\overline{T}_0 = T_0 \cup (v_0, v_1) \cup (v_0, v_{n-1})$ and using different types of parentheses to describe the edges of $\overline{T}_0$ and edges in $\mathcal{T} \setminus \overline{T}_0$. We use parentheses of the first type, namely $'('$ and $')'$, to encode the tree $\overline{T}_0$, while other types of parentheses, $'['$, $']'$, $'\{'$, $'\}'$ to encode the edges of T_1 and T_2.

Let $v_0, \ldots, v_{n-1}$ be the ccw preorder of the vertices of $\overline{T}_0$. Then the string S_0 is simply the balanced parenthesis encoding of the tree $\overline{T}_0$ [15]: S_0 can be obtained by performing a ccw preorder traversal of the contour of $\overline{T}_0$, writing down an open parenthesis when an edge of $\overline{T}_0$ is traversed the first time, and a closing parenthesis when it is visited the second time. During the traversal of $\overline{T}_0$, we insert in S a pair of parentheses $'['$ and $']'$ for each edge of T_1, and a pair of parentheses $'\{'$ and $'\}'$ for each edge in T_2. More precisely, when visiting in ccw order the edges incident to a vertex v_i, we:

– Insert a $'['$ for each edge (v_i, v_j) in T_1, where $i < j$, before the parenthesis $')'$ corresponding to v_i;
– Insert a $']'$ for each edge (v_i, v_j) in T_1, where $i < j$, after the parenthesis $'('$ corresponding to v_j;
– Insert a $'\}'$ for each edge (v_i, v_j) in T_2, where $i > j$, after the parenthesis $'('$ corresponding to v_i;
– Insert a $'\{'$ for each edge (v_i, v_j) in T_2, where $i > j$, before the parenthesis $')'$ corresponding to v_j.

The relative order of the parentheses '[', ']', '}' and '{' inserted between two consecutive parentheses of the other type, i.e. '(' or ')', do not matter. Thus string S is of length $2m$, consisting of three types of parenthesis. It is easy to observe that substrings S_1 and S_2 are balanced parenthesis sequences of length $2(n-1)$ and $2(n-2)$, respectively.

We first observe some basic properties of string S. Recall that a node v_i can be referred to by its preorder number in T_0, and by the position of the matching parenthesis pair $(_i$ and $)_i$ (let p_i and q_i denote their positions in S). Let be p_f (or q_l) be the position of the open (or closing) parenthesis in S corresponding to the first (or last) child of node v_i in T_0.

Property 1. The following basic facts hold:
- Two nodes v_i and v_j are adjacent if and only if there is one common incident edge (v_i, v_j) in exactly one of the trees T_0, T_1 or T_2;
- $p_i < p_f < q_l < q_i$;
- The number of edges incident to v_i and not belonging to the tree T_0 is $(p_f - p_i) + (q_i - q_l)$;
- If v_i is not a leaf in $\overline{T_0}$, between the occurrences of the '(' that correspond to the vertices v_i and v_{i+1} (note that the '(' corresponding to v_{i+1} is at position p_f), there is exactly one ']'. Similarly, there is exactly one '{' between the ')' that correspond to the vertices v_i and the ')' at position q_l.

Observe that S_0 is the balanced parenthesis encoding of the tree $\overline{T_0}$ [15], so that if we store S_0 and construct the auxiliary data structures for S_0 as in [15], we can support a set of navigational operators on $\overline{T_0}$. S can be represented using Lemma 1 in $2m \log_2 6 + o(m)$ bits, which is essentially 5.17 bits per edge. However, this encoding does not support the computation of an arbitrary word in S_0, so that we cannot navigate the tree $\overline{T_0}$ without storing S_0 explicitly, which will cost essentially 2 additional bits per node. To reduce this space redundancy, we have:

Lemma 5. *The string S can be stored in $2m \log_2 6 + o(m)$ bits to support in $O(1)$ time the operators listed in Lemma 1, as well as the computation of an arbitrary word, or $\Theta(n)$ bits of the balanced parenthesis sequence of $\overline{T_0}$.*

Proof. We construct a conceptual bit vector B_1 of $2m$ bits, so that $B_1[i] = 1$ iff $S[i] =$ '(' or $S[i] =$ ')'. We construct another conceptual bit vector B_2 of $2m - 2n$ bits for the 0s in B_2 (recall that there are $2n$ parentheses in $\overline{T_0}$), so that $B_2[i] = 1$ iff the parenthesis corresponds to the i^{th} 0 in B_1 is either '[' or ']'. We store B_1 and B_2 using the fully indexable dictionary encoding by Raman *et al.* [16] to support rank/select operations on them. The space cost of storing B_1 and B_2 is thus $\lg \binom{2m}{n} + o(m) + \lg \binom{2m-2n}{n} + o(m)$. As $m = 3n - 3$, the above space cost is approximately $2m \log_2 3 + o(m)$.

