Navigating Planar Topologies in Near-Optimal Space and Time*

José Fuentes-Sepúlveda[a,c], Gonzalo Navarro[b,c], Diego Seco[a,c]

[a] *Department of Computer Science, Universidad de Concepción, Chile.*
[b] *Department of Computer Science, University of Chile, Chile.*
[c] *Millennium Institute for Foundational Research on Data, Chile.*

Abstract

We show that any embedding of a planar graph can be encoded succinctly while efficiently answering a number of topological queries near-optimally. More precisely, we build on a succinct representation that encodes an embedding of m edges within $4m$ bits, which is close to the information-theoretic lower bound of about $3.58m$. With $4m + o(m)$ bits of space, we show how to answer a number of topological queries relating nodes, edges, and faces, most of them in any time in $\omega(1)$. Indeed, $3.58m + o(m)$ bits suffice if the graph has no self-loops and no nodes of degree one. Further, we show that with $O(m)$ bits of space we can solve all those operations in $O(1)$ time.

Keywords: Planar graphs, Topology queries, Succinct data structures

1. Introduction

Plane embeddings, which are drawings of planar graphs on the plane, arise naturally in many applications, especially in those that are geometrical in nature like VLSI, computer graphics, and Geographic Information Systems (GIS) [1]. In this work we focus on efficiently answering queries that relate nodes, edges, and faces in planar embeddings. Those are the building blocks, for example, of the topological model, widely used in GIS applications to describe topological relationships among objects. With this underlying motivation, we define a comprehensive set of topological queries and show that they can be efficiently answered within very little space.

To achieve such space-efficient representations, we build on compact data structures [2], whose main goal is to support efficient query operations while using space close to the information-theoretic lower bound. Compact data struc-

*Funded by ANID - Millennium Science Initiative Program - Code ICN17_002, Chile and by European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 690941 (project BIRDS). The authors received funding from Fondecyt grants 77190038, 1200038, and 1170497, respectively. An early partial version of this paper appeared in *Proc. SPIRE 2019*.

tures have achieved remarkable results, both in theory and practice, to handle very large volumes of data in different domains, including graph and geometric data [3, 4, 5].

We build on Turán's encoding [6] of plane embeddings of connected planar graphs, where the *dual* of the graph (where the faces become nodes) is also explicitly represented. This representation requires $4m$ bits for a graph of m edges, where the optimum is $3.58m$ [7]. This representation can be reduced to $3.8m$ bits if the graph has no self-loops [8]. We use duality to show that this size is also achieved if the graph has no nodes of degree one. When the graph has none of them, the size drops to $3.58m$ bits [8].

Using only $o(m)$ extra bits on top of that representation, we extend previous results [8] in order to provide a succinct-space representation of the topological model that efficiently supports a rich set of topological queries (most of them in any time in $\omega(1)$), which include those defined in current standards and flagship implementations. Our main technical results are new $\omega(1)$-time algorithms for determining if two nodes are neighbors, and if a node touches a face, by orienting edges; many other results are derived via analogous structures and exploiting duality.

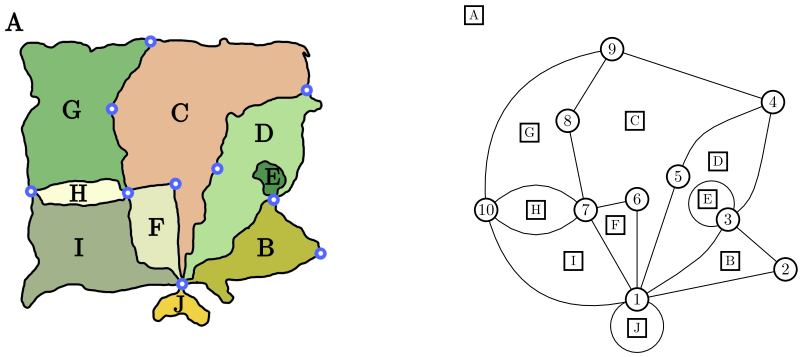
We then improve the time complexities by relaxing the space usage to $O(m)$ bits. We show that, within this space, all the operations can be supported in $O(1)$ time. Our main technique in this second part is a new $O(m)$-bits representation that allows us determining in constant time whether two nodes are in touch with the same unknown face.

