Fast and Compact Planar Embeddings[☆]

Leo Ferres[a], José Fuentes-Sepúlveda[b,c], Travis Gagie[c,d,*], Meng He[e], Gonzalo Navarro[b,c,f]

[a]*Faculty of Engineering, Universidad del Desarrollo & Telefónica I+D, Santiago, Chile*
[b]*Department of Computer Science, University of Chile, Santiago, Chile*
[c]*Center of Biotechnology and Bioengineering (CeBiB), Chile*
[d]*School of Computer Science and Telecommunications, Diego Portales University, Santiago, Chile*
[e]*Faculty of Computer Science, Dalhousie University, Halifax, Canada*
[f]*Millennium Institute for Foundational Research on Data (IMFD), Chile*

Abstract

There are many representations of planar graphs, but few are as elegant as Turán's (1984): it is simple and practical, uses only 4 bits per edge, can handle self-loops and multi-edges, and can store any specified embedding. Its main disadvantage has been that "it does not allow efficient searching" (Jacobson, 1989). In this paper we show how to add a sublinear number of bits to Turán's representation such that it supports fast navigation while retaining simplicity. As a consequence of the inherited simplicity, we offer the first efficient parallel construction of a compact encoding of a planar graph embedding. Our experimental results show that the resulting representation uses about 6 bits per edge in practice, supports basic navigation operations within a few microseconds, and can be built sequentially at a rate below 1 microsecond per edge, featuring a linear speedup with a parallel efficiency around 50% for large datasets.

Keywords: Planar embedding, Compact data structures, Parallel construction

1. Introduction

The rate at which we generate data is increasing even faster than the speed and capacity of computing hardware. Thus, if we want to use efficiently what we generate, we need to represent it in better ways. The surge in the number and complexity of the maps we want to have available on mobile devices is particularly pronounced and has resulted in diverse ways to store planar graphs.

[☆]A previous version of this paper appeared in the 15th Algorithms and Data Structures Symposium (WADS 2017)[1].

[*]Corresponding author

Email addresses: `lferres@udd.cl` (Leo Ferres), `jfuentess@dcc.uchile.cl` (José Fuentes-Sepúlveda), `travis.gagie@mail.udp.cl` (Travis Gagie), `mhe@cs.dal.ca` (Meng He), `gnavarro@dcc.uchile.cl` (Gonzalo Navarro)

Each of these representations has its disadvantages, however, so in this paper we introduce a compact representation of a planar graph that supports fast navigation, can represent arbitrary planar embeddings and is easy to build, and we demonstrate its practicality.

More concretely, as described in Section 2, Tutte [2] showed that a planar embedding of a connected graph with n nodes and m edges can be represented in $m \log 12 \approx 3.58m$ bits, and Blelloch and Farzan [3] gave a data structure matching this bound except for $o(m)$-bit redundancy, that in addition supports efficient navigation. Their structure is complex, however, and no implementation has been attempted. The much simpler representation of Turán [4] uses $4m$ bits, which is still close to the lower bound, but it does not support navigation. Other representations have the same issue with nagivation [5], take more than $4m + o(m)$ bits [6, 7], or cannot represent multi-edges or certain planar embeddings [8, 5, 9]. For graphs large enough to require compact representations, how to build those representations is also likely to be an issue and, in particular, whether it is possible to do so in parallel. We are not aware of any previous work on efficient parallel construction of compact representations of planar graphs and we finish Section 2 by reviewing some obstacles for specific representations.

Our contribution in this paper is threefold:

1. We show how to add $o(m)$ bits to Turán's representation such that it supports fast navigation. We can list the edges incident to any vertex in clockwise or counter-clockwise order using constant time per edge, including starting the enumeration at any desired neighbour. As a consequence, we can also list the nodes on a face in constant time per node. We can also find a vertex's degree in time $\mathcal{O}(f(m))$ for any $f(m) \in \omega(1)$, and determine whether two vertices are neighbours in $\mathcal{O}(f(m))$ time for any $f(m) \in \omega(\log m)$.

2. We give a parallel algorithm that builds our data structure from any spanning tree of the planar embedding, in $\mathcal{O}(m)$ work and $\mathcal{O}(\log m)$ span ($\mathcal{O}(\log^2 m)$ span to support the neighbour query). This is the first linear-work practical parallel algorithm for building compact representations of planar embeddings.

3. We implement and experimentally evaluate the space, query, and construction performance of our representation. In practice, our structure uses less than $6m$ bits, performs navigation operations within a few microseconds, and can be built sequentially at a rate below 1 microsecond per edge. The parallel algorithm scales linearly, with an efficiency around 50% for large datasets, with up to 24 processors.

Summarizing, we offer the first simple compact representation of planar embeddings, which is easy to program, uses little space, and is efficiently built and navigated. Our structure is thousands of times faster than the classical adjacency list structure when such structure cannot fit in main memory, but ours fits due to its smaller memory footprint. We have made the code publicly available at `http://www.dcc.uchile.cl/~jfuentess/pemb/`.

Turán chooses an arbitrary spanning tree of the graph, roots it at a vertex on the outer face and traverses it, writing its balanced-parentheses representation as he goes. This sequence is interleaved with another, over a different binary alphabet, consisting of an occurrence of one character for the first time he sees each edge not in the tree and an occurrence of the other character for the second time he sees that edge. These two sequences can be written as three sequences over $\{0, 1\}$: one of length $2n - 2$ encoding the balanced-parentheses representation of the tree; one of length $2m - 2n + 2$ encoding the interleaved sequence; and one of length $2m$ indicating how they are interleaved. Our extension of this representation is based on the observation that the interleaved sequence encodes the balanced-parentheses representation of the complementary spanning tree of the dual of the graph. By adding a sublinear number of bits to each balanced-parentheses representation, we can support fast navigation in the trees, and by storing the sequence indicating the interleaving as a bitvector with support for operations rank and select [6], we can support fast navigation in the graph.

Section 2 surveys the related work on compact representations of planar embeddings. Section 3 describes bitvectors and the balanced-parentheses representation of trees, which are the building blocks of our extension of Turán's representation. We also describe the model of parallelism we use in our construction algorithms. In Section 4 we prove the observation mentioned above on Turán's interleaved sequence. In Section 5 we describe our data structure and how we implement queries. Section 6 describes our parallel construction algorithm and discusses some implementation issues. In Section 7 we describe our experiments on space, query and construction performance, and discuss the results. Finally, in Section 8 we present our conclusions and future work directions.

2. Related work

Before discussing the related work, let us distinguish between planar graphs and planar embeddings. A planar graph is a graph that can be drawn in the plane without intersection of its edges, except in their ending points. A particular drawing of a planar graph induces an embedding, which is defined by the resulting ordering of the neighbors of the nodes when read in, say, counterclockwise order. This defines the resulting faces, and the embedding must also define which is the external face (this is not the case of an embedding in a sphere). In unconnected planar graphs, in addition, the embeddings must also define which components are drawn inside which faces of other components. A planar graph may have more than one planar embedding. For example, in the graph of Figure 1, if we move the vertex 3 towards the interior of the face delimited by the vertices 1, 2, 6, and 5, we obtain a different embedding of the same graph. Some compact representations of planar graphs, explained later in this section, need to select a suitable embedding of the graph. In many practical applications, however, we are asked to store a particular embedding of a graph, and cannot change it to suit our representation. With respect to operations, when listing the neighbours of a vertex, the order of the neighbors is relevant

for planar embeddings, but not for planar graphs. Other operations, like listing the vertices that delimit a face, make sense only for planar embeddings.

For the rest of the document we assume that the planar graphs and embeddings are connected. We will refer explicitly to unconnected planar graphs and embedding when we state results about them. Further, we note that all the representations described below renumber the vertices in some convenient way (since storing a specified order on the nodes takes $\Theta(n \log n)$ bits). Applications can then use these arbitrary node identifiers assigned by the representations.

Tutte [2] showed that representing a specified embedding of a connected planar multi-graph with n vertices and m edges requires $m \log 12 \approx 3.58m$ bits in the worst case. Turán [4] gave a very simple representation that uses $4m$ bits. Jacobson [6] argued that this representation "does not allow fast searching" and (introducing techniques that we will apply to Turán's representation) proposed one that instead uses $\mathcal{O}(m)$ bits and supports fast navigation, based on book embeddings [10]. Munro and Raman [8] estimated that Jacobson's representation uses $64n$ bits and proposed one using $2m + 8n + o(m)$ bits that retains fast navigation, still based on the same book embeddings (but this does not handle self-loops). Keeler and Westbrook [5] also noted that "the constant factor in [Jacobson's] space bound is relatively large" and gave a representation that uses $m \log 12 + \mathcal{O}(1)$ bits for planar graphs (not embeddings), as well as for planar embeddings containing either no self-loops or no vertices with degree 1; however, they again gave up fast navigation. Chiang, Lin and Lu [9], improving previous work by Chuang *et al.* [11], gave a representation (without allowing self-loops) that uses $2m + 3n + o(m)$ bits with fast navigation, based on so-called orderly spanning trees. However, although all planar graphs can be represented with orderly spanning trees, some planar embeddings cannot. For simple planar embeddings (i.e., no self-loops nor multiple edges, thus $m \leq 3n$), their space decreases to $2n + 2m + o(m) \leq 4m + o(m)$ on connected graphs. Barbay *et al.* [7] gave a data structure that uses $\mathcal{O}(m)$ bits to represent simple planar graphs with fast navigation, based on realizers of planar triangulations [12]. Still, their constant is relatively large, $18n + o(m)$. Finally, Blelloch and Farzan [3], extending the work of Blandford *et al.*. [13], matched for the first time Tutte's lower bound on general planar embeddings, with a structure that uses $m \log 12 + o(m)$ bits and supports fast navigation. Their structure is based on small vertex separators [14]. They can also represent any planar graph within its lower-bound space plus a sublinear redundancy, even when the exact lower bound is unknown for general planar graphs [15]. While Blelloch and Farzan closed the problem in theoretical terms, their representation is complicated and has not been implemented. Other authors [16, 17, 18, 19, 20, 21] have considered special kinds of planar graphs, notably tri-connected planar graphs and triangulations. We refer the reader to Munro and Nicholson's [22] and Navarro's [23, Chapter 9] recent surveys for further discussion of compact data structures for graphs.

Most of the navigable representations we have mentioned require complicated construction algorithms, generally defying a parallel implementation. It is not known how to compute a book embedding [10] in parallel, which is necessary to build the representations of Jacobson and of Munro and Raman. There

are also no parallel algorithms to build orderly spanning trees [9], necessary for the representation of Chiang *et al.* Its predecessor [11] uses instead a triangulation and a canonical ordering; for the latter there is only a CREW construction running in $\mathcal{O}\big(\log^4 n\big)$ time with n^2 processors [24]. As for the vertex separators [14] required to build the representation of Blandford *et al.* and of Blelloch and Farzan, Kao *et al.* [25] designed a linear-work, logarithmic-span algorithm for computing a cycle separator of a planar graph. However, the construction of these representations of planar embeddings decompose the input graph by repeatedly computing separators until each piece is sufficiently small. This increases the total work to $\mathcal{O}(n \log n)$ even if this optimal parallel algorithm is used. The best linear-work parallel algorithms [26] for building the realizers [12] used in the construction of Barbay *et al.*'s representation have $\mathcal{O}(\log n)$ span in the expected case but $\mathcal{O}(\log n \log \log n)$ deterministic span.

3. Preliminaries

3.1. Bitvectors and parentheses

A bitvector is a binary string that supports the queries rank and select in addition to random access, where $\mathsf{rank}_b(i)$ returns the number of bits set to b in the prefix of length i of the string and $\mathsf{select}_b(j)$ returns the position of the jth bit set to b. For convenience, we define $\mathsf{select}_b(0) = 0$.

It is possible to represent a bitvector of length ℓ in $\ell + o(\ell)$ bits and support random access, rank and select in constant time [6, 27, 28]. Furthermore, if the bitvector has k 1s, it can be represented in $\log \binom{\ell}{k} + o(\ell)$ bits [29], which is $\ell H(k/\ell) + o(\ell) = k \log(\ell/k) + \mathcal{O}(k) + o(\ell)$, with $H(x) = -x \log x - (1-x) \log(1-x)$. All our logarithms are to the base 2 unless otherwise stated.