In addition, we store S_0, S_1 and S_2 using the encoding of balanced parentheses by Munro and Raman [15] to support the rank/select of each parenthesis among the parentheses of the same type, locating the match parenthesis of a given parenthesis, and the computation of the closing enclosing pair of a given parenthesis pair. The space cost of storing these three sequences is $2n + o(n) + 2(n-1) + o(n) + 2(n-2) + o(n) = 2m + o(m)$ bits. Thus the total space cost is $2m \log_2 6 + o(m)$ bits.

As B_1 and B_2 can be used to compute the rank/select operations over S if we treat each type of (open and closing) parentheses as the same character, and S_0, S_1 and S_2 can be used

to support operations on the parentheses of the same type, the operations listed in Lemma 1 can be easily supported in constant time. As we store S_0 explicitly in our representation, we can trivially support the computation of an arbitrary word of S_0. □

The same approach can be directly applied to a multiple parenthesis sequence of $O(1)$ types of parentheses:

Corollary 1. *Consider a multiple parenthesis sequence M of $2n$ parenthesis of p types, where $p = O(1)$. M can be stored using $2n \lg(2p) + o(n)$ bits to support in $O(1)$ time the operators listed in Lemma 1, as well as the computation of an arbitrary word, or $\Theta(n)$ bits of the balanced parenthesis sequence of the parentheses of a given type in M.*

The following theorem shows how to efficiently support the navigational operations in a triangulation and how to compute the number of an internal node in π_1 or π_2 given its number in π_0 (and vice-versa).

Theorem 1. *A planar triangulation $\mathcal{T}$ of n vertices and m edges can be represented using $2m \log_2 6 + o(m)$ bits to support* `adjacency`, `degree`, `select_neighbor_ccw`, `rank_neighbor_ccw` *and the following operation in $O(1)$ time (for $j \in \{1, 2\}$):*
- *$\Pi_j(i)$: given the number of a node v_i in π_0 it returns the number of v_i in π_j;*
- *$\Pi_j^{-1}(i)$: given the number of a node v_i in π_j it returns its rank in π_0.*

Proof. We construct the string S for $\mathcal{T}$ as shown in this section, and store it using $2m \log_2 6 + o(m)$ bits by Lemma 5. Recall that S_0 is the balanced parenthesis encoding of $\overline{T_0}$, and that we can compute an arbitrary word of S_0 from S. Thus we can construct additional auxiliary structures using $o(n) = o(m)$ bits [14, 15] to support the navigational operations on $\overline{T_0}$. As each vertex is denoted by its number in canonical ordering, we have that vertex x corresponds to the x^{th} open parenthesis in S_0. We now show that these structures are sufficient to support the navigational operations on $\mathcal{T}$.

To compute `adjacency`(x, y), recall that x and y are adjacent iff one is the parent of the other in one of the trees $\overline{T_0}$, T_1 and T_2. As S_0 encodes the balanced parenthesis sequence of $\overline{T_0}$, we can trivially check whether x (or y) is the parent of y (or x) using existing algorithms on S_0 [15]. To test adjacency in T_1, we recall that x is the parent of y iff the (only) incoming edge of y, denoted by a $']'$, is an outgoing edge of x, denoted by a $'['$. It then suffices to retrieve the first $']'$ after the y^{th} $'('$ in S, given by $\texttt{m\_first}_{']'}(S, \texttt{m\_select}(S, y, '('))$, and compute the index, i, of its matching closing parenthesis, $'['$, in S. We then check whether the nearest succeeding closing parenthesis $')'$ of the $'['$ retrieved, located using $\texttt{m\_first}_{')'}(S, i)$, is the x^{th} closing parenthesis in S. If it is, then x is the parent of y. We can use a similar approach to test the adjacency in T_2.

