Fast and Compact Planar Embeddings[⋆]

Leo Ferres[1], José Fuentes[2], Travis Gagie[3], Meng He[4], and Gonzalo Navarro[2]

[1] Faculty of Engineering, Universidad del Desarrollo
[2] CeBiB; Department of Computer Science, University of Chile
[3] CeBiB; EIT, Diego Portales University
[4] Faculty of Computer Science, Dalhousie University

Abstract. There are many representations of planar graphs but few are as elegant as Turán's (1984): it is simple and practical, uses only four bits per edge, can handle 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, thus overcoming this disadvantage. Other data structures for planar embeddings may be asymptotically faster or smaller but ours is simpler, and that can be a theoretical as well as a practical advantage: e.g., we show how our structure can be built efficiently in parallel.

1 Introduction

The rate at which we store data is increasing even faster than the speed and capacity of computing hardware. Thus, if we want to use what we store efficiently, 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 a bewildering number of ways to store planar graphs. Each of these representations has its disadvantages, however: e.g., some do not support fast navigation, some are large, some cannot represent multi-edges or certain embeddings, and some are complicated to build in practice, especially in parallel, which is a concern when dealing with massive datasets.

Tutte [26] showed that representing a specified embedding of a connected planar multi-graph with n vertices and m edges takes $m \lg 12 \approx 3.58m$ bits in the worst case. Turán [25] gave a very simple representation that uses $4m$ bits, but Jacobson [15] noted that it "does not allow fast searching" and proposed

[⋆] The second and fifth authors received travel funding from EU grant H2020-MSCA-RISE-2015 BIRDS GA No. 690941. The second, third and fifth authors received funding from Basal Funds FB0001, Conicyt, Chile. The third author received funding from Academy of Finland grant 268324. Early parts of this work were done while the third author was at the University of Helsinki and while the third and fifth authors were visiting the University of A Coruña. Many thanks to Jérémy Barbay, Luca Castelli Aleardi, Arash Farzan, Ian Munro, Pat Nicholson and Julian Shun. The third author is grateful to the late David Gregory for his course on graph theory.

one that instead uses $\mathcal{O}(m)$ bits and supports fast navigation. Keeler and Westbrook [17] noted in turn that "the constant factor in [Jacobson's] space bound is relatively large" and gave a representation that uses $m \lg 12 + \mathcal{O}(1)$ bits when the graph contains either no self-loops or no vertices with degree 1, but gave up fast navigation again. Chiang, Lin and Lu [8] gave a representation that uses $2m + 3n + o(m)$ bits with fast navigation, but it is based on orderly spanning trees; although all planar graphs can be represented with orderly spanning trees, some planar embeddings cannot. Blelloch and Farzan [6] extended work by Blandford et al. [5] and gave a representation that uses $m \lg 12 + o(m)$ bits with fast navigation of any specified embedding, but it is complicated and has not been implemented. Barbay et al. [3] gave a data structure that uses $\mathcal{O}(n)$ bits to represent a simple planar graph on n nodes with fast navigation, but the hidden coefficient is about 18. Other authors (see, e.g., [7, 13, 14]) have considered special kinds of planar graphs, notably as tri-connected planar graphs and triangulations. We refer the reader to Munro and Nicholson's [20] and Navarro's [21, Chapter 9] recent surveys for further discussion of compact data structures for graphs.

In this paper 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 counter-clockwise order using constant time per edge, and determine whether two vertices are neighbours or find a vertex's degree in $\mathcal{O}(f(m))$-time for any given function $f(m) \in \omega(1)$. Our data structure is faster, smaller or more expressive than any of the structures listed above except Blelloch and Farzan's, and it is much simpler than theirs. Our structure's simplicity is a theoretical as well as a practical advantage, in that we can build it in parallel with linear work and logarithmic span (albeit without support for fast neighbour and degree queries). We summarize our construction algorithm in this paper and will provide details in a subsequent paper. In contrast, we do not have such efficient parallel algorithms for finding the book embeddings [27], orderly spanning trees and triangulations of planar subdivisions required by, respectively, Jacobson's, Chiang et al.'s and Barbay et al.'s constructions. Blandford et al.'s and Blelloch and Farzan's constructions are based on finding small vertex separators [19] and, although Kao et al. [16] designed a linear-work and logarithmic-span algorithm for computing a cycle separator of a planar graph, both Blandford et al.'s and Blelloch and Farzan's constructions decompose the input graph by repeatedly computing separators until each piece is sufficiently small, which increases the total work to $\mathcal{O}(n \log n)$ even when this optimal parallel algorithm is used.