2. Our Contribution in Context

2.1. An Application in GIS: The Topological Model

Geographic Information Systems (GIS) enable *capture, modeling, manipulation, retrieval, analysis and presentation* [9] of geographically referenced data. On the logical level, the most popular GIS model (together with the raster model) is the vector model. There are three common representations of collections of vector objects, called *spaghetti*, *network*, and *topological* model, which mainly differ in the expression of topological relationships among the objects [10]. In the spaghetti model, the geometry of each object is represented independently of the others and no explicit topological relations are stored. Despite its drawbacks, this is the most used model in practice because of its simplicity and the lack of efficient implementations of the other models. Those other two models are similar, and explicitly store topological relationships among objects. The network model is tailored to graph-based applications, such as transportation networks, whereas the topological model focuses on planar networks (e.g., all sorts of maps). This model is more efficient to answer topological queries, which are usually expensive, thus it is gaining popularity in spatial databases like Oracle Spatial.

The topological model represents a planar subdivision into adjacent polygons. Hereinafter, we will refer to these polygons as *faces*. A face is represented



(a) A planar subdivision

(b) The plane embedding of Fig. 1a.

Figure 1: Example of the topological model representing a planar subdivision.

as a sequence of *edges*, each of them being shared with an adjacent face, which may be the outer face. An edge connects two *nodes*, which are associated with a point in space, usually the Euclidean space. Edges also have a geometry, which represents the boundary shared between its two faces. In Fig. 1, faces are named with capital letters, A to J, A being the outer face. Face C is defined by the sequence of nodes $\langle 1, 5, 4, 9, 8, 7, 6\rangle$, and edge $(7, 8)$ is shared by faces C and G. Note, however, that a pair of nodes is insufficient in general to name an edge, because multiple edges may exist between two nodes. Note also that every intersection point is represented as a node (for example, node 4 represents the top right intersection between faces C and D), but the model also admits additional nodes such as 2, which may represent an important point of face B.

Those topological concepts are related with geographic entities. The basic geographic entity is the point, defined by two coordinates. Each node in the topological model is associated with a point, and each edge is associated with a sequence of points describing a sequence of segments that form the boundary between the two faces that share such edge. Each face is related to the area limited by its edges (the external face is infinite).

The international standard ISO/IEC 13249-3:2016 [11] defines a basic set of primitive operations for the model, which are also implemented in flagship database systems[1]. Some of the queries relate the geometry with the topology, for example, find the face covering a point given its coordinates. Those queries require data structures that store coordinates, and are therefore bound to use considerable space. Instead, we focus on *pure topological* queries, which can be solved within much less space and can encompass many problems once mapped to topological space. We also restrict our work to a static version of the model, in which case our representation supports a much richer set of access operations.

[1]A prominent example is PostGIS, the topology extension to PostgreSQL, see `http://postgis.net/docs/Topology.html`

The topological model has several advantages in comparison with the (non-topological) spaghetti model. First, by storing shared segments between faces only once, this model reduces the redundancy in the stored geometries, providing smaller representations. Second, it reduces inconsistencies in the data and facilitates the digitizing and editing of data. When data are represented in a non-topological model, each time a face is digitized or edited, a data cleaning process is necessary to detect and repair inconsistencies such as gaps between adjacent features. Finally, it is more efficient to answer topological queries. Although such queries can also be solved using the geometries, this approach is computationally very expensive (proportional to the number of points that make up the geometries and/or to the number of geometries).