By adding some further operations on the bitvectors, we can represent an ordered tree or forest of t vertices using $2t + o(t)$ bits and support natural navigation queries in constant time. One of the most popular such representations is a string of balanced parentheses: we traverse each tree from left to right, writing an opening parenthesis when we first visit a vertex (starting at the root) and a closing parenthesis when we leave it for the last time (or, in the case of the root, when we finish the traversal). We can encode the string of parentheses as a bitvector of length $2t$, with 0s encoding opening parentheses and 1s encoding closing parentheses. By adding $o(t)$ further bits, we can support in constant time, among others, the following queries used by our solution [8, 30, 31]:

- $\mathsf{match}(i)$ locates the position of the parenthesis matching the ith parenthesis in the bitvector (i.e., finds the other parenthesis referring to the same node);

- $\mathsf{parent}(v)$ returns the parent of v, or 0 if v is the root of its tree. Nodes v and $\mathsf{parent}(v)$ are represented as their pre-order rank in the traversal.
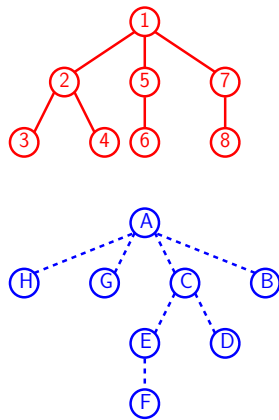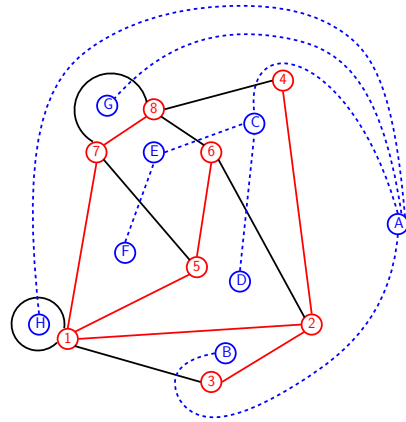
5

As we focus on practical parallel algorithms, we describe and analyze our construction using the *Dynamic Multithreading (DyM) Model* [32] (we nevertheless express our final results in terms of the PRAM model as well). In the DyM model, a multithreaded computation is modelled as a directed acyclic graph (DAG) where vertices are instructions and edge (u, v) represents precedence between instructions u and v. The model is based on two parameters of the multithreaded computation: its *work* T_1 and its *span* T_∞. The work is the running time on a single thread, that is, the number of nodes (i.e., instructions) in the DAG, assuming each instruction takes constant time. The span is the length of the longest path in the DAG, that is, the intrinsically sequential part of the computation. The time T_p needed to execute the computation on p threads then has complexity $\Theta(T_1/p + T_\infty)$, which can be reached with a greedy scheduler. The improvement of a multithreaded computation using p threads is called *speedup*, T_1/T_p. The upper bound on the achievable speedup, T_1/T_∞, is called *parallelism*. Finally, the *efficiency* is defined as T_1/pT_p and can be interpreted as the percentage of improvement achieved by using p cores or how close we are to the linear speedup. In the DyM model, the workload of the threads is balanced by using the *work-stealing* algorithm [33].

To describe parallel algorithms in the DyM model, we augment sequential pseudocode with three keywords. The **spawn** keyword, followed by a procedure call, indicates that the procedure should run in its own thread and may thus be executed in parallel to the thread that spawned it. The **sync** keyword indicates that the current thread must wait for the termination of all threads it has spawned. Finally, **parfor** is "syntactic sugar" for **spawn**ing one thread per iteration in a for loop, thereby allowing these iterations to run in parallel, followed by a **sync** operation that waits for all iterations to complete. In practice, the **parfor** keyword is implemented by halving the range of loop iterations, spawning one half and using the current procedure to process the other half recursively until reaching one iteration per range. After that, the iterations are executed in parallel. Therefore, this implementation adds an overhead bounded above by the logarithm of the number of loop iterations. We include such overheads in our complexities.

4. Spanning trees of planar graphs

It is well known [34, 35, 36] that for any spanning tree T of a connected planar graph G, the edges dual to T are a spanning tree T^* of the dual of G; see Figure 1 for an illustration (including multi-edges and a self-loop). (The trees T and T^* are sometimes called interdigitating because of the way their branches interlock.) If we choose T as the spanning tree of G for Turán's representation, then we store a 0 and a 1, in that order, for each edge in T^*. We now show that these bits encode a traversal of T^*, using the notion of *incidence*. Given a face f and a vertex v, an incidence of the face f in the vertex v is a pair of edges e and e', such that v is part of f, and both e and e' are incident edges of

Figure 1: **Top left:** A planar embedding of a planar graph G, with a spanning tree T of G shown in red and the complementary spanning tree T^* of the dual of G shown in blue with dashed lines. **Bottom left:** The two spanning trees, with T rooted at the vertex 1 on the outer face and T^* rooted at the vertex A corresponding to the outer face. **Right:** The list of edges we process while traversing T starting at 1 and processing edges in counter-clockwise order, with the edges in T shown in red and the ones in $G - T$ shown in black; the edges of T^* corresponding to the edges in $G - T$ are shown in blue.

	T	$G - T$	T^*
1		(1, 3)	(A, B)
2	(1, 2)		
3	(2, 3)		
4		(1, 3)	(A, B)
5	(2, 3)		
6	(2, 4)		
7		(4, 8)	(A, C)
8	(2, 4)		
9		(2, 6)	(C, D)
10	(1, 2)		
11	(1, 5)		
12	(5, 6)		
13		(2, 6)	(C, D)
14		(6, 8)	(C, E)
15	(5, 6)		
16		(5, 7)	(E, F)
17	(1, 5)		
18	(1, 7)		
19		(5, 7)	(E, F)
20	(7, 8)		
21		(6, 8)	(C, E)
22		(4, 8)	(A, C)
23		(7, 8)	(A, G)
24	(7, 8)		
25		(7, 8)	(A, G)
26	(1, 7)		
27		(1, 1)	(A, H)
28		(1, 1)	(A, H)

f and are consecutive in the adjacency list of v. For example, in Figure 1, the edges $(1, 7)$ and $(1, 1)$, or $(1, 1)$ and $(1, 3)$ are incidences of the outer face A in the vertex 1. Notice that the edges $(1, 1)$ and $(1, 1)$ are not a valid incidence of the face A and vertex 1, since those edges are incident to the face H.

Lemma 1. *Consider any planar embedding of a planar graph G, any spanning tree T of G and the complementary spanning tree T^* of the dual of G. Suppose*

we perform a depth-first traversal of T starting from any vertex on the outer face of G and always process the edges incident to the vertex v we are visiting in counter-clockwise order. At the root, we choose arbitrarily an incidence of the outer face in the root and start from the last edge of the incidence in counter-clockwise order; at any other vertex, we start from the edge immediately after the one to that vertex's parent. Then each edge not in T corresponds to the next edge we cross in a depth-first traversal of T^*.

PROOF. Suppose the traversal of T^* starts at the vertex of the dual of G corresponding to the outer face of G. We now prove by induction that the vertex we are visiting in T^* always corresponds to the face of G incident to the vertex we are visiting in T and to the previous and next edges in counter-clockwise order.

Our claim is true before we process any edges, since we order the edges starting from an incidence of the outer face in the root of T. Assume it is still true after we have processed $i < m$ edges, and that at this time we are visiting v in T and v^* in T^*. First suppose that the $(i+1)$th edge (v, w) we process is in T. We note that $w \neq v$, since otherwise (v, w) could not be in T. We cross from v to w in T, which is also incident to the face corresponding to v^*. Now (v, w) is the previous edge — considering their counter-clockwise order at w, starting from (v, w) — and the next edge (which is (v, w) again if w has degree 1) is also incident to v^*. This is illustrated on the left side of Figure 2. In fact, the next edge is the one after (v, w) in a clockwise traversal of the edges incident to the face corresponding to v^*.

Now suppose (v, w) is not in T and let w^* be the vertex in T^* corresponding to the face on the opposite side of (v, w), which is also incident to v. We note that $w^* \neq v^*$, since otherwise (v, w) would have to be in T. We cross from v^* to w^* in T^*. Now (v, w) is the previous edge — this time still considering their counter-clockwise order at v — and the next edge (which may be (v, w) again if it is a self-loop) is also incident to w^*. This is illustrated on the right side of Figure 2. In fact, the next edge is the one that follows (v, w) in a clockwise traversal of the edges incident to the face corresponding to w^*.

Since our claim remains true in both cases after we have processed $i + 1$ edges, by induction it is always true. In other words, whenever we should process an edge e in G that is not in T, we are visiting in T^* one of the vertices corresponding to the faces incident to e (i.e., one of the endpoints of the edge in the dual of G that corresponds to e). Since we process each edge in G twice, once at each of its endpoints or twice at its unique endpoint if it is a self-loop, it follows that the list of edges we process that are not in T, corresponds to the list of edges we cross in a traversal of T^*. □

For example, in Figure 1, if we choose the valid incidence $(1, 7)$ and $(1, 1)$, then we start the traversal of Lemma 1 at edge $(1, 1)$. On the other hand, if we choose the incidence $(1, 1)$ and $(1, 3)$, we start the traversal at edge $(1, 3)$. We process the edges in counter-clockwise order so that the traversals of T and T^* are from left to right and from right to left, respectively; processing them in clockwise order would reverse those directions. For example, for the embedding

8

Figure 2: **Left:** If we process an edge (v, w) in T, then we move to w in our traversal of T and the next edge, (w, x) in this case, is also incident to the vertex v^* we are visiting in our traversal of T^*. **Right:** If (v, w) is not in T, then in T^* we move from v^* to the vertex w^* corresponding to the face on the opposite side of (v, w) in G. The next edge, (v, y) in this case, is also incident to w^*.

in Figure 1, if we start the traversal of the red tree T at vertex 1 and start processing the edges at $(1, 3)$, then we process them in the order shown at the right of the figure.

5. Data structure

Our extension of Turán's representation of a planar embedding of a connected planar graph G with n vertices and m edges consists of the following components, which take $4m + o(m)$ bits:

- a bitvector $A[1..2m]$ in which $A[i] = 1$ if and only if the ith edge we process in the traversal of T described in Lemma 1, is in T;

- a bitvector $B[1..2(n-1)]$ in which $B[i] = 0$ if and only if the ith time we process an edge in T during the traversal, is the first time we process that edge;

- a bitvector $B^*[1..2(m-n+1)]$ in which $B^*[i] = 0$ if and only if the ith time we process an edge not in T during the traversal, is the first time we process that edge.

Notice B encodes the balanced-parentheses representation of T, except that it lacks the leading 0 and trailing 1 encoding the parentheses for the root. By Lemma 1, B^* encodes the balanced-parentheses representation of a traversal of the spanning tree T^* of the dual of G complementary to T (the right-to-left traversal of T^*, in fact), except that it also lacks the leading 0 and trailing 1 encoding the parentheses for the root. Therefore, since B and B^* encode forests, we can support match and parent with them.

To build A, B and B^* given the embedding of G and T, we traverse T as in Lemma 1. Whenever we process an edge, if it is in T then we append a 1 to A and append the edge to a list L; otherwise, we append a 0 to A and append the edge to another list L^*. When we have finished the traversal, we replace each edge in L or L^* by a 0 if it is the first occurrence of that edge in that list, and by a 1 if it is the second occurrence; this turns L and L^* into B and B^*, respectively. For the example shown in Figure 1, L and L^* eventually contain

9

the edges shown in the columns labelled T and $G - T$, respectively, in the table on the on the right side of the figure, and

$$
\begin{aligned}
A[1..28] &= 0110110101110010110100010100 \\
B[1..14] &= 00101100110011 \\
B^*[1..14] &= 01001001110101 \,.
\end{aligned}
$$

We identify each vertex v in G by its pre-order rank in our traversal of T. We say that, while we visit v, we process all the edges that lead from v to other nodes w. Note that each edge (v, w) is processed twice, while visiting v and while visiting w, but these correspond to two distinct positions in our traversal. Consider the following queries:

first(v): return i such that the first edge we process while visiting v is the ith we process during our traversal;

last(v): return i such that the last edge we process while visiting v is the ith we process during our traversal;

next(i): return j such that if we are visiting v when we process the ith edge during our traversal, then the next edge we process when visiting v, in counter-clockwise order, is the one we process jth;

prev(i): return j such that if we are visiting v when we process the ith edge during our traversal, then the previous edge we processed when visiting v, in counter-clockwise order, is the one we process jth;

mate(i): return j such that we process the same edge ith and jth during our traversal;

vertex(i): return the vertex v such that we are visiting v when we process the ith edge during our traversal.

With these it is straightforward to reenact our traversal of T and recover the embedding of G. For example, with the following queries we can list the edges incident to the root of T in Figure 1 and determine whether they are in T:

first(1) = 1	mate(1) = 4	vertex(4) = 3	$A[1] = 0$
next(1) = 2	mate(2) = 10	vertex(10) = 2	$A[2] = 1$
next(2) = 11	mate(11) = 17	vertex(17) = 5	$A[11] = 1$
next(11) = 18	mate(18) = 26	vertex(26) = 7	$A[18] = 1 \,.$

To see why we can recover the embedding from the traversal, consider that if we have already correctly embedded the first i edges processed in the traversal, then we can embed the $(i{+}1)$th correctly given its endpoints and its rank in the counter-clockwise order at those vertices. Queries last and prev are superfluous for this task, but they allow traversing the neighbours of a node in clockwise order.

5.1. Implementing the basic queries

We now explain our constant-time implementations of first, next, prev, mate and vertex.