To compute `degree`(x), let d_0, d_1 and d_2 be the degrees of x in the trees $\overline{T_0}$, T_1 and T_2 (we denote the degree of a node in a tree as the number of nodes adjacent to it), respectively, so that the sum of these three values is the answer. To compute d_0, we use S_0 and the algorithm to compute the degree of a node in an ordinal tree using its balanced parenthesis representation by Chuang *et al.* [7]. To compute $d_1 + d_2$, if x has children in $\overline{T_0}$, we first compute the indices, i_1 and i_2, of the x^{th} and the $x + 1^{\text{th}}$ $'('$ in S, and the indices, j_1 and j_2, of the $(n - x)^{\text{th}}$ and the $(n - x + 1)^{\text{th}}$ $')'$ in S in constant time. By the third item of Property 1, we have the property $d_1 + d_2 = (i_2 - i_1) + (j_2 - j_1)$. The case when x is a leaf in $\overline{T_0}$ can be handled similarly.

To support `select_neighbor_ccw` and `rank_neighbor_ccw`, we make use of the local condition of realizers in Definition 1. The local condition tell us that, given a vertex x, its neighbors, when listed in ccw order, form the following six types of vertices: x's parent in $\overline{T_0}$, x's children in T_2, x's parent in T_1, x's children in $\overline{T_0}$, x's parent in T_2, and x's children in T_1. The i^{th} child of x in ccw order in $\overline{T_0}$ can be computed in constant time, and the number of siblings before a given child of x in ccw order can also be computed in constant time using the algorithms of Lu and Yeh [14]. The children of x in T_1 corresponds to the parentheses $'['$ between the $(n-x)^{\text{th}}$ and the $(n-x+1)^{\text{th}}$ $')'$ in S, and because of the construction of S, if u and v are both children of x, and u occurs before v in π_1, then u is also before v in ccw order among the children of x. The children of x in T_2 have a similar property. Thus the operators supported on S allow us to perform rank/select operations on the children of x in T_1 and T_2 in ccw order. As we can also compute the number of each type of neighbors of x in constant time, this allows us to support `select_neighbor_ccw` and `rank_neighbor_ccw` in constant time.

To compute $\Pi_1(i)$, we first locate the position, j, of the i^{th} $'('$ in S, which is $\texttt{m\_select}(S, i, '(')$. We then locate the position, k, of the first $']'$ after position j, which is $\texttt{m\_first}_{\eta'}(S, j)$. After that, we locate the matching parenthesis of $S[j]$ using $\texttt{m\_match}(S, j)$ (p denotes the result). $S[p]$ is the parenthesis $'['$ that corresponds to the parent of u_i in T_1, and by the construction algorithm of S, then rank of $S[p]$ is the answer, which is $\texttt{m\_rank}(S, p, '[')$. The computation of Π_1^{-1} is exactly the inverse of the above process. Π_2 and Π_2^{-1} can be supported similarly. □

3.3 Vertex Labeled Planar Triangulations

In addition to unlabeled operators, we present a set of operators that allow efficient navigation on a labeled graph (these are natural extensions to navigational operators on labeled trees):

- `lab_degree`(α, x), the number of neighbors of vertex x in G associated with label α;
- `lab_select_ccw`(α, x, y, r), the r^{th} vertex labeled α among neighbors of vertex x after vertex y in ccw order, if y is a neighbor of x, and ∞ otherwise;
- `lab_rank_ccw`(α, x, y, z): the number of the neighbors of vertex x labeled α between vertices y and z in ccw order if y and z are neighbors of x, and ∞ otherwise.

Recall that Lemma 5 encodes the string S constructed in Section 3.2 to support the computation of an arbitrary word of S_0, which is the balanced parenthesis sequence of the tree $\overline{T_0}$. In this section, we consider the `DFUDS` sequence (or *Depth First Unary Degree Sequence* [2] of $\overline{T_0}$), as the `DFUDS` order traversal visits the children of a node consecutively.

Lemma 6. *The string S can be stored in $(2\log_2 6+\epsilon)m+o(m)$ bits, for any ϵ such that $0<\epsilon<1$, to support in $O(1)$ time the operators listed in Lemma 1, as well as the computation of an arbitrary word, or $\Theta(n)$ bits of the balanced parenthesis sequence, and the* `DFUDS` *sequence of $\overline{T_0}$.*

Proof. We construct the same data structures as in Lemma 5, except when we encode S_0, we use the `TC` representation of the tree $\overline{T_0}$ [10]. To be specific, we encode S_0 using $(2+\epsilon)n+o(n)$ bits, for any ϵ such that $0<\epsilon<1$, and this encoding supports the computation of an

arbitrary word of the balanced parenthesis sequence, and the DFUDS sequence of $\overline{T_0}$ in constant time. As we can compute an arbitrary of the original sequence of S_0 in constant time and all the other structures are the same as in Lemma 5, we can still support the operators listed in Lemma 1 in constant time. □

The following lemma explains how to support label-based navigation operators on labeled planar triangulations. As Barbay *et al.* did for multi-labeled trees [1], we construct succinct indexes for labeled planar triangulations. We define the interface of the ADT of labeled planar triangulations through the operator `node_label`(v, i), which returns the i-th label associated to vertex v in lexicographic order.