According to the first[2] OGC Simple Features Implementation Specification for SQL [12], *the basic approach to comparing two geometries is to make pairwise tests of the intersections between the Interiors, Boundaries and Exteriors of the two geometries and to classify the relationship between the two geometries based on the entries in the resulting intersection matrix" [13].* Given the cost of this approach, for some popular topological queries, such as `ST_Touches`[3] (which corresponds to our query 1.d in Table 1), flagship implementations use a filter step based on the bounding box of the geometries and then a refinement step based on the comparison of the actual geometries. A similar filter and refinement strategy is used when the query involves comparing more than two geometries, such as retrieving the objects neighboring a given one. In that case, the filter is performed using a spatial index constructed over the bounding boxes of the objects, such as the R-tree [14], and the refinement is computed over the actual geometries of a set of candidates.

As we mentioned above, the geometric approach is computationally expensive mainly because the geometry of each object is complex, composed of a large number of points or pairs of geographic coordinates. To illustrate this point we provide some statistics computed over the popular TIGER dataset,[4] provided by the U.S. Census Bureau, which contains geographic and cartographic data about administrative divisions in the USA. The dataset provides many layers with different characteristics. For example, the layer that represents the states of the USA (including the District of Columbia and inhabited territories) has an average of $17,344$ points per state; counties use an average of $2,502$ points per county; county subdivisions use an average of $2,891$ points per element; and school districts use an average of 718 points per district. As it can be seen, the number of points per object varies with the type of layer, spatial resolution, etc. but it can be in the order of hundreds or thousands of points per object, which makes this approach expensive.

We propose instead an approach in which most of the work is done on an in-memory compact index on the topology, resorting to the geometric data only

[2]Current versions of the specification do not address how to implement such operations.
[3]`https://postgis.net/docs/ST_Touches.html`
[4]TIGER dataset, version 2021. `https://www2.census.gov/geo/tiger/TIGER2021/`

when necessary. Such an approach enables handling geometries that do not fit in main memory, but whose topologies do, and still solving queries on them with reasonable efficiency because secondary-memory accesses are limited. To illustrate this, consider the example of *given the coordinates of two query points, tell if they lie on adjacent faces, and if so, which edge separates them.* In our approach, this type of query can be solved with just two mappings from the geographical space to the topological space, and then using pure topological queries.

Using the same example dataset, the largest layer in TIGER (called Faces) contains about $f = 20$ million faces, $m = 44$ million edges, and $n = 24$ million nodes. The shapefiles that store all geographic information (besides some alphanumeric data) use almost 300 GB. An implementation of the topological model based on the half-edge data structure[5] (see Section 3.2 for a discussion on how such a data structure relates to our approach) requires a pointer for each node, face and edge to identify a half-edge, and each half-edge stores its twin, next, vertex, edge, and face. Hence, the total space is $n+m+f+5\times 2m = 12m$ words, which in our example amounts to about 2 GB. Instead, the $4m$ bits of Turán's compact representation, on which we build, entail just over 20 MB. In this example, then, the use of the topological model reduces the data size by two orders of magnitude, and the use of a compact representation slashes two further orders of magnitude in the space usage.

Table 1 lists the set of topological queries we consider on the topological model, together with the time complexities we achieve in this paper within $4m + o(m)$ bits (we solve them all in $O(1)$ time within $O(m)$ bits). These comprehensively consider querying about relations between two given entities of the same or different type, and listing or counting entities related to a given one. The set considerably extends the queries available in standards or flagship implementations, which comprise just `ST_Touches` (1.d and 5.b), `GetNodeEdges` (3.c), and `ST_GetFaceEdges` (3.e); we are here using the operation names defined in the standard [11]. The queries can be useful not only in GIS applications, but also in many others where data is modelled with plane embeddings, such as road networks, city maps, chip design, network routing, and polygon meshes, to name a few.