Query first. If $m = 0$ then $\mathsf{first}(v)$ is undefined, which we indicate by returning 0. Otherwise, we first process an edge at v immediately after first arriving at v. Since we identify v with its pre-order rank in our traversal of T and B lacks the opening parenthesis for the root, while first arriving at any vertex v other than the root we write the $(v-1)$th 0 in B and, thus, the $B.\mathsf{select}_0(v-1)$th 1 in A. If v is the root then $\mathsf{first}(v) = 1$ and so, since $\mathsf{select}_x(0) = 0$, this case is also handled by the formula below:

$$\mathsf{first}(v) = \begin{cases} A.\mathsf{select}_1(B.\mathsf{select}_0(v-1)) + 1 & \text{if } m \geq 1 \\ 0 & \text{otherwise.} \end{cases}$$

In our example,

$$\mathsf{first}(5) = A.\mathsf{select}_1(B.\mathsf{select}_0(4)) + 1 = A.\mathsf{select}_1(7) + 1 = 12$$

and indeed the twelfth edge we process, $(5,6)$, is the first one we process at vertex 5. Note that the formula works for nodes with only one edge too.

Query last. The logic of last is similar to that of first; we must locate the closing parenthesis that represents v in T.

$$\mathsf{last}(v) = \begin{cases} A.\mathsf{select}_1(B.\mathsf{match}(B.\mathsf{select}_0(v-1))) & \text{if } m \geq 1 \\ 0 & \text{otherwise.} \end{cases}$$

Query next. If the ith edge we process is the last edge we process at a vertex v then $\mathsf{next}(i)$ is undefined, which we again indicate by returning 0. This is the case when $i = 2m$, or $A[i] = 1$ and $B[A.\mathsf{rank}_1(i)] = 1$. Otherwise, if the ith edge we process is not in T, then $A[i] = 0$, and we process the next edge at v one time step later. Finally, if the ith edge e we process is in T and not the last one we process at v, then we next process an edge at v immediately after returning to v by processing e again at time $\mathsf{mate}(i)$. This is the case when $A[i] = 1$ and $B[A.\mathsf{rank}_1(i)] = 0$. In other words,

$$\mathsf{next}(i) = \begin{cases} i+1 & \text{if } i < 2m \text{ and } A[i] = 0 \\ \mathsf{mate}(i) + 1 & \text{if } i < 2m \text{ and } A[i] = 1 \text{ and } B[A.\mathsf{rank}_1(i)] = 0 \\ 0 & \text{otherwise.} \end{cases}$$

In our example, since $A[12] = 1$, $B[A.\mathsf{rank}_1(12)] = B[8] = 0$, the twelfth edge we process is $(5,6)$ and it is also the fifteenth edge we process,

$$\mathsf{next}(12) = \mathsf{mate}(12) + 1 = 16\,,$$

and indeed the second edge we process at vertex 5 is $(5,7)$.

11

Query prev. The logic for prev is similar to that of next; we only need to consider that, once we move one position backwards, we might arrive at a closing parenthesis. The formula follows.

$$\mathsf{prev}(i) = \begin{cases} i - 1 & \text{if } i > 1 \text{ and } A[i-1] = 0 \\ \mathsf{mate}(i-1) & \text{if } i > 1 \text{ and } A[i-1] = 1 \text{ and } B[A.\mathsf{rank}_1(i-1)] = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Query mate. To implement $\mathsf{mate}(i)$, we check $A[i]$ to determine whether we wrote a bit in B or in B^* while processing the ith edge, and use rank on A to find that bit in the corresponding sequence. We then use match to find the bit encoding the matching parenthesis, and finally use select on A to find where we wrote in A that matching bit. Therefore,

$$\mathsf{mate}(i) = \begin{cases} A.\mathsf{select}_0(B^*.\mathsf{match}(A.\mathsf{rank}_0(i))) & \text{if } A[i] = 0 \\ A.\mathsf{select}_1(B.\mathsf{match}(A.\mathsf{rank}_1(i))) & \text{otherwise.} \end{cases}$$

To compute $\mathsf{mate}(12)$ for our example, since $A[12] = 1$,

$$\begin{aligned} \mathsf{mate}&(12) \\ &= \quad A.\mathsf{select}_1(B.\mathsf{match}(A.\mathsf{rank}_1(12))) \\ &= \quad A.\mathsf{select}_1(B.\mathsf{match}(8)) \\ &= \quad A.\mathsf{select}_1(9) \\ &= \quad 15 \, . \end{aligned}$$

Query vertex. Suppose the ith edge e we process is not in T and we process it at vertex v. If the preceding time we processed an edge in T was the first time we processed that edge, we then wrote a 0 in B, encoding the opening parenthesis for v; otherwise, we then wrote a 1 in B, encoding the closing parenthesis for one of v's children. Now suppose e is in T. If that is the first time we process e, we move to the other endpoint w of e — which is a child of v — and write a 0 in B, encoding the opening parenthesis for w. If it is the second time we process e, then we write a 1 in B, encoding the closing parenthesis for v itself. Therefore,

$$\mathsf{vertex}(i) = \begin{cases} B.\mathsf{rank}_0(A.\mathsf{rank}_1(i)) + 1 \\ \quad \text{if } A[i] = 0 \text{ and } B[A.\mathsf{rank}_1(i)] = 0 \\ B.\mathsf{parent}(B.\mathsf{rank}_0(B.\mathsf{match}(A.\mathsf{rank}_1(i)))) + 1 \\ \quad \text{if } A[i] = 0 \text{ and } B[A.\mathsf{rank}_1(i)] = 1 \\ B.\mathsf{parent}(B.\mathsf{rank}_0(A.\mathsf{rank}_1(i))) + 1 \\ \quad \text{if } A[i] = 1 \text{ and } B[A.\mathsf{rank}_1(i)] = 0 \\ B.\mathsf{rank}_0(B.\mathsf{match}(A.\mathsf{rank}_1(i))) + 1 \\ \quad \text{otherwise.} \end{cases}$$

In our example, since $A[16] = 0$ and $B[A.\mathsf{rank}_1(16)] = B[9] = 1$,

$$\mathsf{vertex}(16)$$

12

Function degree	**Function** listing	**Function** face
Input: node v	**Input:** node v	**Input:** edge e
1 $d = 0$	1 $edg = \texttt{first}(v)$	1 $edg = e$, $fst = \textbf{true}$
2 $edg = \texttt{first}(v)$	2 **while** $edg \neq 0$ **do**	2 **while** $edg \neq e$ **or** fst **do**
3 **while** $edg \neq 0$ **do**	3 $\quad$ $mt = \texttt{mate}(edg)$	3 $\quad$ $fst = \textbf{false}$
4 $\quad$ $edg = \texttt{next}(edg)$	4 $\quad$ **output**	4 $\quad$ $mt = \texttt{mate}(edg)$
5 $\quad$ $d = d + 1$	$\quad\quad$ $\texttt{vertex}(mt)$	5 $\quad$ **output** $\texttt{vertex}(mt)$
6 **return** d	5 $\quad$ $edg = \texttt{next}(edg)$	6 $\quad$ $edg = \texttt{next}(mt)$

$$
\begin{aligned}
&= \quad B.\mathsf{parent}(B.\mathsf{rank}_0(B.\mathsf{match}(A.\mathsf{rank}_1(16)))) + 1 \\
&= \quad B.\mathsf{parent}(B.\mathsf{rank}_0(B.\mathsf{match}(9))) + 1 \\
&= \quad B.\mathsf{parent}(B.\mathsf{rank}_0(8)) + 1 \\
&= \quad B.\mathsf{parent}(5) + 1 \\
&= \quad 5 \, ,
\end{aligned}
$$

and indeed we process the sixteenth edge $(5, 7)$ while visiting 5.

We remind the reader that since B lacks parentheses for the root of T, $B.\mathsf{parent}(5)$ refers to the parent of the fifth vertex in an in-order traversal of T not including the root, i.e., the parent vertex 5 of vertex 6. Adding 1 includes the root in the traversal, so the final answer correctly refers to vertex 5. The lack of parentheses for the root also means that, e.g., $B.\mathsf{parent}(4)$ refers to the parent of vertex 5 and returns 0 because vertex 5 is the root of its own tree in the forest encoded by B, without vertex 1. Adding 1 to that 0 also correctly turns the final value into 1, the in-order rank of the root. Of course, we have the option of prepending and appending bits to A, B and B^* to represent the roots of T and T^*, but that slightly confuses the relationship between the positions of the bits and the time steps at which we process edges.

We also note that, if we do not require that node identifiers are precisely preorder ranks in T, then we can use the positions of their 0 in B as their identifiers. This removes the need for using $B.\mathsf{rank}_0$ and $B.\mathsf{select}_0$ in all the formulas that convert between node identifiers and positions in T.

5.2. More complex queries

We can define more complex queries on top of the basic ones. For example, we give the pseudocode of three queries: $\mathsf{degree}(v)$ returns the number of neighbours of vertex v; $\mathsf{listing}(v)$ returns the list of neighbours of vertex v, in counter-clockwise order; $\mathsf{face}(e)$ returns the list of vertices, in clockwise order, of one of the face where the edge e belongs. We also support the other order (clockwise or counter-clockwise, or the other face where e belongs) by using last and prev instead of first and next.

Queries $\texttt{listing}(v)$ and $\texttt{face}(e)$ are implemented in optimal time, that is, $\mathcal{O}(1)$ per returned element. Instead, $\texttt{degree}(v)$ requires time $\mathcal{O}(\texttt{degree}(v))$. We can also determine $\mathsf{neighbour}(u, v)$, that is, whether two vertices u and v are neighbours, by listing the neighbours of each in interleaved form, in time

$\mathcal{O}(\min(\mathsf{degree}(u), \mathsf{degree}(v)))$. These times are not so satisfactory compared with the $\mathcal{O}(1)$ achieved by other representations [8, 9] to compute $\mathsf{neighbour}(u, v)$ and $\mathsf{degree}(v)$.

For $\mathsf{degree}(v)$, we can get arbitrarily close to constant time by adding $o(m)$ further bits to our representation, that is, we can solve the query in time $\mathcal{O}(f(m))$ for any given function $f(m) \in \omega(1)$. To do this, we store a bitvector $D[1..n]$ marking with 1s the (at most) $m/f(m) = o(m)$ vertices with degree at least $f(m)$, which takes $nH(m/(nf(m)))+o(n) = \mathcal{O}((m/f(m)) \log(nf(m)/m))+ o(n) = o(m)$ bits by using a sparse bitvector representation [29] (recall that G is connected, so $m \geq n - 1$). We also store a second bitvector $E[1..2m]$ where we append, for each $D[v] = 1$, $\mathsf{degree}(v) - 1$ copies of 0s followed by a 1. Since E has $m/f(m)$ 1s, it can also be stored as a sparse bitvector using $\mathcal{O}((m/f(m)) \log f(m)) + o(m) = o(m)$ bits. Therefore, if $D[v] = 1$, its degree is obtained in constant time with $\mathsf{select}_1(E, r) - \mathsf{select}_1(E, r - 1)$, where $r = \mathsf{rank}_1(D, v)$. If, instead, $D[v] = 0$, then we know that $\mathsf{degree}(v) < f(m)$ and thus we apply the procedure that sequentially counts the neighbours, in time $\mathcal{O}(f(m))$.

We can use a similar idea, albeit more complex, to answer $\mathsf{neighbour}(u, v)$ queries in time $\mathcal{O}(f(m))$, for any $f(m) = \omega(\log m)$. We consider the graph induced by the $\mathcal{O}(m/f(m)) = o(m/\log m)$ nodes with degree $f(m)$ or higher and eliminate multi-edges and self-loops. The resulting graph G' is simple and still planar, so it has average degree less than or equal to 6 and thus $o(m/\log m)$ edges. We represent G' in classical adjacency-list form, with the nodes inside each list sorted by increasing node identifier. This requires $o(m)$ bits in total. To answer $\mathsf{neighbour}(u, v)$ on G, we check whether either u or v is low-degree (assuming we mark low-degree nodes in a bitvector D' analogous to D) and, if so, list its neighbours in $\mathcal{O}(f(m))$ time. If not, we translate nodes u and v to their corresponding nodes $u' = \mathsf{rank}_1(D', u)$ and $v' = \mathsf{rank}_1(D', v)$ in G' and query G' in time $o(f(m))$. To solve $\mathsf{neighbour}(u, v)$ in G', we find the position of the adjacency list of u in G' by computing $u' = \mathsf{rank}_1(D', u)$, then use binary search for v in the list of u in time $\mathcal{O}(\log m) = o(f(m))$.

The following theorem summarizes the results of this section.

Theorem 1. *We can store a given planar embedding of a connected planar graph G with m edges in $4m + o(m)$ bits such that later, given a vertex v, we can list the edges incident to v in clockwise or counter-clockwise order, even if we are given a particular starting edge incident to v, using constant time per edge. We can also traverse the edges limiting a face in constant time per edge. Further, we can find a vertex's degree in $\mathcal{O}(f(m))$ time for any given function $f(m) \in \omega(1)$, and determine whether two vertices are neighbours in $\mathcal{O}(f(m))$ time for any given function $f(m) \in \omega(\log m)$.*

5.3. Reducing space on simple planar graphs

Chiang *et al.* [9] use $2m + 3n + o(m)$ bits to represent planar graphs without self-loops, which can be more than the $4m + o(m)$ bits used in our representation.