Theorem 2. *Consider a multi-labeled planar triangulation $\mathcal{T}$ of n vertices, associated with σ labels in t pairs ($t > n$). Given the support of* `node_label` *in $f(n, \sigma, t)$ time on the vertices of $\mathcal{T}$, there is a succinct index using $t \cdot o(\lg \sigma)$ bits which supports* `lab_degree`, `lab_select_ccw` *and* `lab_rank_ccw` *in $O((\lg\lg\lg\sigma)^2(f(n, \sigma, t) + \lg\lg\sigma))$ time.*

Proof. The main idea is to combine our succinct representation of planar triangulations with three instances of the succinct indexes for related binary relations [1, 2].

We represent the combinatorial structure of $\mathcal{T}$ using Theorem 1, in which we use Lemma 6 to store S. Thus we can construct the auxiliary data structures for the DFUDS representation of $\overline{T_0}$ [13]. Observe that the sequence of the vertices (for simplicity we only consider internal vertices) referred by their numbers in three different orders, the order corresponding to an enumeration of the nodes of $\overline{T_0}$ in DFUDS order, π_1 and π_2, form three binary relations R_0, R_1 and R_2 with their associated labels.

We adopt the same strategy used previously for multi-labeled trees [1]. We can convert the rank of the vertices between π_0 and π_1, and between π_0 and π_2 in constant time by Theorem 1. We can also convert between the preorder number of the nodes in $\overline{T_0}$ (note that this is the same order as π_0) and the DFUDS number of the nodes in $\overline{T_0}$ in constant time [13]. Therefore, we can use the operator `node_label` to support the ADT of R_0, R_1 and R_2. Thus, for each of the binary relations R_1, R_2 and R_d we construct a succinct index using $t \cdot o(\lg \sigma)$ bits using Lemma 3.

Using properties of realizers of triangulations and the technique used in Barbay et al. [1] for dealing with each binary relation R_i, we are able to enumerate all the neighbors of a vertex in $O(\lg\lg\sigma\lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time per neighbor.

The main idea is using the fact that in each order π_i the children of a node lying in a tree T_i (the descendant) are enumerated consecutively: and the cyclic order of neighbors around a vertex is well described by the *local condition* characterizing realizers (see Definition 1). More precisely, to enumerate the children of a node v in T_0 we use DFUDS order on T_0, where children are listed consecutively. To enumerate the children of v in the tree T_1 (or T_2) it suffices to observe that they are listed consecutively by π_1 (or π_2): so this corresponds to a list of consecutive labels in the relation R_1 (or R_2): access to these labels can be efficiently performed using `label_rank` and `label_select` operators on R_1 (or R_2 or R_d). Finally, observe that even if the orders π_0, π_1 and π_2 on the vertices are completely different from known orders on trees (like preorder, postorder or DFUDS), Theorem 1 provides efficient tools for converting the rank of a node from an order π_i to a different one in $O(1)$ time. Hence the support of the rank/select operations on the neighbors of a node matching a label α is

simply reduced to a combination of `label_select` and `label_rank` operations on one of the binary relations R_1, R_2 or R_d at each time, following the orders defined on the three different types of parentheses.

Finally, we observe that the space requirements of our representation is dominated by the cost of the succinct indexes for the binary relations, each using $t \cdot o(\lg \sigma)$ bits. □

4 Edge Labeled Graphs with Pagenumber k

4.1 Multiple Parentheses

In this section, we consider the succinct representation of multiple parenthesis sequences of p types of parentheses, where p is not a constant. We consider the following operations on a multiple parenthesis sequence $S[1..2n]$ in addition to those defined in Lemma 1:

- `m_rank'`(S, i), the rank of the parenthesis at position i among parentheses of the same type in S;
- `m_findopen`(S, i) (`m_findclose`(S, i)), the matching closing (open) parenthesis of the same type for the open (closing) parenthesis at position i in S.

Note that `m_findopen` and `m_findclose` are identical to the operator `m_match`. We define them here for the simplicity of the proofs in this section.