2.2. Planar Graphs

A graph is *planar* if it can be drawn on the plane without crossing edges. The topology of a specific drawing of a planar graph on the plane is called a *plane embedding*. Representing a plane embedding with m edges requires $m \log 12 \approx 3.58m$ bits [7, Eq. 5.1], which opens the door to $O(m)$-bit representations. This is remarkable because representing general graphs with n nodes and m edges needs $\Omega(m \log \frac{n^2}{m})$ bits. The lower bound for representing planar graphs is unknown, but it is of course at most $m \log 12$ bits because every planar

[5] https://cs184.eecs.berkeley.edu/sp19/article/15/the-half-edge-data-structure

Table 1: The queries we consider on the topological model and our time complexities within $4m + o(m)$-bit space. We put in boldface those where we contributed.

1. Relations between entities of the same type		
(1.a) Do edges e and e' share a node?	$O(1)$	[8] + Lemma 2
(1.b) Do edges e and e' border the same face?	$O(1)$	[8] + Lemma 2
(1.c) Do nodes u and v share an edge?	any in $\omega(1)$	Lemma 4
(1.d) Do faces x and y share an edge?	any in $\omega(1)$	Lemma 5
2. Relations between entities of different type		
(2.a) Is edge e incident on node u?	$O(1)$	[8] + Lemma 2
(2.b) Is edge e on the border of face x?	$O(1)$	[8] + Lemma 2
(2.c) Is face x incident on node u?	any in $\omega(1)$	Lemma 6
3. Listing related entities (time per element output)		
(3.a) Endpoints of edge e	$O(1)$	[8] + Lemma 2
(3.b) Faces divided by edge e	$O(1)$	[8] + Lemma 2
(3.c) Nodes/edges neighbors of node u	$O(1)$	[8]
(3.d) Faces bordering face x	$O(1)$	[8] and duality
(3.e) Faces incident on node u	$O(1)$	Lemma 3
(3.f) Nodes/edges bordering face x	$O(1)$	Lemma 3
4. Counting related entities		
(4.a) Nodes/edges/faces neighbors of node u	any in $\omega(1)$	[8] extended
(4.b) Faces/edges/nodes bordering face x	any in $\omega(1)$	[8] and duality
5. Relations via a third entity		
(5.a) Do nodes u and v border the same face?	any in $\omega(\sqrt{m})$	Lemma 7
(5.b) Do faces x and y share a node?	any in $\omega(\sqrt{m})$	Lemma 7

graph has a plane embedding but there may be more than one plane embedding of the same planar graph.

There are several succinct representations of planar graphs, most of which cannot represent a particular embedding. Some offer the basic graph queries, such as determining the adjacency of two nodes, and listing or counting the neighbors of a node. Table 2 lists the main developments.

In this paper we use plane embeddings to represent topological models. Succinct representations of plane embeddings build on spanning trees, book embeddings [22], realizers [23], and small node separators [24]. Turán [6] introduced a succinct representation using $4m$ bits, and Keeler and Westbrook [15] reached the optimal $m \log 12 + O(1)$ bits, though disallowing either self-loops or nodes with degree 1. Both used spanning trees. He et al. [16] used graph separators and obtained $m \log 12 + o(m)$ bits without restrictions. Those representations do not support efficient navigation of the compressed representation, however.

There exist a number of navigable representations, which support a few basic queries in optimal time: adjacency (are these two nodes connected?), neigh-

Planar graphs (unknown lower bound is $\leq 3.58m$)							
Source	Bits	Functionality			Graph features		
		Adj	Neigh	Deg	Multi	Loops	Sticks
Keeler & Westbrook [15]	$3.58m$				✓	✓	✓
He et al. [16]	optimal				✓	✓	✓
Jacobson [17]	$36n$	$O(\log n)$	$O(\log n)$		✓	✓	✓
Munro & Raman [18]	$2m + 8n$	$O(1)$	$O(1)$	$O(1)$	✓		✓
Chuang et al. [19]	$2m + (5+\epsilon)n$	$O(1)$	$O(1)$	$O(1)$	✓		✓
	$\frac{5}{3}m + (5+\epsilon)n$	$O(1)$	$O(1)$	$O(1)$			✓
	$2m + \frac{14}{3}n$	$O(1)$	$O(1)$		✓		✓
	$\frac{4}{3}m + 5n$	$O(1)$	$O(1)$				✓
Chiang et al. [20]	$2m + 3n$	$O(1)$	$O(1)$	$O(1)$	✓		✓
	$2m + 2n$	$O(1)$	$O(1)$	$O(1)$			✓
Blelloch & Farzan [21]	optimal	$O(1)$	$O(1)$	$O(1)$	✓	✓	✓