However, if G is simple (i.e., has no loops nor multiple edges), their representation requires only $2m + 2n + o(m) \leq 4m + o(m)$ bits. We remind the reader that this representation can handle any simple planar graph, but does not always respect the given embedding, so they cannot represent arbitrary embeddings.

We show that, if there are no self-loops, our representation can use less than $4m + o(m)$ bits, by exploiting some redundancy in our representation and without changing the main scheme. Assume we represent a single sequence $S[1..2m]$ over an alphabet of four symbols, $\Sigma = \{(,), [,]\}$, that replaces A, B, and B^*. That is, the parentheses are the 0s and 1s in B, the brackets are the 0s and 1s and B^*, and A corresponds to whether the symbols are parentheses or brackets. In our running example, the sequence is

$$S[1..2m] \quad = \quad [\,(\,(\,[\,]\,)\,(\,[\,)\,[\,)\,(\,(\,]\,[\,]\,)\,[\,)\,(\,]\,(\,]\,]\,[\,)\,]\,)\,[\,].$$

The zeroth-order entropy of S is defined as $H_0(S) = \sum_{c \in \Sigma} \frac{m_c}{2m} \log \frac{2m}{m_c}$, where c occurs m_c times in S. The kth-order entropy, for any $k > 0$, is defined as $H_k(S) = \sum_{C \in \Sigma^k} \frac{|S_C|}{2m} H_0(S_C)$, where S_C is the string formed by the symbols that follow the context C in S (assume S is circular for simplicity, so that $S[1]$ follows $S[2m]$).

Ferragina and Venturini [37] show how to store a string S within $|S|H_k(S) + o(|S| \log |\Sigma|)$ bits, for any $k = o(\log_{|\Sigma|} |S|)$, so that any substring of length $\mathcal{O}(\log |S|)$ can be extracted in constant time. We use their result to store S in $2mH_1(S) + o(m)$ bits. Instead of a structure on parentheses on bitvector B and another on bitvector B^*, we build both parentheses structures on top of sequence S. Both are similar to the original $o(m)$-bit structure of Navarro and Sadakane [31], only that the structure built to navigate parentheses ignores the bracket symbols, and vice versa (a similar arrangement is described by Navarro [23, pp. 311–315]). The only changes are that each symbol uses 2 bits instead of 1, that there are two symbols that do not change the "excess" count (number of opening minus closing parentheses up to some position), and that in order to extract a chunk of $\Theta(\log m)$ symbols, we use the extraction method of Ferragina and Venturini [37]. A rank/select functionality on top of A is also easily provided on top of S, by using the same $o(m)$-bit structures [27, 28] and interpreting both parentheses as 1s and both brackets as 0s. Therefore, with $o(m)$ further bits, we provide the necessary functionality on top of the $H_1(S)$ bits needed to encode S.

This entropy gives precisely 2 bits per symbol (and thus $4m$ bits in total) for general planar embeddings, but if there are no self-loops, then the substring "[ ]" cannot appear in S (other longer strings cannot appear either, but we would need a higher-entropy model to capture them). An upper bound to the first-order entropy when this substring is forbidden is obtained by noticing that we can have only 3 symbols, instead of 4, following an opening bracket; therefore we can encode S using $n \log 4 + n \log 4 + (m-n) \log 3 + (m-n) \log 4 = m \log 12 + n \log(4/3) \approx 3.58m + 0.42n$. This is still $4m$ in the worst case. To obtain a nontrivial bound in terms of m, we calculate the exact first-order entropy of S when substring "[ ]" is forbidden.

Let us use the names $op = ($, $cp =)$, $ob = [$, and $cb =]$. Let us call x_y number of symbols y following a symbol x in S; for example op_{ob} is the number of opening brackets following opening parentheses, that is, the number of occurrences of substring "([" in S. It must then hold that $\sum op_* = \sum cp_* = n$ and $\sum ob_* = \sum cb_* = m - n$. It also holds $\sum *_{op} = \sum *_{cp} = n$ and $\sum *_{ob} = \sum *_{cb} = m - n$. The system of restrictions must be satisfied while maximizing

$$2mH_1(S) = nH(op_*) + nH(cp_*) + (m - n)H(ob_*) + (m - n)H(cb_*),$$

where $H(x_1, \ldots, x_4) = \sum \frac{x_i}{x} \log \frac{x}{x_i}$ and $x = \sum x_i$. Forbidding self-loops implies the additional restriction $ob_{cb} = 0$.

We solve the optimization problem with a combination of algebraic and numeric computation, using Maple and C, up to 4 significant digits. We find that the entropy is maximized at a value slightly below $3.8m$.[1] Therefore, the resulting space with no self-loops and using the described compressed representation can be bounded by $3.8m + o(m)$ bits. Simple graphs have no self-loops and no multiple-edges, but this second restriction translates into longer forbidden substrings, whose effect is harder to analyze.

We remind the reader that the representation of Keeler and Westbrook [5], on the other hand, achieves $m \log 12 \approx 3.58m$ bits when no self-loops (or, alternatively, no degree-one nodes) are permitted, yet it does not support queries. When neither self-loops nor degree-one nodes are permitted, they reach $3m$ bits. In this case, both "[ ]" and "( )" are forbidden strings. While we have not been able to compute the exact first-order entropy in this case, this must be at most $n \log 3 + n \log 4 + (m - n) \log 3 + (m - n) \log 4 = m \log 12 \approx 3.58m$, which is obtained by using $\log 4$ bits to encode the symbol that follows a closing bracket or parenthesis, and $\log 3$ bits to encode the symbol that follows an opening bracket or parenthesis.

We note that these space improvements can also be applied on top of the representation of Chiang *et al.* [9] since, when encoding a simple graph, the difference between both representations is that they use a particular spanning tree (which may also force a particular embedding).

5.4. Unconnected planar graphs

Our representation can be easily extended to unconnected planar graphs, because our parentheses representations can immediately be extended to handle forests instead of just individual trees. To handle an unconnected planar graph, we first find all the connected components of the graph and then compute an arbitrary spanning tree for each connected component. Then, we construct the binary sequences: the sequence B will represent the *forest* of the spanning trees, concatenating all the balanced-parentheses representations; the sequence B^* will represent the complementary spanning tree of the dual of the graph.

---

[1]The maximum is $3.7999m$, found for $m = 1.731n$, $op_{op} = cp_{op} = 0.2683n$, $ob_{op} = 0.2679n$, $cb_{op} = 0.1955n$, $op_{cp} = cp_{cp} = 0.2677n$, $ob_{cp} = 0.2673n$, $cb_{cp} = 0.1974n$, $op_{ob} = cp_{ob} = 0.1961n$, $ob_{ob} = 0.1958n$, $cb_{ob} = 0.1429n$, $op_{cb} = cp_{cb} = 0.2679n$, $ob_{cb} = 0, cb_{cb} = 0.1952n$.

Finally, sequence A indicates the interleaving of the sequences B and B^*. We visit the connected components in arbitrary order.

Note that, in the case of connected planar graphs, our navigation queries and the fact that the first edge we list is adjacent to the external face, are sufficient to recover the embedding. This is not the case if the graph has $k > 1$ connected components. Concretely, some components may be embedded inside faces of other components, whereas our arrangement assumes that all the connected components lie on the outer face (our navigation queries cannot distinguish between those cases).

To recover the embedding we might add $k - 1$ edges to the spanning tree, so that all the connected components lying in a single face are threaded through a node in their frontier, and the first one is linked to a node on the containing face. Therefore the total length of A will be $2(m + k - 1)$ and the length of B will be $2(n + k - 2)$. The fake edges will be marked in a bitvector $K[1..n + k - 2]$ indexed by preorder value. Since K contains $k - 1$ 1s, it can be encoded in $k \log(n/k) + \mathcal{O}(k)$ bits [38]. Since $n \leq m + k$, the space of the whole structure can be written in terms of m and k as $4m + k \log(m/k) + \mathcal{O}(k) + o(m)$ bits.

The $k \log(m/k) + \mathcal{O}(k)$ or $k \log(n/k) + \mathcal{O}(k)$ bits to describe the embedding are asymptotically optimal: consider a chain of t triangles (delimited with $m = 2t + 1$ edges) and $k - 1$ isolated nodes (so there are k connected components in total) to represent all the ways to distribute $k - 1$ balls into t bins. This requires $\log \binom{k+t-2}{k-1} = k \log(t/k) + \mathcal{O}(k) = k \log(m/k) + \mathcal{O}(k)$ bits with any encoding. This is also $k \log(n/k) + \mathcal{O}(k)$ bits, since this graph has $n = 2t + k$ nodes.

We use K to avoid listing fake edges in any of the traversal operations. The fake edges increase the degree of a node by a constant factor: a node may have one fake edge per face it participates in, which at most doubles its degree. Further, a node in the frontier of its component may have two extra fake edges threading it with other connected components. Therefore, the time complexity of the navigation operations is not affected.

The fake edges may, in addition, be useful for a more ambitious `face` operation that takes into account the actual embedding, where a face is surrounded by a sequence of edges but is also limited by the frontier edges of the connected components it has inside. To find all those edges, we also traverse the fake edges in the `face` traversal, yet without listing them. The fake edges will lead us to the other connected components that are contained and/or surround the face we are listing.

6. Parallel construction

In this section we discuss the parallel construction of our extension of Turán's representation. Since the representation is based on spanning trees and tree traversals, we can borrow ideas of well-known parallel algorithms, such as parallel Euler Tour traversal or parallel computation of spanning trees.

We assume that a tree T is represented with adjacency lists. Such representation consists of an array of nodes $V_T[1..n]$, and an array of edges $E_T[1..2n - 2]$.

Each node $v \in V_T$ stores two indices in E_T, *v.first* and *v.last*, delimiting the adjacency list of v, which starts with v's parent edge (except the root) and is sorted counter-clockwise around v. The number of children of v is then $v.last - v.first$ (plus 1 for the root). Each edge $e \in E_T$ has three fields: *e.src* and *e.tgt* are the positions in V_T of its source and target vertices, and *e.mat* is the position in E_T of the mate edge e' of e, where $e'.src = e.tgt$ and $e'.tgt = e.src$. Our representation of graphs is similar, with the exception that the concept of *parent* of a vertex is not valid in graphs; therefore the first edge in the adjacency list of a vertex v cannot be interpreted as v's parent edge.

6.1. Parallel construction of compact planar embeddings

We will first assume that the input consists of a connected planar graph embedding $G = (V_G, E_G)$ and a spanning tree $T = (V_T, E_T)$ of G, together with an array C that stores the number of edges of $G \backslash T$ between any two consecutive edges in T, in counter-clockwise order. In Section 6.3 we will explain how to obtain T and C in parallel.

With the spanning tree, we construct the bitvectors A, B, and B^* by performing an Euler Tour over T. During the tour, by writing a 0 for each forward (parent to child) edge and a 1 for each backward (child to parent) edge, we obtain the bitvector B. By reading in C the number of edges of $G \setminus T$ between two consecutive edges of T, representing these edges with 0s and the edges of T with 1s, we obtain the bitvector A. Finally, by using the previous Euler Tour and the array C we can obtain the bitvector B^*, by finding out which is the first (0) and which is the second (1) occurrence of each edge.

Algorithm 1 gives the detailed pseudocode. It works in the following steps:

1. In lines 1–4, it initializes the output bitvectors (A and B^* are set to 0s) and creates an auxiliar array LE that is used to store the traversal of the tree following the Euler Tour. Each entry of LE represents one traversed edge of T and stores four fields: *value* is 0 or 1 depending on whether the edge is a forward or a backward edge, respectively; *succ* is the index in LE of the next edge in the Euler tour; *rankA* is the rank of the edge in A; and *rankB* is the rank of the edge in B.

2. In lines 5–19, the algorithm traverses T to create the Euler Tour. For each edge $e_j \in E_T$, *rankA* is set to $C[E_T[j].mat] + 1$ and *rankB* to 1 (lines 6–7). Those ranks will be used later to compute the final positions of the edges in A, B, and B^*. For each forward edge, a 0 is written in the corresponding *value* field and the *succ* field is connected to the next edge in the Euler Tour. For backward edges the procedure is similar. Note that all the edges in the adjacency list of a node of T are forward edges, except (for non-root nodes) the first one, which is the parent edge.

3. Line 20 computes the final ranks in A and B using a parallel list ranking algorithm that adds up the weights from the beginning of the list to each element. The weights are stored in the fields *rankA* and *rankB* of LE. We use the list ranking algorithm of Helman and Jájá [39].

Algorithm 1: Parallel compact planar embedding algorithm.