Theorem 3. *A multiple parenthesis sequence of $2n$ parentheses of p types, in which the parentheses of any given type are balanced, can be represented using $2n \lg p + o(n \lg p)$ bits to support* `m_access`, `m_rank'`, `m_findopen` *and* `m_findclose` *in $O(\lg \lg p)$ time, and* `m_select`*,in $O(1)$ time. Alternatively, it can be represented using $4n \lg p + o(n \lg p)$ bits to support these operations in $O(1)$ time.*

Proof. We store the sequence as a string P over alphabet $\{(_1, (_2, ..., (_p,)_1,)_2, ...,)_p\}$ using Lemma 4.1 of [1]. P occupies at most $n \lg p + o(n \lg p)$ bits, and the operations `string_access` and `string_rank` can be supported in $O(\lg \lg p)$ time, and `string_select` can be supported in $O(1)$ time. We define another operation `string_rank'`(α, i), which returns the number of characters α in $P[1..i]$ if $P[i] = \alpha$. From the proof of Lemma 4.1 in [1], we have that `string_rank'`(i) can also be supported in $O(1)$ time.

We construct a balanced parenthesis sequence B_i, where $B_i[j]$ is an open (closing) parenthesis iff the j^{th} parenthesis of type i in the multiple parenthesis sequence is an open (closing) parenthesis. We denote the length of B_i by n_i, where n_i is the number of parentheses of type i. We store each B_i using the representation by Munro and Raman [15]. Thus the total space cost is $\sum_{i=1}^{p}(n_i + o(n_i)) = 2n + o(n)$.

The operation `m_access` can be supported by calling `string_access` on P once, so it can be supported in $O(\lg \lg p)$ time. To support `m_rank'`(S, i), we first compute the parenthesis, α, at position i using `m_access` in $O(\lg \lg p)$ time. Then `m_rank'`(S, i) = `string_rank'`$_P(\alpha, i)$. We also have `m_select`(S, α, i) = `string_select`$_P(\alpha, i)$. Finally, to support `m_findopen`(S, i) (the support for `m_findclose` is similar), we first compute the parenthesis $)_j$ at position i using `m_access`. Then we have `m_findopen`(S, i) = `m_select`$(S, (_j,$ `findclose`$_{B_\alpha}($`string_rank'`$_P((_p, i)))$.

To support all the above operations in constant time, we only need to support `string_access` on P in constant time. This can be achieved by writing down the sequence

P explicitly in addition to the data structures constructed above, and the total space will be increased by $2n \lg p$ bits. □

4.2 Graphs with Pagenumber k for large k

In this section, on unlabeled graphs with page number k, we consider the operations `adjacency` and `degree` defined in Section 3.2, and the operator `neighbors`(x), which returns the neighbors of vertex x.

Previous results on succinctly representing k-book embedded graphs [9, 15] support `adjacency` in $O(k)$ time. The lower-order term in the space cost of the result of Gavoille and Hanusse [9] is $o(km)$, which is dominant when k is large. Thus previous results mainly deal with the case when k is small. We consider the case when k is larger.

Theorem 4. *A graph of n vertices and m edges with pagenumber k can be represented using $n + 2m \lg k + o(m \lg k))$ bits to support operations* `adjacency` *in $O(\lg k \lg \lg k)$ time,* `degree` *in $O(1)$ time, and* `neighbors`(x) *in $O(d(x) \lg \lg k)$ time where $d(x)$ is the degree of x. Alternatively, it can be represented in $n + 4m \lg k + o(m \lg k))$ bits to support operations* `adjacency` *in $O(\lg k)$ time,* `degree` *in $O(1)$ time, and* `neighbors`(x) *in $O(d(x))$ time.*

Proof. We denote each vertex by its rank along the spine of the book (i.e. vertex x is the x^{th} vertex along the spine). An edge between vertices x and y is a *left edge* (or *right edge*) of x if $y > x$ (or $y < x$).