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 and interleaving that with a sequence 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 his 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, we can support fast navigation in the graph.

In Section 2 we briefly describe bitvectors and the balanced-parentheses representation of trees, which are the building blocks of our extension of Turán's representation. For further discussion of these data structures, we again direct the reader to Navarro's text [21]. In Section 3 we prove the observation mentioned above. In Section 4 we describe our data structure and how we implement queries. We summarize our parallel construction algorithm in Section 5 and report the results of our preliminary experiments.

2 Preliminaries

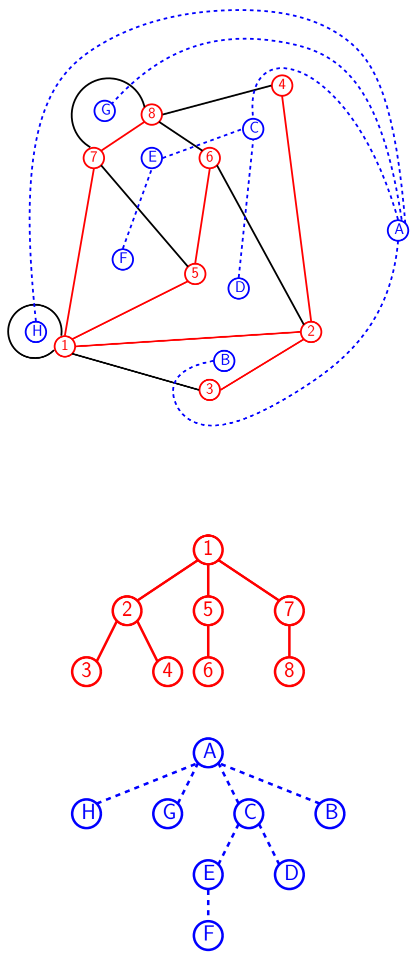
A bitvector is a binary string that supports the queries rank and select in addition to random access, where $rank_b(i)$ returns the number of bits set to b in the prefix of length ℓ of the string and $\mathsf{select}_b(j)$ returns the position of the jth bit set to b. For convenience, we define $\mathsf{select}_b(0) = 0$. There are many different implementations that represent a bitvector of length n in $n + o(n)$ bits and support random access, rank and select in constant time.

With bitvectors we can represent an ordered tree or forest on n vertices using $2n + o(n)$ bits and support natural navigation queries quickly. One of the most popular such representations is as 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, with 0s encoding opening parentheses and 1s encoding closing parentheses, and achieve the space bound stated above while supporting each of the follow queries used by our solution in constant time:

- $\mathsf{match}(i)$, locates the parenthesis matching the ith parenthesis,
- $\mathsf{parent}(v)$, returns the parent of v, given as its pre-order rank in the traversal, or 0 if v is the root of its tree.

3 Spanning trees of planar graphs

It is well known 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, with T and T^* interdigitating; see Figure 1 for an illustration (including multi-edges and a self-loop) and, e.g., [4, 11, 22] for discussions. 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^*.



	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)

Fig. 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.

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. If we perform a depth-first traversal of T starting from any vertex on the outer face of G and always process the edges (of the graph) incident to the vertex v we are visiting in counter-clockwise order (starting from the edge immediately after the one to v's parent or, if v is the root of T, from immediately after any incidence of the outer face), then each edge not in T corresponds to the next edge we cross in a depth-first traversal of T^*.*

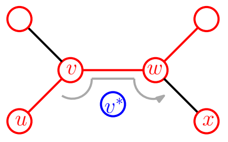
Proof. Suppose the traversal of T^* starts at the vertex of the dual of G corresponding to the outer face of G. We now prove by induction that the vertex we are visiting in T^* always corresponds to the face of G incident to the vertex we are visiting in T and to the previous and next edges in counter-clockwise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

Our claim is true before we process any edges, since we order the edges starting from an incidence of the outer face to 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 $(i+1)$st 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 next 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^*. $\square$

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 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.