Input : A planar graph embedding $G = (V_G, E_G)$, a spanning tree $T = (V_T, E_T)$
of G, an array C of size $|E_T|$, and the starting vertex *init*.
Output: Bitvectors A, B and B^* induced by G and T.

---

1 A = a bitvector of length $|E_G|$ initialized with 0s
2 B = a bitvector of length $|E_T| - 2$
3 B^* = a bitvector of length $|E_G| - |E_T| + 2$ initialized with 0s
4 LE = an array of length $|E_T|$
5   **parfor** $j = 1$ **to** $|E_T|$ **do**
6 $LE[j].rankA = C[E_T[j].mat] + 1$
7 $LE[j].rankB = 1$
8   **if** $E_T[j].src = init$ **or** $E_T[j].src.first \neq j$ **then** *// forward edge*
9 $LE[j].value = 0$ *// opening parenthesis*
10   **if** $E_T[j].tgt.first = E_T[j].tgt.last$ **then** *// target is a leaf*
11 $LE[j].succ = E_T[j].mat$
12   **else** *// target has children*
13 $LE[j].succ = E_T[j].tgt.first + 1$
14   **else** *// backward edge*
15 $LE[j].value = 1$ *// closing parenthesis*
16   **if** $E_T[j].mat = E_T[j].tgt.last$ **then** *// j was the last edge of target, return*
17 $LE[j].succ = E_T[j].tgt.first$
18   **else** *// continue with next edge from target*
19 $LE[j].succ = E_T[j].mat + 1$

20 `parallelListRanking`(LE)
21 **parfor** $j = 1$ **to** $|E_T|$ **do**
22 $A[LE[j].rankA] = 1$
23 $B[LE[j].rankB] = LE[j].value$
24 D_{pos}, D_{edge} = arrays of length $|E_G|$ and $|E_G| - |E_T| + 2$, respectively
25 **parfor** $j = 1$ **to** $|E_T|$ **do**
26 $p = LE[j].rankA - LE[j].rankB$
27 $base = ref(E_T[j].mat)$
28 $delta = p - base - 1$
29   **parfor** $k = base + 1$ **to** $base + C[E_T[j].mat]$ **do**
30 $D_{pos}[k] = k + delta$
31 $D_{edge}[k + delta] = k$
32 **parfor** $j = 1$ **to** $|E_G| - |E_T| + 2$ **do**
33 $mat = E_G[D_{edge}[j]].mat$
34   **if** $j > D_{pos}[mat]$ **then**
35 $B^*[j] = 1$
36 `createRankSelect`(A), `createBP`(B), `createBP`(B^*)

---

4. Bitvectors A and B are written in lines 21–23. Since initially all the elements of A are 0s, it is enough to set to 1 all the elements in the *rankA* fields. For B, the algorithm copies the content of field *value* at position *rankB*, for all the elements in LE.

5. The algorithm now computes the position of each edge of $G \setminus T$ in B^*. That information is implicit in the fields *rankA* and *rankB* of LE (line 26), once the list ranking of step 3 is carried out. For each edge $e \in E_T$, the algorithm computes the positions in B^* of the edges of $G \setminus T$ that follow, in counter-clockwise order, the mate edge of e (lines 27–31). The algorithm

19

uses two auxiliary arrays, D_{pos} and D_{edge}. Let edge $E_G[j]$ belong to $G \setminus T$. Then $D_{pos}[j]$ stores the position of the edge in B^*. The array D_{edge} is the inverse of D_{pos}: $D_{edge}[i]$ is the position of the i-th edge of B^* in E_G. This step uses function $ref(e)$, which maps the position e of an edge in E_T to its position in E_G. This is naturally returned by the spanning tree construction, which gives the identity in G of the edges selected for T.

6. In lines 32–35, the algorithm computes whether the edges stored in D_{pos} are forward or backward edges. For each edge e in $G \setminus T$, it compares the positions in B^* of e and its mate. If the position of e is greater, then e is a backward edge and, therefore, is represented with a 1.

7. Finally, in line 36 the structures to support operations rank, select, match, and parent are constructed. For the bitvector A, the parallel algorithm of Labeit *et al.* [40] (`createRankSelect`) is used. For B and B^* the parallel algorithm of Ferres *et al.* [41] for balanced parenthesis sequences (`createBP`) is used.

We have omitted some implementation details for simplicity. For example, the pseudocode uses **parfor** throughout, whereas the implementation uses the threads in a more controlled manner. Line 29, in particular, is more efficiently done in sequential form. We have also omitted some space optimizations, such as the reuse of some fields instead of allocating new arrays.

Analysis. Step 1 initializes the arrays, which requires $T_1 = \mathcal{O}(m)$ work and $T_\infty = \mathcal{O}(\log m)$ span (due to the overhead of the implicit **parfor**). In step 2, the algorithm traverses the edges of T, performing an independent computation on each edge. Therefore, with the overhead of the **parfor** loop, we obtain $T_1 = \mathcal{O}(n)$ and $T_\infty = \mathcal{O}(\log n)$ time. Step 3 uses a parallel list ranking algorithm [39] over n elements, which has complexities $T_1 = \mathcal{O}(n)$ and $T_\infty = \mathcal{O}(\log n)$. Step 4 assigns the values to A and B independently for each entry, thus we have again $T_1 = \mathcal{O}(n)$ and $T_\infty = \mathcal{O}(\log n)$. In step 5, the algorithm traverses all the edges in $G \setminus T$. Since the loop in line 29 is also processed in parallel, we obtain $T_1 = \mathcal{O}(m - n)$ and $T_\infty = \mathcal{O}(\log(m - n))$. Similarly to step 4, in step 6 the algorithm sets the entries of bitvector B^*, which can be done independently for each entry, obtaining times $T_1 = \mathcal{O}(m - n)$ and $T_\infty = \mathcal{O}(\log(m - n))$. Finally, step 7 builds the rank/select structures in times $T_1 = \mathcal{O}(m)$ and $T_\infty = \mathcal{O}(\log m)$ [40]. The construction of the structures supporting match and parent over balanced parentheses is constructed in times $T_1 = \mathcal{O}(m)$ and $T_\infty = \mathcal{O}(\log m)$ [41].

In addition to the size of the compact data structure, our algorithm uses $\mathcal{O}(m \log m)$ bits for the arrays LE, D_{pos} and D_{edge}. As said, the constant is kept low in practice by reusing fields. Notice that the memory consumption is independent of the number of threads.

6.2. Structures for degree and neighbour queries

Before discussing how to construct the structures to speed up degree(v) and neighbour(u, v) queries, let us discuss the parallel construction of the sparse bitvector of Raman *et al.* [29]. Let ℓ be the length of the sparse bitvector. Their

representation divides the bitvector into blocks of length $b = (\log \ell)/2$. The ith block is described as a pair (c_i, o_i), where c_i corresponds to the number of 1s inside the block, also known as the *class* of the block, and o_i corresponds to its *offset*, an identifier among all the different blocks sharing the same class. Thus, the bitvector is represented as two arrays, $C[1..\lceil \ell/b \rceil]$ and $O[1..\lceil \ell/b \rceil]$, where $C[i] = c_i$ and $O[i] = o_i$. We can compute in parallel each entry of the arrays C and O independently, using linear time on each block [23, Sec. 4.1]. Thus, we have $\mathcal{O}(\ell)$ work and $\mathcal{O}(\log(\ell/b) + b) = \mathcal{O}(\log \ell)$ span. In order to reduce the space consumption of the arrays C and O, the entries of the arrays are packed into the bits of consecutive machine words. Notice that the size of the elements of C is fixed, $\lceil \log(b+1) \rceil$ bits, whereas the size of those of O, $\lceil \log o_i \rceil$ bits, is variable. To pack the entries of O in parallel, we need to compute an array $P[1..\lceil \ell/b \rceil]$ pointing to the starting position of each element in O. Array P is computed with a parallel parallel prefix sum over the values $\lceil \log o_i \rceil$. This takes linear work and logarithmic span [40], and then we can write each value o_i to its packed position in parallel. The array P is retained to provide efficient access to O. To reduce its space to $o(\ell)$ bits, only the entries of the form $P[i \cdot \log n]$ are stored in absolute form, whereas the others are stored as differences from the preceding multiple of $\log n$, using $\mathcal{O}(\log \log n)$ bits. This space reduction is easily computed in parallel within the same time bounds. Once the data structures C, O, and P, using $\ell H + o(\ell)$ bits, are built, we can access in constant time any chunk of $\mathcal{O}(\log \ell)$ bits from the bitvector by using tables [29]. Therefore, we can provide rank and select functionality by building the classical $o(\ell)$-bit data structures on top of the bitvector, in parallel [40]. In total, we use $\mathcal{O}(\ell)$ work and $\mathcal{O}(\log \ell)$ span.

The structures to support degree(v) can then be constructed in parallel as follows: First, we construct the bitvector D by checking all the vertices with degree at least $f(m)$. Remember that the degree of a vertex v can be computed in constant time with $v.last - v.first$. Since the degree of each vertex can be obtained independently, we can do this in parallel with $\mathcal{O}(m)$ work and $\mathcal{O}(\log m)$ span. Then, we construct the bitvector E by writing in unary the degree of each high-degree vertex. To do that, we perform a parallel prefix sum over all the degrees of high-degree vertices. The prefix sum returns the positions where we have to write a 1 in E. Thus, we construct E with $\mathcal{O}(m)$ work and $\mathcal{O}(\log m)$ span. Finally, we construct the compact representation of D and E in $\mathcal{O}(m)$ work and $\mathcal{O}(\log m)$ span, using the sparse bitvectors of Raman *et al.* [29].

For the neighbour(u, v) query, we must contract the original graph G into a smaller graph $G' = (V', E')$, induced by all the vertices with degree at least $f(m)$. To build G' efficiently in parallel we do as follows. We first compute $D'[1..n]$ similarly to D. We then fill two arrays $X[1..n]$ and $Y[1..2m]$, so that $X[i] = D'[i]$; and $Y[j] = 1$ if $D'[E_G[j].src] = 1$ and $D'[E_G[j].tgt] = 1$, and $Y[j] = 0$ otherwise. Next, we perform a parallel prefix sum over X, so that $X[i]$ is the name of node i in G' (if $D'[i] = 1$). We also perform a parallel prefix sum on Y, so as to write contiguously in array E' the mapped edge targets, $E'[j'] = X[E_G[j].tgt]$ for those entries j where $Y[j] = 1$, where $j' = \sum_{k=1}^{j} Y[k]$. For each such edge, we also check if it is the first with this $X[E_G[j].src]$ value,

and if so, we record that j' is the start of the adjacency list of node $X[E_G[j].src]$, in an array $V'[X[E_G[j].src]] = j'$.

Thus V' and E' are an adjacency list representation of G', built with $\mathcal{O}(m)$ work and $\mathcal{O}(\log m)$ span. Instead of sorting the adjacency lists, however, we build a wavelet tree representation on E' [40]. This supports the operation rank generalized to sequences, and therefore we use that high-degree nodes u and v of G are connected if and only if $X[v]$ is mentioned in the adjacency list of $X[u]$, that is, $E'.\mathsf{rank}_{X[v]}(V'[X[u]+1]-1) - E'.\mathsf{rank}_{X[v]}(V'[X[u]]-1) > 0$. The generalized rank operation takes time $\mathcal{O}(\log |V'|)$ and the wavelet tree is built with $\mathcal{O}(|E'|) = o(m/\log m)$ work and $\mathcal{O}(\log^2 |E'|) = \mathcal{O}(\log^2 m)$ span.

Lemma 2. *Given a connected planar graph embedding G with m edges and a spanning tree of G, we can compute in parallel a compact representation of G, using $4m + o(m)$ bits and supporting the navigational operations described in Section 5, in $\mathcal{O}(m)$ work and $\mathcal{O}(\log m)$ span ($\mathcal{O}(\log^2 m)$ span if operation neighbour is supported), using $\mathcal{O}(m \log m)$ bits of additional memory.*

6.3. Parallel computation of spanning trees

In this section we discuss the parallel computation of the spanning tree $T = (V_T, E_T)$ and the array C used in Section 6.1.

Generating a rooted (or a directed) spanning tree turns out to be a difficult to parallelize problem. Even if it seems to be easier on planar embeddings, we do not know of good worst-case results on the DyM model. We discuss practical solutions later.

Such a spanning tree algorithm returns an array of parent references for each vertex. With this array of references, we can construct the corresponding adjacency list representation of the spanning tree. To do that, we mark with a 1 each edge E_G that belongs to E_T and with a 0 the rest of the edges. Using a parallel prefix sum algorithm over E_G, we compute the position of all the marked edges of E_G in E_T. The *first* and *last* fields of each node in the spanning tree are computed similarly. As a byproduct of the computation of E_T, we can compute the array C, which stores the number of edges of $G \setminus T$ between two consecutive edges in T, in counter-clockwise order. This can be done by using the marks in the edges, counting the number of 0s between two consecutive 1s. Note that the starting vertex for the spanning tree must be in the outer face of G, to meet the description of the compact data structure for planar embeddings. Overall, we require times $T_1 = \mathcal{O}(m)$ and $T_\infty = \mathcal{O}(\log m)$ once the spanning tree is built, which is the complexity of the variants of the parallel prefix sum algorithm we employ. By combining the results with Lemma 2, we have the main result on construction.