We construct a bit vector B of $n + m$ bits to encode the degree of each node in unary as in [12], in which vertex x corresponds to the x^{th} 1 followed by $d(x)$ 0s. We encode B using $n + m + o(n + m)$ bits to support rank / select operations [8]. We construct a multiple parenthesis sequence S of $2m$ parentheses of k types as follows. For each node $x = 1, 2, ...n$ and for each page $i = 1, 2, ..., k$,

1. If there are j left edges of x on page i where $j > 0$, we write down $j - 1$ copies of the symbol $)_i$.
2. Assume that the left edges of x are on pages $p_1, p_2, ..., p_l$. We sort the sequence $)_{p_1},)_{p_2}, ...,)_{p_l}$ by the largest span of the left edges of x on these pages and write down the sorted sequence, i.e. in the sorted sequence, $')'_{p_u}$ appears before $')'_{p_v}$ if the largest span of the left edges of x on page p_u is less than the maximum span of the left edges of x on page p_v.
3. Similarly, we assume that the right edges of x are on pages $q_1, q_2, ..., q_r$. We sort the sequence $(_{q_1}, (_{q_2}, ..., (_{p_l}$ by the maximum span of the right edges of x on these pages and write down the sorted sequence.
4. If there are j' right edges of x on page i where $j' > 0$, we write down $j' - 1$ copies of the symbol $(_i$.

The Sequence S, although appears to be similar to the sequence in Theorem 2 of [9], it differs in the order we store the parentheses corresponding to the edges of a given vertex. It also has $2m$ parentheses of k types, and we encode it using Theorem 3 in $2m \lg k + o(m \lg k)$ bits. Finally we construct a bit vector B' of $2m$ bits in which $B[i] = 1$ iff $S[i]$ is a closing parentheses, and encoding it using $2m + o(m)$ bits to support rank / select operations. Thus the total space cost is $n + 2m \lg k + o(m \lg k)$ bits.

With the above definitions and structures, the algorithm [12] to check whether there is an edge between vertices x and y on page p can be described as follows (assume, without loss of generality, that $x < y$). We first retrieve the index, w, of the parenthesis in S that corresponds to the right edge of x with the largest span on page p. Because this occurrence is the first occurrence of the character $(_p$ in S after position $\texttt{bin\_rank}_B(0, \texttt{bin\_select}_B(1, x))$, we can compute w in $O(\lg \lg k)$ time. We then retrieve the index, t, of the closing parenthesis that matches $B[w]$ in $O(\lg \lg k)$ time, and if it corresponds to a left edge of y (this is true iff $\texttt{bin\_rank}_B(1, t) = y$), then there is an edge between x and y. Similarly, we retrieve the parenthesis in S that corresponds to the left edge of y with the largest span on page p, and if its matching open parenthesis corresponds to a right edge of x, then x and y are adjacent. If the above process cannot find an edge between x and y, then x and y are not adjacent. All these steps takes $O(\lg \lg k)$ time.

To compute $\texttt{adjacency}(x, y)$ (assume, without loss of generality, that $x < y$), we first observe that by Step 2 of the construction algorithm of S, the open parentheses that correspond to the right edges of x with the largest spans among the right edges of x on the same pages form a substring of S. We can compute the starting position of this substring using B and B' in constant time. Because these parentheses are sorted by the spans of the edges they correspond to, we can perform a doubling searching to check whether one of these edges connects x and y. In each step of the doubling search, we perform the algorithm in the last paragraph in $O(\lg \lg k)$ time. There are at most k such parentheses, so we perform the algorithm at most $O(\lg k)$ times. Similarly, we perform doubling search on the left edges of y with the largest spans among the left edges of y on the same pages. Thus we can test the adjacency between two vertices in $O(\lg k \lg \lg k)$ time.

The degree of any vertex can be easily computed in constant time using B. We can also perform the algorithms in previous work [12] to compute $\texttt{neighbors}(x)$, and it takes $O(d(x) \lg \lg k)$ time on our data structures.

Finally, to improve the time efficiency, we can store S using $4m \lg k + o(m \lg k)$ bits using Theorem 3 to achieve the other tradeoff. □

4.3 Edge Labeled Graphs with Pagenumber k

We consider the following operations on edge labeled graphs:
- $\texttt{lab\_adjacency}(\alpha, x, y)$, whether there is an edge labeled α between vertices x and y;
- $\texttt{lab\_degree\_edge}(\alpha, x)$, the number of edges of vertex x that are labeled α;
- $\texttt{lab\_edges}(\alpha, x)$, the edges of vertex x that are labeled α.

We first show how to design succinct representation for an edge labeled graph with one page.