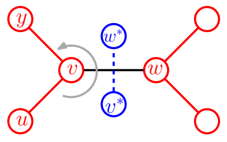


Fig. 2. Left: If we process an edge (v, w) in T, then we move to w in our traversal of T and the next edge, (w, x) in this case, is also incident to the vertex v^* we are visiting in our traversal of T^*. **Right:** If (v, w) is not in T, then in T^* we move from v^* to the vertex w^* corresponding to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

4 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 $A[i]$ indicates whether 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 $B[i]$ indicates whether the ith time we process an edge in T during the traversal, is the second time we process that edge;
- a bitvector $B^*[1..2(m-n+1)]$ in which $B^*[i]$ indicates whether the ith time we process an edge not in T during the traversal, is the second 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 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. Consider the following queries:

$\mathsf{first}(v)$**:** return i such that the first edge we process while visiting v is the ith we process during our traversal;

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

$\mathsf{mate}(i)$**:** return j such that we process the same edge ith and jth during our traversal;

$\mathsf{vertex}(i)$**:** 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:

$$\begin{array}{llll}
\mathsf{first}(1) = 1 & \mathsf{mate}(1) = 4 & \mathsf{vertex}(4) = 3 & A[1] = 0 \\
\mathsf{next}(1) = 2 & \mathsf{mate}(2) = 10 & \mathsf{vertex}(10) = 2 & A[2] = 1 \\
\mathsf{next}(2) = 11 & \mathsf{mate}(11) = 17 & \mathsf{vertex}(17) = 5 & A[11] = 1 \\
\mathsf{next}(11) = 18 & \mathsf{mate}(18) = 26 & \mathsf{vertex}(26) = 7 & A[18] = 1\,.
\end{array}$$

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 $(i+1)$st correctly given its endpoints and its rank in the counter-clockwise order at those vertices.

We now explain our constant-time implementations of first, next, mate and vertex. 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 $(v-1)$st 0 in B and, thus, the $B.\mathsf{select}_0(v-1)$st 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.

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,

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

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 $(5, 7)$.

To implement $\mathsf{mate}(i)$, we check $A[i]$ and use rank to determine whether we wrote a bit in B or in B^* while processing the ith edge, and to find that bit. We use match to find the bit encoding the matching parenthesis, and then use select on A to find the bit we wrote in A when we wrote 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 $A[12] = 1$,

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

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 $B[A.\mathsf{rank}_1(16)] = B[9] = 1$,

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

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.

Clearly we can determine whether two vertices u and v are neighbours by listing the neighbours of each in parallel in $\mathcal{O}(\min(\mathsf{degree}(u), \mathsf{degree}(v)))$ time, and we can find $\mathsf{degree}(v)$ in $\mathcal{O}(\mathsf{degree}(v))$ time. Moreover, given a function $f(m) \in \omega(1)$, we can make both kinds of queries take $\mathcal{O}(f(m))$ time. To do this, we store a bitvector marking the $\mathcal{O}(m/f(m)) = o(m)$ vertices with degree at least $f(m)$, which takes $o(m)$ bits. To be able to answer $\mathsf{neighbour}$ queries quickly, we consider the graph induced by those high-degree vertices and eliminate multi-edges and self-loops. The resulting simple graph G' is still planar — so it has average degree less than 6 and thus $o(m)$ edges — and preserves the neighbour relation between those vertices. We can store G' using Blelloch and Farzan's representation or, since the neighbour relation does not depend on the embedding, using one of the other compact representations of planar graphs that supports constant-time $\mathsf{neighbour}$ queries, which also takes $o(m)$ bits. To answer $\mathsf{neighbour}(u, v)$ now, we check whether either u or v is low-degree and, if so, list its neighbours in $\mathcal{O}(f(m))$ time; if not, we query our auxiliary representation in $\mathcal{O}(1)$ time. To be able to answer degree queries quickly, we simply store the degrees of the high-degree vertices in unary using a bitvector, which takes $o(m)$ bits. To find $\mathsf{degree}(v)$ now, we check whether v is low-degree and, if so, list and count its incident edges; if not, we look up $\mathsf{degree}(v)$.