Theorem 2. *The compact representation introduced in Theorem 1 of a connected planar graph embedding G with m edges can be constructed under the Dynamic Multithreaded parallel model with $\mathcal{O}(m + \mathrm{spw})$ work and $\mathcal{O}(\log m + \mathrm{sps})$ span ($\mathcal{O}(\log^2 m + \mathrm{sps})$ span if operation neighbour is supported), where spw and sps are the work and span, respectively, of any rooted spanning tree algorithm on planar embeddings.*

In practice. The generation of a spanning tree is also difficult to parallelize in practice. Bader and Cong [42] mention that "the spanning tree problem is notoriously hard for any parallel implementation to achieve reasonable speedup", and propose an algorithm that is shown to perform well in practice. This is the one we use in our implementation.

Their algorithm works as follows. Given a starting vertex of the graph G with n vertices and m edges, the algorithm computes sequentially a spanning tree of size $\mathcal{O}(p)$, called *stub spanning tree*, where p is the number of available threads. Then, the leaves of the stub spanning tree are evenly assigned to the p threads as starting vertices. Each thread traverses G, using its starting vertices, constructing spanning trees with a DFS traversal using a stack. For each vertex, a reference to its parent is assigned. Since a vertex can be visited by several threads, the assigment of the parent of the vertex may genarate a *race condition*. However, since the parent assigned by any thread already belongs to a spanning tree, any assignment will generate a correct tree. Thus, the race condition is benign. Once a thread has no more vertices on its stack, it tries to steal vertices from the stack of another thread by using the work-stealing algorithm. Since the spanning trees generated by all the threads are connected to the stub spanning tree, the union of all the spanning tree generates a spanning tree of G.

They analyze their algorithm in expectation on random graphs, obtaining $\mathcal{O}(m/p)$ time when $p \ll m$, but general random graphs have a very small diameter. The diameter seems to be a lower bound for the span of their algorithm, and this is $\Theta(n^{1/4})$ on random planar graphs [43]. Also, their best possible time is $\mathcal{O}(\sqrt{m})$, achieved when using $p = \sqrt{m}$ processors. Despite its analysis, the algorithm of Bader and Cong has a good practical behavior and its implementation is simple.

PRAM model. We can also analyze our algorithm under the PRAM model. Algorithm 1 is easily translated into the EREW model, reaching $\mathcal{O}(m/\log m)$ processors and $\mathcal{O}(\log m)$ time, dominated by the parallel list ranking of line 20, the expansion from n to m processors in line 29, and the construction of succinct structures in line 36. The construction in Section 6.2, of the structures that speed up `degree` and `neighbour` queries, is also easily carried out in the EREW model within those bounds, except for the sorting of the edges of G'. This can be done in $\mathcal{O}(\log m)$ time with $\mathcal{O}(m)$ processors in the EREW model [44], and in $\mathcal{O}(\log^2 m)$ time with $\mathcal{O}(m/\log m)$ processors in the CREW model [45]. The postprocessing we have described in this section, once the spanning tree is built, also runs in $\mathcal{O}(\log m)$ time and $\mathcal{O}(m/\log m)$ EREW processors.

The most costly part of the process is likely to be the construction of the spanning tree. The best PRAM results we know of are $\mathcal{O}(\log^2 m \log^* m)$ time and $\mathcal{O}(m)$ processors in the EREW model [46], $\mathcal{O}(\log^2 m)$ time and $\mathcal{O}(m/\log m)$ processors in the arbitrary CRCW model [47], and $\mathcal{O}(\log m)$ time and $\mathcal{O}(m^3)$ processors in the same model [48].

Theorem 3. *The compact representation introduced in Theorem 1 of a connected planar graph embedding G with m edges can be constructed under the*

PRAM EREW model with $\mathcal{O}(m)$ processors and $\mathcal{O}(\log^2 m \log^* m)$ time, and under the PRAM arbitrary CRCW model with $\mathcal{O}(m/\log m)$ processors and $\mathcal{O}(\log^2 m)$ time, or $\mathcal{O}(m^3)$ processors and $\mathcal{O}(\log m)$ time.

In the case of unconnected planar graphs, we can use the algorithm of Shun *et al.* [49] to find the connected components, and then proceed as in Section 5.4 to represent them. This algorithm, in the PRAM arbitrary CRCW model, requires $\mathcal{O}(n)$ expected work and $\mathcal{O}(\log^2 n \log^* n)$ span with high probability.[2]

Corollary 1. *The parallel construction of Theorem 3 can be applied to unconnected planar graphs of n nodes, in the PRAM arbitrary CRCW model, with $\mathcal{O}(n/\log n)$ processors in expectation, and $\mathcal{O}(\log^2 n \log^* n)$ time with high probability.*

7. Experiments

We implemented the data structure construction and queries in C and compiled it using GCC 5.4. For the parallel construction we used Cilk Plus extension, an implementation of the DyM model. We build only the basic structures, excluding those to speed up operations degree and neighbour. The code and data needed to replicate our results are available at `http://www.dcc.uchile.cl/~jfuentess/pemb/`.

The experiments were carried out on a NUMA machine with two NUMA nodes (Machine A). Each NUMA node includes a 14-core Intel® Xeon® CPU (E5-2695) processor clocked at 2.3GHz. The machine runs Linux 4.4.0-83-generic, in 64-bit mode. The machine has per-core L1 and L2 caches of sizes 64KB and 256KB, respectively and a per-processor shared L3 cache of 35MB, with a 768GB DDR3 RAM memory (384GB per NUMA node), clocked at 1867MHz. Hyperthreading was enabled, giving a total of 28 logical cores per NUMA node.

To show the impact of a machine with limited memory in the running time of queries, additional experiments were carried out on a machine (Machine B) with Intel® Core™ i7-7500U CPU, with four physical cores running at 2.70GHz. The computer runs Linux 4.8.0-53-generic, in 64-bit mode. This machine has per-core L1 and L2 caches of sizes 64KB and 256KB, respectively, and a shared L3 cache of 4MB, with a 8GB DDR4 RAM. To reduce the size of the available physical memory, we set the `mem` parameter of the Linux Kernel to `mem=600MB`. The machine has a hard disk with a capacity of 1TB, 5400 RPM and a SATA connection. The results using this machine are reported only in the last part of Section 7.3.

7.1. Datasets

Our experiments ran on real and artificial datasets with different numbers of nodes. The datasets are shown in Table 1. For the artificial datasets we

---

[2]See the note after their proof of Theorem 1.

	Dataset	Vertices (n)	Edges (m)
1	wc	2,243,467	6,730,395
2	pe5M	5,000,000	14,999,983
3	pe10M	10,000,000	29,999,979
4	pe15M	15,000,000	44,999,983
5	pe20M	20,000,000	59,999,975
6	pe25M	25,000,000	74,999,979

Table 1: Datasets used in our experiments.

generated points (x, y) with the function `rnorm` of `R`.[3] The real dataset, `wc`, corresponds to the coordinates of $2,243,467$ unique cities in the world.[4] From those real or generated points, we obtained a *Delaunay Triangulation* using *Triangle*, a software for the generation of meshes and triangulations[5]. Finally, we generated planar embeddings from the Delaunay triangulations with the *Edge Addition Planarity Suite*[6]. The minimum and maximum degrees of the dataset `wc` were 3 and 36, respectively. For the rest of the datasets, the minimum degree was 3 and the maximum degree was 16. We also tested our construction algorithm on several sparsified versions of one of the triangulations, as described in Section 7.4. We do not see any reason for the time per edge of queries to change with sparsification.

7.2. Space usage

There are no other implemented compact representations of planar embeddings. In this subsection we aim to show that representations designed for other kinds of graphs are indeed not competitive for this graph family. We compare our compact representation with four solutions designed to compress Web graphs, social networks and planar graphs [50, 51, 52, 53], and with one parallel framework for processing general graphs in compressed form [54]. The three solutions for Web graphs and social networks require reordering the vertices of the graph. The solution of Apostolico and Drovandi [50] (AD) enumerates the vertices through a BFS traversal of the graph. The reordering induces two

---

[3]The `rnorm` function generates random numbers with normal distribution given a mean and a standard deviation. In our case, the x and y components were generated using mean 0 and standard deviation 10000. For more information about the `rnorm` function, visit `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Normal.html`

[4]The dataset containing the coordinates was created by *MaxMind*, available from `https://www.maxmind.com/en/free-world-cities-database`. The original dataset contains $3,173,959$ cities, but some of them have the same coordinates. We selected the $2,243,467$ cities with unique coordinates to build our dataset `wc`.

[5]Available at `http://www.cs.cmu.edu/~quake/triangle.html`. Our triangulations were generated using the options `-cezCBVPNE`.

[6]Available at `https://github.com/graph-algorithms/edge-addition-planarity-suite`. Our embeddings were generated using the options `-s -q -p`.

Dataset	PLAIN	LIGRA+	BRSV	AD	K^2-TREE	GS	PEMB
wc	74.67	52.50	14.57	14.73	16.40	14.88	6.00
pe5M	74.67	52.99	14.97	14.14	15.33	15.12	5.93
pe10M	74.67	53.15	15.03	14.33	14.73	15.12	5.93
pe15M	74.67	53.20	15.04	14.38	14.38	15.12	5.72
pe20M	74.67	53.24	15.07	14.43	14.15	15.12	5.93
pe25M	74.67	53.32	15.11	14.50	13.96	15.14	5.80

Table 2: Bits per edge (bpe) of the plain representation, alternative compressed graph representations, and ours.

useful properties: *locality* (a vertex with index i will have neighbours with indexes close to i), and *similarity* (vertices with similar index will have similar adjacency lists). Thus, the vertices and their adjacency lists are compressed following the ordering induced by the BFS traversal. The solution of Boldi *et al.* [51] (BRSV) reorders the nodes based on a clustering algorithm called *Layered Label Propagation (LLP)*. The LLP algorithm is used in combination with the *WebGraph* framework [55]. Brisaboa *et al.* [52] proposed the k^2-*tree* structure for graph compression. The k^2-tree is a compact tree representation of the adjacency matrix of a graph. The structure exploits the clustering of the edges in the adjacency matrix, representing large empty areas of the matrix efficiently. The clustering is dependent on the ordering of the vertices of the graph. In our comparison, we used the k^2-tree structure combined with the BFS traversal of [50], as suggested by Hernández and Navarro [56]. Blandford *et al.* [53] proposed a compact representation based on graph separators (GS). To construct the compact representation, the vertices of the graph must be renumbered. The new numbering is computed recursively, decomposing the graph by the computation of graph separators. The sequence of computed separators generate the new numbering. After the renumbering step, adjacent vertices tend to be close in the numbering. The representation takes advantage of that and reorders the adjacency list of each vertex, storing the difference between consecutive neighbours. Finally, the adjacency lists are encoded space-efficiently. In our experiments, we use the *child-flipping* heuristic [53] to compute the numbering of the vertices and *snip* code to encode the adjacency lists, which was the best among the choices we tested. Shun *et al.* [54] introduced *Ligra+*, a lightweight graph processing framework for shared-memory multicore machines. In Ligra+, the graph is stored in compressed form, by compressing the adjacency list of each vertex. The adjacency list of each vertex is sorted in increasing order and then the consecutive differences are run-length encoded. Finally, we also consider a plain representation (PLAIN) composed by an array of length $2m$, representing the concatenation of the adjacency lists, and an array of length n, representing the beginning of the adjacency list of each vertex.

Table 2 shows the bits per edge (*bpe*) of all the representations, where our solution is called PEMB, for *planar embedding*. In the table, we consider four

Dataset	PLAIN				PEMB			
	deg	list	face	dfs	deg	list	face	dfs
wc	0.01	0.12	0.35	0.51s	4.04	20.01	8.28	14.87s
pe5M	0.02	0.14	0.51	1.39s	4.24	20.65	8.55	34.61s
pe10M	0.03	0.14	0.60	2.65s	4.41	21.24	8.82	70.05s
pe15M	0.03	0.15	0.62	4.51s	4.51	21.61	8.98	106.50s
pe20M	0.03	0.15	0.64	5.66s	4.60	21.77	9.17	142.20s
pe25M	0.03	0.15	0.64	7.46s	4.64	22.15	9.40	181.09s
lim25M	9.31ms	42.62ms	12.79ms	$\gg$2hr	2.04μs	8.47μs	5.26μs	150.20s

Table 3: Median times of degree, listing and face queries, and the DFS traversal. All the values are in microseconds (μs), except the dfs columns and the lim25M row, which explicitly indicate μs, ms, s (seconds), or hr (hours). The row lim25M represents the results of the dataset pe25M in a machine with limited memory (Machine B).

bytes for each vertex and edge in the plain representation, equivalent to an integer number in common programming languages. Our compact representation reaches the best results, using less than half of the space of its closest competitor. Note that the other results, using widely different techniques, obtain very close results, around 15 bpe. This seems to suggest that exploiting planarity is the key to obtain a drastic reduction in space. Our results, with at most 6 bpe, are in accordance with the $4m + o(m)$ bits of Theorem 1.[7] Notice that due to the neighbor reordering needed by the other representations, they are not suitable for representing a particular planar embedding.