Lemma 7. *An outerplanar graph of n vertices and m edges in which each edge has a label from alphabet $[\sigma]$ can be represented using $n + m(\lg \sigma + o(\lg \sigma))$ bits to support operation* `lab_adjacency` *in $O(1)$ time,* `lab_degree_edge` *in $O(\lg \lg \sigma (\lg \lg \lg \sigma)^2)$ time, and* $\texttt{lab\_edges}(\alpha, x)$ *in $O(\texttt{lab\_degree\_edge}(\alpha, x) \lg \lg \sigma \lg \lg \lg \sigma)$ time.*

Proof. We construct a bit vector B of $n + m$ bits to encode the degree of each node in unary as in the proof of Theorem 4. We construct a balanced parenthesis sequence P as follows. List the vertices from left to right along the spine, and each node is represented by zero or

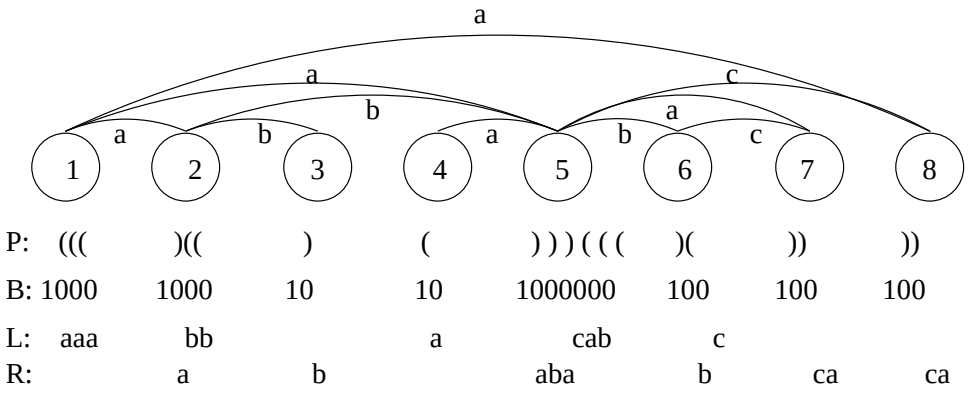

Fig. 3. An example of the succinct representation of a labeled graph with one page.

more closing parentheses followed by zero or more open parentheses, where the number of closing (or open) parentheses is equal to the number of left (or right) edges. We construct a string R, where $R[i]$ is the label of the edge corresponds to the i^{th} open parenthesis in P. We build a succinct index for R to support rank / select operations using Theorem 3.1 of [1]. We have a similar conceptual string L, where $L[i]$ is the label of the edge corresponds to the i^{th} closing parenthesis in P, but we do not store R explicitly. Instead, we construct a succinct index for L to support rank / select operations using Theorem 3.1 of [1]. These data structures occupy $n + m(\lg \sigma + o(\lg \sigma))$ in total. See Figure 3 for an example.

To use the succinct index constructed above to support rank / select on L, we need to show how to compute `string_access`(i) on R. We first compute the index, j, of the i^{th} closing parenthesis in P. Then we find its matching open parenthesis and let u denote its index in P and $L[u]$ is the results. All these operations can be computed in constant time, so we can support `string_access` on L in constant time. Thus we can support `string_rank` and `string_select` on L in $O(\lg \lg \sigma (\lg \lg \lg \sigma)^2)$ time and $O(\lg \lg \sigma \lg \lg \lg \sigma)$ time, respectively.

To compute `lab_adjacency`(x, y), we first use the algorithm in [12] to check whether x and y are adjacent. If they are, we retrieve the index of open parenthesis in P that corresponds to the edge between x and y, compute its rank, v, among open parenthesis, and if $R[v] = \alpha$, we return true. This takes constant time.

To compute `lab_degree_edge`(α, x), we need to compute the number, l, of the left edges of x that are labeled α, and the number, r, of the right edges of x that are labeled α. To compute l, we first compute the indices l_1 and l_2 such that each parenthesis in the substring $P[l_1..l_2]$ is a closing parentheses that corresponds to an left edge of x, using rank / select operations on B and P in constant time. We then use `string_rank` and `string_select` to compute the number of occurrences of α in the substring $L[l_1..l_2]$ in $O(\lg \lg \sigma (\lg \lg \lg \sigma)^2)$ time. Similarly we can compute r by performing rank / select operations on B, P and R, and the sum of l and r is the answer. To further list all the edges of x that is labeled α, we need to perform `string_select` on L and R to retrieve the indices of the corresponding parentheses in P, and perform rank / select operations on B to retrieve the vertices that these edges connect to. □

To support an edge labeled graph with k pages, we can use Lemma 7 to represent each page and combine all the pages represented in this way to support navigational operations.

Alternatively, we can use the result of Theorem 4 and a similar approach in the proof of Lemma 7 to achieve a different tradeoff to improve the time efficiency when k is large.