Table 2: Comparison of space and functionality of the succinct representations of connected planar graphs. In the functionalities, Adj refers to determining whether two nodes are adjacent, Neigh to listing the neighbors of a node, and Deg to computing the degree of a node. In the supported graph features, Multi refers to multiple edges between pairs of nodes, Loops to self-loops, and Sticks to degree-1 nodes. The number of nodes in the graph is denoted by n and the number of edges by m.

borhood (list the neighbors of this node, in clockwise (cw) or counter-clockwise (ccw) order), and degree (how many neighbors this node has?). Barbay et al. [25] provided a representation for simple plane embeddings based on realizers. They use $O(m)$ bits and solve those queries in constant time per retrieved answer, but the precise space complexity is over $6m$ bits. Blelloch and Farzan [21] devised a representation using the optimal $m \log 12 + o(m)$ bits based on small node separators. Ferres et al. [8] use spanning trees to provide a simple and implementable representation using $4m + o(m)$ bits, though adjacency and degree queries take superconstant time.

Succinct representations of several other subclasses of planar graphs have been studied. For example, triconnected planar graphs require $2m$ bits [7, Eq. 8.17], which was matched by Castelli Aleardi et al. [4], who support constant-time adjacency queries between nodes and faces in $O(1)$ time and within $o(m)$ extra bits of space. They also matched the lower bound of triangulations ($1.08m$ bits [26, Eq. 5.11]) while supporting the same queries. The space was matched earlier [16] without query support. Instead, Yamanaka and Nakano [27], and later Ferres et al. [8], support constant-time adjacency, degree, and neighbor listing queries in $O(1)$ per returned result, using (the non-optimal) $2m + o(m)$ bits. Those results subsume earlier ones [15, 19].

2.3. Our Contribution

In this paper we are interested in the representation of Ferres et al. [8], which extends Turán's encoding with $o(m)$ extra bits in order to support efficient navigation operations. They list neighbors in optimal time, and show how to list all the edges of a face in optimal time as well. However, computing degrees requires (any) time in $\omega(1)$ and determining adjacency of two nodes requires (any) time in $\omega(\log n)$. Compared to other more efficient representations, however, Turán's encoding is interesting because it includes an explicit representation of both the plane embedding and of its dual, that is, one can directly refer to faces and pose queries on them. We use this feature to extend the set of primitives so as to support a full set of topological queries, formed by all the operations listed in Table 1. Moreover, we improve their performance for adjacency and related queries to any time in $\omega(1)$. Table 3 puts our contribution in context.

A warmup result essentially hinted by Ferres et al., our Lemma 2, sorts out a number of simple queries (all [123].[ab]) in constant time. A consequence of Lemma 2 is Lemma 3, which extends the algorithm of Ferres et al. listing the neighbors of a node (3.c, `GetNodeEdges`) in optimal time to list the faces incident on a node (3.e) and, by duality, to list the faces or edges bordering a face (3.d, `ST_GetFaceEdges`) and the nodes bordering a face (3.f), all in optimal time. We also extend their results that count the edges incident on a node (4.a) in $\omega(1)$ time to count nodes, edges, or faces incident on a node or bordering a face (4.b).