7.3. Query times

We test the time to carry out the three basic queries introduced in Section 5: degree (deg), listing (list) and face (face). Additionally, we test a more complex operation: a depth-first search traveral (dfs) starting from an arbitrary vertex and using a stack. We solve degree by sequentially traversing the edges, as we have not built the extra data structures to speed up this query. Observe that, given an adjacency list representation, answering degree and listing queries is straightforward. We measured the time of queries deg and list 10 times per vertex, face 10 times per edge, and dfs 10 times for 30 vertices chosen at random (i.e., 300 repetitions in total). Table 3 shows the median time per query, both for the plain representation and for our compact representation. The plain representation answers degree and listing queries 200 and 150 times faster than the compact representation, respectively. This result was expected, since the plain representation we use already has the list of neighbours in counter-

---

[7]We can get closer to 4 bpe by sparsifying the sublinear-size structures used to query bitvectors and parentheses, thus trading space for query time.

Dataset	Ligra+		BRSV		AD		κ^2-tree		GS	
	deg	list	deg	list	deg	list	deg	list	deg	list
wc	0.02	0.20	0.14	0.41	125.82	126.76	15.80	15.84	0.09	0.23
pe5M	0.02	0.26	0.17	0.43	124.45	121.16	13.73	13.75	0.09	0.26
pe10M	0.02	0.35	0.18	0.40	125.32	126.49	14.34	14.37	0.10	0.27
pe15M	0.02	0.39	0.19	0.41	125.02	122.02	14.85	14.86	0.10	0.27
pe20M	0.02	0.40	0.19	0.42	125.41	123.05	15.05	15.06	0.10	0.28
pe25M	0.02	0.41	0.20	0.41	126.08	126.56	15.45	15.41	0.10	0.28

Table 4: Median times of degree and listing queries for alternative compact representations. All the values are in microseconds (μs).

clockwise order. For the face query and the DFS traversal, the adjacency list representation is only 16 and 26 times faster, respectively.

This slowdown is the price of a representation that uses about 13 times less space, that is, it could hold graphs 13 times larger in main memory. To illustrate the effect of holding the compressed graph representation in main memory versus having to handle it on disk, we replicate the experiments in a machine with artificially limited memory (Machine B). For these new experiments we use the pe25M dataset, whose plain representation requires 668MB, whereas its compact representation needs only 52MB. The machine was set to use at most 600MB of RAM memory, just slightly less than the necessary to hold the whole input representation. The results are shown in the last row of Table 3. For degree query, the compact representation is around 4,500 times faster than the plain representation. For the listing query, the difference is around 5,000 times. For the face query, the compact representation is around 2,400 times faster than the plain representation. We aborted the experiment on the DFS traversal for the adjacency list representation after two hours; a projection of the other results suggests that more than a day would have been needed.

Thus, the compact representation pays off when it is the key to allow holding the graph in main memory.

We also measure the running time of degree and listing queries on the alternative compact graph representations introduced in Section 7.2. The results are shown in Table 4. We remark that these structures do not represent embeddings but graphs, and therefore are not a replacement for our representation. Still, the comparison gives more thorough information on the space/time tradeoff offered by our structure. Structure Ligra+ is much faster than our representation, but also an order of magnitude larger, roughly in the same category of the plain representation. All the other structures are about 2.5 times larger than ours, but their performance varies widely: while κ^2-tree and AD range from slightly faster to much slower that ours, the structures BRSV and GS are much faster (indeed, not much slower than a plain representation). For complex operations, such as DFS, the difference of our representation with respect to BRSV and

p	wc	pe5M	pe10M	pe15M	pe20M	pe25M
seq	2.93	7.36	15.46	23.61	31.76	40.01
1	3.56	8.93	18.77	28.78	39.33	49.20
2	2.24	5.15	10.74	16.24	21.88	27.26
4	1.33	2.98	5.94	8.94	12.31	15.25
8	0.76	1.73	3.43	5.03	6.53	8.14
12	0.54	1.22	2.43	3.59	4.70	5.84
16	0.43	1.00	1.86	2.80	3.66	4.54
20	0.36	0.80	1.63	2.37	3.13	3.88
24	0.31	0.72	1.41	2.05	2.83	3.44
28	0.27	0.65	1.23	1.97	2.50	3.08
32	0.27	0.60	1.21	1.77	2.30	2.88
36	0.27	0.59	1.12	1.69	2.25	2.74
40	0.24	0.54	1.05	1.57	2.07	2.60
44	0.23	0.52	0.99	1.49	1.96	2.46
48	0.22	0.49	0.95	1.42	1.91	2.35
52	0.22	0.48	0.92	1.38	1.82	2.28
56	0.22	0.47	0.91	1.34	1.78	2.23

Table 5: Running times of the parallel construction algorithm in seconds.

GS is half the difference of the `list` query, reported in Table 4.

7.4. Parallel construction

We now evaluate the performance of our parallel construction. In our implementation of the parallel spanning tree algorithm of Bader and Cong [42], to limit the worst case, we included a treshold of $\mathcal{O}(m/p)$ elements in the stack size of each thread. Each time a thread has more nodes that the threshold, it creates a new parallel thread with half of its stack. Additionally, we also return, for each node, the reference to its parent. This yields better performance than forcing the first edge of each node to lead to its parent.

Additionally, we implemented a sequential algorithm called `seq`, which corresponds to a sequential DFS algorithm to build the spanning tree, followed by the serialization of the parallel algorithm. To serialize a parallel algorithm in the DyM model, we replaced each **parfor** keyword for the **for** keyword and deleted the **spawn** and **sync** keywords. Each data point is the median of 15 measurements.

Table 5 shows the running times obtained in our experiments, and Figure 3 shows the speedups compared with the `seq` algorithm. On average, the `seq` algorithm took about 82% of the time obtained by the parallel algorithm running with 1 thread. With $p \geq 2$, the parallel algorithm shows better times than the `seq` algorithm. We observe an almost linear speedup up to $p = 24$, with an efficiency of at least 40% for the smaller datasets and almost 50% for the bigger ones. With $p = 28$ the speedup has a slowdown, due to the topology of our machine. Up to 24 cores, all the threads were running in the same
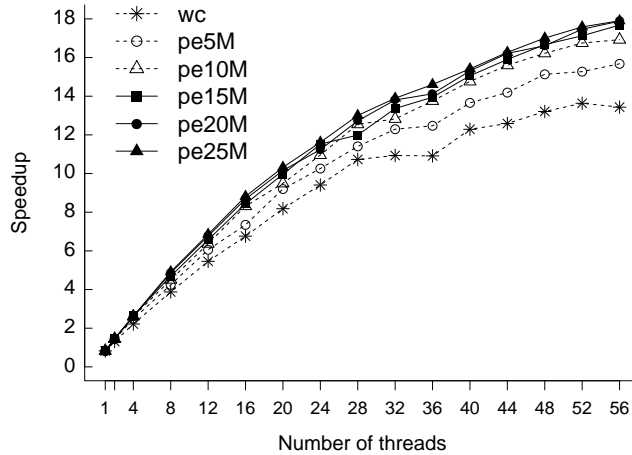
Figure 3: Speedup of the parallel algorithm.

NUMA node. With $p \geq 28$, both NUMA nodes are used, which implies higher communication costs. The communication costs intra NUMA nodes are lower than the communication costs inter NUMA nodes [57]. In particular, the case of $p = 28$ also uses both NUMA nodes, since at least one core on our machine was available to OS processes. For $p = 56$, the wc dataset exhibits an efficiency of only 24%, as it is the smallest one. For the bigger datasets, the lowest efficiency is 32%.

The running times and speedups reported in Table 5 and Figure 3 include the construction of bitvectors and balanced parentheses sequences, to support rank, select, parent, and match operations. To measure the efficiency of our algorithm, without the influence of the construction of those additional data structures, we repeated all the construction experiments, excluding the additional data structures. In the new experiments, we observed that the speedup increases on average 2.7% for $p \leq 24$ and 3.2% for $p \geq 28$, reaching a maximum speedup of 18.8, compared to the values reported in Figure 3.

Table 6 shows the running time for different edge densities of the dataset pe25M, and Figure 4 shows the corresponding speedups compared with the algorithm seq. The different densities are generated by deleting x million edges from the dataset pe25M, with $x \in \{5, 10, 15, 20, 25, 30\}$. If several components are generated, we reconnect them by restoring as few edges as possible, and then choosing new edges to be deleted. Thus, we report results for 45 to 75 (45Me to 75Me) million edges. The dataset 75Me corresponds to the original dataset pe25M. We observe a decrease in the running time for all values of p, according to the decrease in the number of edges. With respect to 75Me, the rest of the datasets show a greater decrease in the running time for increasing values of p, reaching speedups of up to 19.5 for 45Me. In the case of datasets with the same number of edges (see columns pe15M and pe20M in Table 5, and columns 45Me and 60Me in Table 6), the datasets with higher number of vertices show higher

30

p	45Me	50Me	55Me	60Me	65Me	70Me	75Me
seq	33.20	34.63	36.16	37.17	38.05	38.99	40.01
1	38.35	40.74	43.06	44.87	46.38	47.85	49.20
2	21.42	22.86	24.30	25.17	26.35	26.85	27.26
4	12.10	12.96	13.50	14.31	14.50	15.16	15.25
8	6.45	6.89	7.23	7.51	7.76	7.88	8.14
12	4.62	4.91	5.12	5.46	5.64	5.69	5.84
16	3.59	3.85	4.05	4.20	4.28	4.45	4.54
20	3.05	3.28	3.41	3.59	3.66	3.80	3.88
24	2.71	2.86	3.01	3.08	3.23	3.30	3.44
28	2.45	2.62	2.68	2.77	2.90	2.94	3.08
32	2.30	2.49	2.57	2.69	2.75	2.81	2.88
36	2.19	2.35	2.38	2.50	2.58	2.64	2.74
40	2.04	2.16	2.25	2.33	2.43	2.48	2.60
44	1.94	2.04	2.13	2.21	2.28	2.34	2.46
48	1.83	1.93	2.04	2.12	2.18	2.24	2.35
52	1.77	1.86	1.95	2.05	2.10	2.16	2.28
56	1.71	1.82	1.90	2.00	2.06	2.14	2.23

Table 6: Running times of the parallel construction algorithm varying the edge density for the dataset pe25M. The running times are measured in seconds.

running times. Comparing Figures 3 and 4, we observe that our algorithm scales similarly for triangulated and non-triangulated graphs.

Figure 5 shows the memory consumption of our algorithm. Specifically, the figure shows for each dataset the space used by its adjacency list representation (inputGraph), the peak consumption of our construction (peakMem) in addition to the input and the output, the space of its plain representation (plainGraph), and the size of its compact representation (compGraph). The plain representation, consisting of an array of edges of length $2m$ and an array of vertices of length n, is enough to navigate a graph, but for the construction we need more information about the embedding of the input graph. This richer adjacency list representation is what we call inputGraph. To measure the peak consumption, we use malloc_count[8], which monitors the memory allocated and released with malloc and free, respectively, and reports the peak usage. The observed peak consumption equals the size of the arrays LE, D_{pos} and D_{edge}. Compared with the space consumption of the input adjacency list representation, our implementation uses 73% of extra space. The final compact representation uses about 8% of the plain representation, as we have seen.

---

[8]Timo Bingmann. Malloc_count - Tools for runtime memory usage analysis and profiling. URL:https://panthema.net/2013/malloc_count/. Last accessed: August 08, 2017.
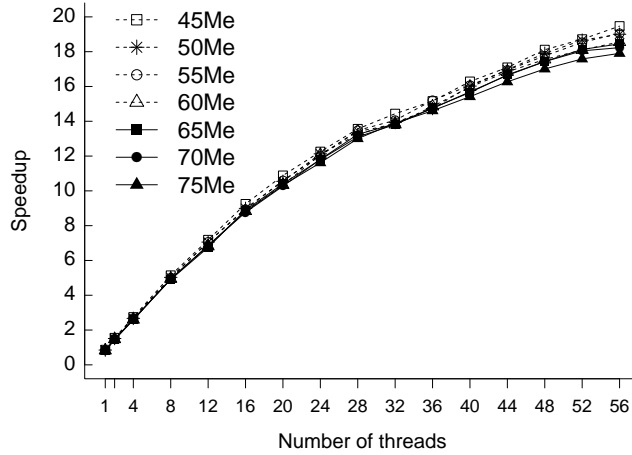
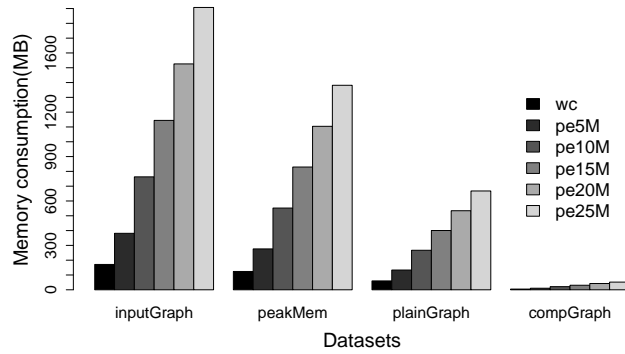Figure 4: Speedup of the parallel algorithm varying the edge density for the dataset `pe25M`.