Theorem 5. *A k-book graph of n vertices and m edges in which each edge has a label from alphabet $[\sigma]$ can be represented using $kn + m(\lg \sigma + o(\lg \sigma)$ bits to support operation* `lab_adjacency` *in $O(k)$ time,* `lab_degree_edge` *in $O(k \lg\lg \sigma (\lg\lg\lg \sigma)^2)$ time, and* `lab_edges`(α, x) *in $O($*`lab_degree_edge`$(\alpha, x) \lg\lg \sigma \lg\lg\lg \sigma + k)$ time. Alternatively, it can be represented using $n + 4m \lg k + o(m \lg k) + m(\lg \sigma + o(\lg \sigma))$ bits to support operation* `lab_adjacency` *in $O(\lg k)$ time,* `lab_degree_edge` *in $O(\lg\lg \sigma (\lg\lg\lg \sigma)^2)$ time, and* `lab_edges`(α, x) *in $O($*`lab_degree_edge`$(\alpha, x) \lg\lg \sigma \lg\lg\lg \sigma)$ time.*

In particular, a labeled planar graph can be represented using $4n + m(\lg \sigma + o(\lg \sigma)$ bits to support operation `lab_adjacency` *in $O(1)$ time,* `lab_degree_edge` *in $O(\lg\lg \sigma (\lg\lg\lg \sigma)^2)$ time, and* `lab_edges`(α, x) *in $O($*`lab_degree_edge`$(\alpha, x) \lg\lg \sigma \lg\lg\lg \sigma)$ time.*

The above approach can be used directly to support multi-labeled graphs with pagenumber k. The only change is that the string L will become a binary relation, encoded using Lemma 3.

5 Concluding Remarks

In this paper, we present a framework of succinctly representing the properties of the graphs in the form of labels. We expect that our approach can be extended to support other types of planar graphs, which is an open research topic. Another open problem is to design succinct representation of vertex labeled graphs with pagenumber k.

Our final comment is that because Theorem 2 provides a succinct index for vertex labeled planar triangulations, we can in fact store the labels in compressed form as Barbay *et al.* have done to compress strings, binary relations and multi-labeled trees [1], while still supporting various operations. The same applies to Theorem 5, as we use succinct indexes for strings to encode the labels.

References

[1] Jérémy Barbay, Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689. ACM-SIAM, ACM, 2007.

[2] David Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing trees of higher degree. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, pages 169–180. Springer-Verlag LNCS 1663, 1999.

[3] Frank Bernhart and Paul C. Kainen. The book thickness of a graph. *Journal of Combinatorial Theory, Series B*, 27(3):320–331, 1979.

[4] L. Castelli-Aleardi, O. Devillers, and G. Schaeffer. Succinct representation of triangulations with a boundary. In *Proc. 9th Workshop on Algorhtms and Data Structures (WADS)*, volume 3608 of *LNCS*, pages 134–145. Springer, 2005.

[5] L. Castelli-Aleardi, O. Devillers, and G. Schaeffer. Optimal succinct representations of planar maps. In *Proc. of 22nd ACM Annual Symposium on Computational Geometry (SoCG)*, pages 309–318, 2006.

[6] Y.-T. Chiang, C.-C. Lin, and H.-I. Lu. Orderly spanning trees with applications to graph encoding and graph drawing. *SODA*, pages 506–515, 2001.

[7] R.C.-N Chuang, A. Garg, X. He, M.-Y. Kao, and H.-I. Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. *Automata, Laguages and Programming*, pages 118–129, 1998.

[8] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[9] Cyril Gavoille and Nicolas Hanusse. On compact encoding of pagenumber k graphs. *Discrete Mathematics & Theoretical Computer Science*, 2004. To appear.

[10] Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct ordinal trees based on tree covering. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming*, 2007. To Appear.

[11] Martin Isenburg and Jack Snoeyink. Face fixer: Compressing polygon meshes with properties. In *Proceedings of SIGGRAPH 2000*, pages 263–270, 2000.

[12] G. Jacobson. Space efficients static trees and graphs. In *Proceedings of the IEEE Symposium on Foundations of Computerb Science (FOCS)*, pages 549–554, 1989.

[13] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[14] Hsueh-I Lu and Chia-Chi Yeh. Balanced parentheses strike back. Accepted to *ACM Transactions on Algorithms* upon minor revision, 2007.

[15] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[16] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[17] Walter Schnyder. Embedding planar graphs on the grid. In *Proc. of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 138–148, 1990.

[18] Mihalis Yannakakis. Four pages are necessary and sufficient for planar graphs. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Theory of Computing*, pages 104–108, 1986.