Summarizing our results so far, we have the following theorem:

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 counter-clockwise order (optionally, starting at a* *given edge e incident to v) using constant time per edge, and determine whether two vertices are neighbours or find a vertex's degree in $\mathcal{O}(f(m))$-time for any given function $f(m) \in \omega(1)$.*

5 Parallel Construction and Experiments

Due to space constraints, in this section we can only summarize our parallel algorithm and then briefly report the result of our experiments on construction and query times. We will provide the full details of the algorithm in the subsequent paper mentioned in Section 1, of which a preprint is available at `http://arxiv.org/abs/1705.00415` .

We construct our extension of Turán's representation in parallel as follows: given a planar graph with a planar embedding G, we first compute a spanning tree T of G in parallel. In our experiments we used Bader and Cong's algorithm [2] because it works well in practice, but its theoretical bounds are for random graphs. To obtain good worst-case bounds, we could use Shiloach and Vishkin's [23] or Awerbuch and Shiloach's [1] algorithms, which use linear work with logarithmic span in the CRCW PRAM model. We recently learned that Shun, Dhulipala and Blelloch's [24] practical connectivity algorithm can be made to return a spanning tree with linear work and polylogarithmic span. As a by-product of the computation of T, we obtain an array C of length $2n-2$ that stores the number of edges of $G \setminus T$ between two consecutive edges in T, in counter-clockwise order. Notice the starting vertex for the spanning tree must be in the outer face of G.

We construct bitvectors A, B and B^* by performing a parallel Euler Tour over T [9]. During the tour, we obtain B by writing a 0 for each forward (parent-to-child) edge and a 1 for each backward (child-to-parent) edge. We obtain A by counting the number of edges of $G \setminus T$ between two consecutive edges of T (stored in C). We represent the former with 0's and the edges of T with 1's. The visiting order of edges of $G \setminus T$ encoded in B^* is implicit in the previous Euler Tour. Therefore, with the Euler Tour and the array C, we have enough information to compute the position of each bit in the bitvector B^*. We can decide if an edge of $G \setminus T$ is a forward or backward edge by checking its relative position and the position of its complement edge on the Euler tour. Finally, in order to support operations on A, B and B^*, we used Labeit et al.'s algorithm for succinct bitvectors [18], and Ferres et al.'s algorithm for succinct trees [12].

We analyze our algorithm (after the computation of the spanning tree) in the *Dynamic Multithreading (DyM)* model of parallel computation [10]. The DyM model relies on two parameters: the work T_1, i.e., the running time on a single core; and the span T_∞, i.e., the complexity of the intrinsically sequential part of the parallel computation. The time T_p needed to execute the computation on p cores is bounded by $T_p = \Theta(T_1/p + T_\infty)$. The Euler tour and the array C can be computed in parallel in $T_1 = O(n)$ and $T_\infty = O(\lg n)$ time. Assigning values to A and B can be done independently for each entry of the bitvectors, which gives us $T_1 = O(n)$ and $T_\infty = O(1)$ time, while B^* takes $T_1 = O(m-n)$ and $T_\infty = O(1)$

time. Rank/select structures can be constructed in parallel with $T_1 = O(m)$ and $T_\infty = O(\lg m)$ time. Succinct trees can be constructed in $T_1 = O(m)$ and $T_\infty = O(\lg m)$ time.

Theorem 2. *Given a spanning tree of a planar embedding, the compact representation from Theorem 1 (without the auxiliary data structures for fast* neighbour *and* degree *queries) can be constructed in parallel with linear work and logarithmic span.*

To provide some grounds for comparison, we also implemented a sequential algorithm based on our parallel one, but instead of using Bader and Cong's algorithm, we used depth-first search to compute T. This sequential implementation of Turán's representation is simpler than our parallel algorithm running on a single core, avoiding the additional steps needed for the parallel computation. To test both the sequential and parallel implementations, we synthetically generated a planar graph (represented as an adjacency list) by computing the Delaunay Triangulation of 25,000,000 random coordinates, yielding 25,000,000 vertices and 74,999,979 edges, with a minimum degree of 3 and a maximum degree of 15. The experiments were carried out on a 28-core machine (two processors with 14 physical cores each) with hyperthreading turned on (for a total of 56 cores), per-core L1 and L2 caches of sizes 64KB and 256KB, respectively and a per-processor shared L3 cache of 35MB and a total of 768GB DDR3 RAM memory.