Figure 5: Memory consumption of the parallel algorithm and the final compact structure.

8. Conclusions and future work

Turán's representation of planar embeddings [4] is much simpler than the known alternatives and encodes any planar embedding of m edges in just $4m$ bits, close to the lower bound of $3.58m$ bits. In this paper we have shown how to add $o(m)$ bits to this encoding in order to support fast nagivation and queries of the graph, in constant time for the most fundamental operations. While there are asymptotically optimal representations [3], the simplicity of Turán's encoding enabled us to introduce the first actual implementation of such a compact data structure, where the basic navigation operations are solved within microseconds. Further, the structure can be built at a rate of about one microsecond per edge, and the construction can be parallelized with linear speedup and an efficiency near 50%. Our parallel construction algorithm has linear work and logarithmic span on the dynamic multithreaded model once a spanning tree of the embedding is computed.

One intriguing question is about the queries we do not support in constant time. Some previous representations [8, 9, 3] can compute the degree of a node in $\mathcal{O}(1)$ time, whereas we can handle any superconstant time. Similarly, they can answer neighbour queries in $\mathcal{O}(1)$ time, whereas our structure needs superlogarithmic time. The representation closest to ours [9] uses the same technique of two types of parentheses, but the arrangement of the parentheses follows a so-called orderly spanning tree. While much more complex to build and unable to represent some embeddings, such spanning tree induces a certain regularity on the representation of the edges leaving each node, which allows determining in constant time the number of such edges, and whether two nodes are connected. It is an interesting question whether we can find a simpler arrangement that retains those properties.

Another future research line is how to make our data structure dynamic. We can use a scheme inspired by Munro et $al.$ [58]. Suppose we store our static data structure and a dynamic buffer that contains information about edges that have been added or deleted. If we want to know if an edge is present, we check our static data structure and then check the buffer to see if its status has changed. Once the buffer becomes too large — e.g., more than $m/\log^\epsilon m$ bits — we rebuild our static structure. Even when updates arrive sequentially, there are some issues to consider, such as how to quickly report the neighbours of a node that originally had many edges but has had most of them deleted (perhaps by moving all the information about a node into the buffer when half its incident edges have been updated) and how to detect if the graph has become non-planar. There are more issues when the updates can be made in parallel, since then we may need locks for nodes and finding a practical design becomes challenging.

Finally, we believe we can generalize our data structure to store efficiently graphs that are almost planar, using for example generalizations of the technique of Fischer and Peters [59] to store graphs that are almost trees. Of course, it is NP-hard to find the maximum planar subgraph of an arbitrary graph [60], but there have been recent advances in approximating it and in practice bridges and tunnels, for example, might already be identified anyway.

Acknowledgments

References

[1] L. Ferres, J. Fuentes, T. Gagie, M. He, G. Navarro, Fast and compact planar embeddings, in: Proceedings of the 15th International Symposium, Algorithms and Data Structures (WADS), Springer International Publishing, 2017, pp. 385–396.

[2] W. T. Tutte, A census of planar maps, Canadian Journal of Mathematics 15 (1963) 249–271.

[3] G. E. Blelloch, A. Farzan, Succinct representations of separable graphs, in: Proceedings of the 21st Annual Conference on Combinatorial Pattern Matching (CPM), Springer-Verlag, 2010, pp. 138–150.

[4] G. Turán, On the succinct representation of graphs, Discrete Applied Mathematics 8 (3) (1984) 289 – 294.

[5] K. Keeler, J. Westbrook, Short encodings of planar graphs and maps, Discrete Applied Mathematics 58 (1995) 239–252.

[6] G. Jacobson, Space-efficient static trees and graphs, in: Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, 1989, pp. 549–554.

[7] J. Barbay, L. C. Aleardi, M. He, J. I. Munro, Succinct representation of labeled graphs, Algorithmica 62 (2012) 224–257.

[8] J. I. Munro, V. Raman, Succinct representation of balanced parentheses and static trees, SIAM Journal on Computing 31 (3) (2001) 762–776.

[9] Y.-T. Chiang, C.-C. Lin, H.-I. Lu, Orderly spanning trees with applications, SIAM Journal on Computing 34 (2005) 924–945.

[10] M. Yannakakis, Embedding planar graphs in four pages, Journal of Computer and System Sciences 38 (1) (1989) 36–67.

[11] R. C.-N. Chuang, A. Garg, X. He, M.-Y. Kao, H.-I. Lu, Compact encodings of planar graphs via canonical orderings and multiple parentheses, in: Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 1443, 1998, pp. 118–129.

[12] W. Schnyder, Embedding planar graphs on the grid, in: Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Society for Industrial and Applied Mathematics, 1990, pp. 138–148.

[13] D. K. Blandford, G. E. Blelloch, I. A. Kash, Compact representations of separable graphs, in: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Society for Industrial and Applied Mathematics, 2003, pp. 679–688.

[14] R. J. Lipton, R. E. Tarjan, A separator theorem for planar graphs, SIAM Journal on Applied Mathematics 36 (1979) 177–189.

[15] N. Bonichon, C. Gavoille, N. Hanusse, D. Poulalhon, G. Schaeffer, Planar graphs, via well-orderly maps and trees, Graphs and Combinatorics 22 (2) (2006) 185–202.

[16] X. He, M. Y. Kao, H.-I. Lu, A fast general methodology for information-theoretically optimal encodings of graphs, SIAM Journal on Computing 30 (2000) 838–846.

[17] L. C. Aleardi, O. Devillers, G. Schaeffer, Succinct representation of triangulations with a boundary, in: Proceedings of the 9th International Conference on Algorithms and Data Structures (WADS), Springer-Verlag, 2005, pp. 134–145.

[18] L. Castelli Aleardi, O. Devillers, G. Schaeffer, Succinct representations of planar maps, Theoretical Computer Science 408 (2-3) (2008) 174–187.

[19] E. Fusy, G. Schaeffer, D. Poulalhon, Dissections, orientations, and trees with applications to optimal mesh encoding and random sampling, ACM Transactions on Algorithms 4 (2) (2008) 19:1–19:48.

[20] K. Yamanaka, S.-I. Nakano, A compact encoding of plane triangulations with efficient query supports, Information Processing Letters 110 (18-19) (2010) 803–809.

[21] L. C. Aleardi, O. Devillers, Array-based compact data structures for triangulations: Practical solutions with theoretical guarantees, Journal of Computational Geometry 9 (1) (2018) 247–289.

[22] J. I. Munro, P. K. Nicholson, Compressed representations of graphs, in: Encyclopedia of Algorithms, Springer, 2016, pp. 382–386.

[23] G. Navarro, Compact Data Structures: A Practical Approach, Cambridge University Press, 2016.

[24] X. He, M.-Y. Kao, Parallel construction of canonical ordering and convex drawing of triconnected planar graphs, in: Proceedings of the 4th International Symposium on Algorithms and Computation (ISAAC), 1993, pp. 303–312.

[25] M. Kao, S. Teng, K. Toyama, An optimal parallel algorithm for planar cycle separators, Algorithmica 14 (1995) 398–408.

[26] M. Kao, M. Fürer, X. He, B. Raghavachari, Optimal parallel algorithms for straight-line grid embeddings of planar graphs, SIAM Journal on Discrete Mathematics 7 (4) (1994) 632–646.

[27] D. R. Clark, Compact PAT trees, Ph.D. thesis, University of Waterloo, Canada (1996).

[28] J. I. Munro, Tables, in: Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 1180, 1996, pp. 37–42.

[29] R. Raman, V. Raman, S. R. Satti, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, ACM Transactions on Algorithms 3 (4).

[30] R. F. Geary, N. Rahman, R. Raman, V. Raman, A simple optimal representation for balanced parentheses, Theoretical Computer Science 368 (3) (2006) 231–246.

[31] G. Navarro, K. Sadakane, Fully functional static and dynamic succinct trees, ACM Trans. Algorithms 10 (3) (2014) 16:1–16:39.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Multithreaded algorithms, in: Introduction to Algorithms, 3rd Edition, The MIT Press, 2009, pp. 772–812.

[33] R. D. Blumofe, C. E. Leiserson, Scheduling multithreaded computations by work stealing, Journal of the ACM 46 (5) (1999) 720–748.

[34] N. Biggs, Spanning trees of dual graphs, Journal of Combinatorial Theory, Series B 11 (2) (1971) 127–131.

[35] D. Eppstein, Dynamic generators of topologically embedded graphs, in: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Society for Industrial and Applied Mathematics, 2003, pp. 599–608.

[36] T. R. Riley, W. P. Thurston, The absence of efficient dual pairs of spanning trees in planar graphs, Electronic Journal of Combinatorics 13 (1).

[37] P. Ferragina, R. Venturini, A simple storage scheme for strings achieving entropy bounds, Theoretical Computer Science 371 (1) (2007) 115–121.

[38] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 2007, pp. 60–70.

[39] D. R. Helman, J. JáJá, Prefix computations on symmetric multiprocessors, Journal of Parallel and Distributed Computing 61 (2001) 265–278.

[40] J. Labeit, J. Shun, G. E. Blelloch, Parallel lightweight wavelet tree, suffix array and fm-index construction, Journal of Discrete Algorithms 43 (2017) 2–17.

[41] J. Fuentes-Sepúlveda, L. Ferres, M. He, N. Zeh, Parallel construction of succinct trees, Theoretical Computer Science. To appear.

[42] D. A. Bader, G. Cong, A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs), Journal of Parallel and Distributed Computing 65 (2005) 994–1006.

[43] G. Chapuy, E. Fusy, O. Giménez, M. Noy, On the diameter of random planar graphs, Combinatorics, Probability & Computing 24 (1) (2015) 145–178.

[44] R. Cole, Parallel merge sort, SIAM Journal on Computing 17 (4) (1988) 770–785.

[45] G. Bilardi, A. Nicolau, Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines, SIAM Journal on Computing 18 (2) (1989) 216–228.

[46] G. E. Shannon, A linear-processor algorithm for depth-first search in planar graphs, Information Processing Letters 29 (3) (1988) 119–123.

[47] M. Kao, S. Teng, K. Toyama, An optimal parallel algorithm for planar cycle separators, Algorithmica 14 (5) (1995) 398–408.

[48] T. Hagerup, Planar depth-first search in $O(\log n)$ parallel time, SIAM Journal on Computing 19 (4) (1990) 678–704.

[49] J. Shun, L. Dhulipala, G. Blelloch, A simple and practical linear-work parallel algorithm for connectivity, in: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2014, pp. 143–153.

[50] A. Apostolico, G. Drovandi, Graph compression by BFS, Algorithms 2 (3) (2009) 1031–1044.

[51] P. Boldi, M. Rosa, M. Santini, S. Vigna, Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks, in: Proceedings of the 20th International Conference on World Wide Web (WWW), 2011, pp. 587–596.

[52] N. Brisaboa, S. Ladra, G. Navarro, Compact representation of web graphs with extended functionality, Information Systems 39 (1) (2014) 152–174.

[53] D. K. Blandford, G. E. Blelloch, I. A. Kash, Compact representations of separable graphs, in: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2003, pp. 679–688.

[54] J. Shun, L. Dhulipala, G. E. Blelloch, Smaller and faster: Parallel processing of compressed graphs with Ligra+, in: Proceedings of the 25th Data Compression Conference (DCC), 2015, pp. 403–412.

[55] P. Boldi, S. Vigna, The webgraph framework I: Compression techniques, in: Proceedings of the 13th International Conference on World Wide Web (WWW), ACM, 2004, pp. 595–602.

[56] C. Hernández, G. Navarro, Compressed representations for web and social graphs, Knowledge and Information Systems 40 (2) (2014) 279–313.

[57] U. Drepper, What every programmer should know about memory (2007). URL http://people.redhat.com/drepper/cpumemory.pdf

[58] J. I. Munro, Y. Nekrich, J. S. Vitter, Dynamic data structures for document collections and graphs, in: Proceedings of the 34th ACM Symposium on Principles of Database Systems (PODS), 2015, pp. 277–289.

[59] J. Fischer, D. Peters, GLOUDS: Representing tree-like graphs, Journal of Discrete Algorithms 36 (2016) 39 – 49.

[60] M. Yannakakis, The eect of a connectivity requirement on the complexity of maximum subgraph problems, Journal of the ACM 26 (1979) 618–630.