Our first main result is Lemma 4, which exploits orientation of edges to determine if two given nodes are connected by an edge (1.c) in any time in $\omega(1)$, adding only $o(m)$ bits to the main structure. The same procedure on the dual graph, Lemma 5, determines in the same time if two given faces share

Plane embeddings (lower bound is $3.58m$)									
Source	Bits	Functionality					Graph features		
		Type 1	Type 2	Type 3	Type 4	Type 5	Multi	Loops	Sticks
Turán [6]	$4m$						✓	✓	✓
Keeler & Westbrook [15]	$3.58m$						✓	✓	
	$3.58m$						✓		✓
	$3m$						✓		
He et al. [16]	$3.58m$						✓	✓	✓
Barbay et al. [25]	$18.51n$	1.c $O(1)$		3.c $O(1)$	4.a $O(1)$				✓
Blelloch & Farzan [21]	$3.58m$	1.c $O(1)$		3.c $O(1)$	4.a $O(1)$		✓	✓	✓
Ferres et al. [8]	$4m$	1.c $\omega(\log n)$		3.c $O(1)$	4.a $\omega(1)$		✓	✓	✓
	$3.80m$	1.c $\omega(\log n)$		3.c $O(1)$	4.a $\omega(1)$		✓		✓
	$3.58m$	1.c $\omega(\log n)$		3.c $O(1)$	4.a $\omega(1)$		✓		
Ours	$4m$	1.a-b $O(1)$, 1.c-d $\omega(1)$	2.a-b $O(1)$, 2.c $\omega(1)$	3.a-f $O(1)$	4.a-b $\omega(1)$	5.a-b $\omega(\sqrt{m})$	✓	✓	✓
	$3.80m$	1.a-b $O(1)$, 1.c-d $\omega(1)$	2.a-b $O(1)$, 2.c $\omega(1)$	3.a-f $O(1)$	4.a-b $\omega(1)$	5.a-b $\omega(\sqrt{m})$	✓		✓
	$3.80m$	1.a-b $O(1)$, 1.c-d $\omega(1)$	2.a-b $O(1)$, 2.c $\omega(1)$	3.a-f $O(1)$	4.a-b $\omega(1)$	5.a-b $\omega(\sqrt{m})$	✓	✓	
	$3.58m$	1.a-b $O(1)$, 1.c-d $\omega(1)$	2.a-b $O(1)$, 2.c $\omega(1)$	3.a-f $O(1)$	4.a-b $\omega(1)$	5.a-b $\omega(\sqrt{m})$	✓		
	$O(m)$	1.a-d $O(1)$	2.a-c $O(1)$	3.a-f $O(1)$	4.a-b $O(1)$	5.a-b $O(1)$	✓	✓	✓

Table 3: Comparison of space and functionality of the succinct representations of connected plane embeddings. The types of functionality refer to those in Table 1. In the supported graph features, Multi refers to multiple edges between pairs of nodes, Loops to self-loops, and Sticks to degree-1 nodes. The number of nodes in the graph is denoted by n and the number of edges by m; we omit the extra $o(m)$ bits in the spaces.

an edge (1.d, a variant of the standard query `ST_Touches`). Our second main result is Lemma 6, which builds on Lemma 4 to determine if a given node is in the frontier of a given face (2.c) in any time in $\omega(1)$, by defining a new graph where faces become nodes as well. Determining if two given nodes border the same face (5.a) or if two given faces share some node (5.b, a variant of query `ST_Touches`) is costlier, $\omega(\sqrt{m})$.

Our general approach is to solve the queries by enumeration (queries of type 3), mapping the "hard" nodes/faces where enumeration would be too expensive to a smaller graph where we can store extra information in $o(m)$ bits that allows handling the query in $O(1)$ additional time. The main challenge is to define what extra information to store, and how to store it, so that we take $o(m)$ bits of space and we can map to the reduced graph in constant time.