Results show that the sequential algorithm took 71.8 seconds to construct the representation of Theorem 1, while the parallel implementation took 5.4 seconds with 28 threads and 3.3 seconds with 56 threads. The space used by the adjacency list representation was 1.02 GB, 117.3 bits per edge. The space used by our compact representation was 44.7 MB, 5 bits per edge, which matches Theorem 1. Memory consumption of our parallel algorithm peaked at 1.4 GB.

With respect to queries, we tested `counting` (number of neighbors) and `listing` (list of neighbors in counter-clockwise order) queries. For the former, the adjacency-list representation took 0.047 microseconds per node and the compact representation took 4.6 microseconds per node. For `listing`, the adjacency-list representation took 0.046 microseconds per node listed and the compact representation took 3.69 microseconds per node listed.

In summary, our parallel algorithm achieves a reasonable speed up in terms of the number of threads and is an already order of magnitude faster than the sequential algorithm when using 28 threads; our compact representation is between one and two orders of magnitude smaller than the adjacency-list representation, but takes two orders of magnitude more time to answer queries. Our code and datasets are available at `https://users.dcc.uchile.cl/~jfuentess/pemb/` .

References

1. B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Computers*, 36(10):1258–1263, 1987.

2. D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *J. Parallel and Distributed Computing*, 65:994–1006, 2005.
3. J. Barbay, L. C. Aleardi, M. He, and J. I. Munro. Succinct representation of labeled graphs. *Algorithmica*, 62:224–257, 2012.
4. N. Biggs. Spanning trees of dual graphs. *J. Comb. Theory, Series B*, 11:127–131, 1971.
5. D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *SODA*, pages 679–688, 2003.
6. G. E. Blelloch and A. Farzan. Succinct representations of separable graphs. In *CPM*, pages 138–150, 2010.
7. Castelli Aleardi, L, O. Devillers, and G. Schaeffer. Succinct representations of planar maps. *TCS*, 408:174–187, 2008.
8. Y.-T. Chiang, C.-C. Lin, and H.-I. Lu. Orderly spanning trees with applications. *SIAM J. Comp.*, 34:924–945, 2005.
9. G. Cong and D. A. Bader. The euler tour technique and parallel rooted spanning tree. In *ICPP*, pages 448–457, 2004.
10. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Multithreaded algorithms. In *Introduction to Algorithms*, pages 772–812. MIT Press, 2009.
11. D. Eppstein. Dynamic generators of topologically embedded graphs. In *SODA*, pages 599–608, 2003.
12. L. Ferres, J. Fuentes-Sepúlveda, M. He, and N. Zeh. Parallel construction of succinct trees. In *SEA*, pages 3–14, 2015.
13. É. Fusy, G. Schaeffer, and D. Poulalhon. Dissections, orientations, and trees with applications to optimal mesh encoding and random sampling. *TALG*, 4:19, 2008.
14. X. He, M. Kao, and H. Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *SIAM J. Discrete Math.*, 12:317–325, 1999.
15. G. Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554, 1989.
16. M. Kao, S. Teng, and K. Toyama. An optimal parallel algorithm for planar cycle separators. *Algorithmica*, 14:398–408, 1995.
17. K. Keeler and J. Westbrook. Short encodings of planar graphs and maps. *DAM*, 58:239–252, 1995.
18. J. Labeit, J. Shun, and G. E. Blelloch. Parallel lightweight wavelet tree, suffix array and FM-index construction. In *DCC*, pages 33–42, 2016.
19. R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Applied Math.*, 36:177–189, 1979.
20. J. I. Munro and P. K. Nicholson. Compressed representations of graphs. In *Encyclopedia of Algorithms*, pages 382–386. Springer, 2016.
21. G. Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.
22. T. R. Riley and W. P. Thurston. The absence of efficient dual pairs of spanning trees in planar graphs. *Electronic J. Comb.*, 13, 2006.
23. Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.
24. J. Shun, L. Dhulipala, and G. E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *SPAA*, pages 143–153, 2014.
25. G. Turán. On the succinct representation of graphs. *DAM*, 8:289–294, 1984.
26. W. T. Tutte. A census of planar maps. *Canadian J. Math.*, 15:249–271, 1963.
27. M. Yannakakis. Embedding planar graphs in four pages. *JCSS*, 38:36–67, 1989.