If we use $(4+\epsilon)m$ bits of space, for any constant $\epsilon > 0$, then we have space to interpret "too expensive" in the previous paragraph to "more than a constant", then all the times of the form "any in $\omega(1)$" in Table 1 become $O(1)$. We note that Ferres et al. are able to reduce the space of their representation to $3.8m + o(m)$ bits if the graph has no self-loops, and to $3.58m + o(m)$ bits if it also has no nodes of degree one, without affecting the algorithms. Those reductions carry over our results, because none of the modifications we make to the graph add self-loops or nodes of degree one. We also prove that the space is $3.8m + o(m)$ bits if the graph has no nodes of degree one. In those cases, our space of the form $(4+\epsilon)m$ bits also becomes $(3.8+\epsilon)m$ and $(3.58+\epsilon)m$, respectively. We finally explore what can be achieved if we allow any space usage in $O(m)$ bits. Lemma 8 shows that, by modifying the arrangement of Lemma 6, we also solve queries (5.a) and (5.b) in constant time. As a result, all the queries in Table 1 can be solved in $O(1)$ time and within $O(m)$ bits of space. The following theorem summarizes our results.

Theorem 1. *An embedding of a connected planar graph with m edges can be represented in $4m+o(m)$ bits so that the queries listed in Table 1 can be answered in the given time complexities. The space decreases to $3.8m+o(m)$ bits if there are no self-loops or no nodes of degree one, and to $3.58m+o(m)$ bits if there are none of both. By using $O(m)$ bits, all the queries can be solved in $O(1)$ time.*

Note that the given space results assume that the graph is connected. Ferres et al. [8] show how an embedding formed by k connected components can be optimally represented by adding $k\lg(m/k)+O(k)$ bits and without any essential change to the algorithms designed for connected graphs.

A preliminary version of this article appeared in *Proc. SPIRE'19* [28]. In this extended version we present the results in greater detail, and manage to improve their time for queries (1.c) and (1.d) from $O(\frac{\log\log m}{\log\log\log m})$, and for query (2.c) from $\omega(\log n)$, to any time in $\omega(1)$. We also reduce the space to $3.8m+o(m)$ bits when there are no nodes of degree one. Further, we obtain constant time on all the queries of Table 1 by relaxing the space usage to $O(m)$ bits.

3. Succinct Data Structures

3.1. Sequences and Parentheses

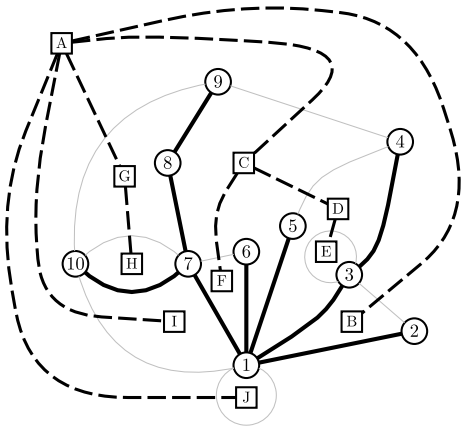
Given a sequence $S[1..n]$ defined over an arbitrary alphabet Σ of size σ, the operation $rank_a(S, i)$ returns the number of occurrences of the symbol $a \in \Sigma$ in the prefix $S[1..i]$, and the operation $select_a(S, i)$ returns the position in S of the ith occurrence of the symbol $a \in \Sigma$. For binary alphabets, the bitvector S can be stored in $n + o(n)$ bits supporting *rank* and *select* in $O(1)$ time [29]. If S has m 1-bits, then it can be represented in $m \lg \frac{n}{m} + O(m) + o(n)$ bits, maintaining $O(1)$-time *rank* and *select* [30].

Binary sequences can be used to represent balanced parenthesis sequences, by interpreting the bit values as opening or closing parentheses. Given a balanced parenthesis sequence S, $open(S, i)/close(S, i)$ returns the position in S of the closing/opening parenthesis matching the parenthesis $S[i]$, and $enclose(S, i)$ returns the rightmost position j such that $j < i < close(S, j)$. A parentheses sequence can be used to represent an ordinal tree, where each node is identified by the position i of an opening parenthesis $S[i]$ and its descendant nodes are listed between positions $i + 1$ and $close(S, i) - 1$. The parent of the node i is $parent(S, i) = enclose(S, i)$. Another relevant operation for this interpretation is $child(S, i, j)$, which yields the opening parenthesis of the jth child of the node identified by position i. The sequence S can be represented in $n + o(n)$ bits and support *open*, *close*, *enclose*, *parent*, and *child*, all in $O(1)$ time [31].

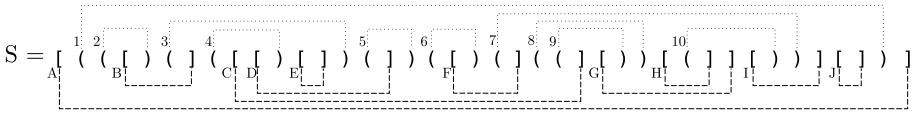
3.2. Ferres et al.'s Representation

Given a plane embedding of a connected planar graph G, the computation of a spanning tree T of G induces a spanning tree T^* in the dual graph of G [32]. The edges of T^* correspond to the edges in the dual graph that cross edges in $G \setminus T$. Fig. 2a shows a primal (thick continuous edges) and a dual (thick dashed edges) spanning trees for the plane embedding of Fig. 1b. Lemma 1 states that a depth-first traversal of T induces a depth-first traversal in T^*. We deviate a bit from the presentation of Ferres et al. and resort to the concept of *half-edge* [33]: every edge is associated with two *twin* half-edges, each belonging to one of the two faces limited by the edge. The half-edges are oriented, so they have a source and a target node, and the twin half-edges have opposite orientation. The successive half-edges belonging to a face run in the same direction along its frontier (i.e., the target of a half-edge is the source of the next). A *corner* is defined by a pair of consecutive half-edges of the same face, such that the target of first half-edge is the source of the second. The traversal of G then visits each half-edge exactly once.

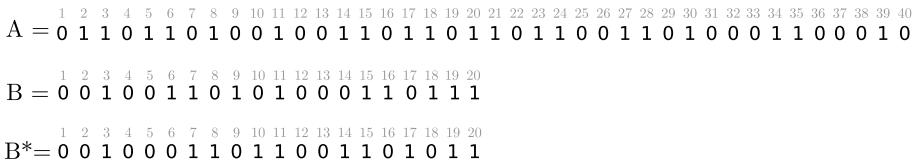
Lemma 1 ([8]). *Consider any plane embedding of a planar graph G, any spanning tree T of G and the complementary spanning tree T^* of the dual G^* of G. Suppose we perform a depth-first traversal of T starting from any node on the outer face of G and process in ccw order the half-edges whose source is the node v we are visiting. To start, we arbitrarily choose two half-edges of the outer face forming a corner on the root of T and start from that second half-edge; at any*



(a) Primal and dual spanning trees of the plane embedding of Fig. 1b.



(b) The sequence of parentheses and brackets encoding the plane embedding



(c) Bitvectors A, B and B^* representing the sequence S

Figure 2: Example of the succinct plane embedding representation of Ferres et al. [8].

other node v with parent u in T, we start from the half-edge with source v that follows (in ccw order) the twin of the one coming from u. Then each edge not in T corresponds to the next edge we cross in a depth-first traversal of T^*, starting in the node of G^* that represents the outer face of G.

For instance, in Fig. 2a, taking node 1 as the spanning tree root and assuming the half-edges of the outer face run cw (as it will be the case if we traverse the node neighbors ccw), the traversal can start at half-edges $(1, 1)$ (due to the corner $(10, 1)$–$(1, 1)$) or $(1, 2)$ (due to the coner $(1, 1)$–$(1, 2)$).

The compact representation of our interest [6, 8] (as well as several others [19, 20, 27]) is based on the traversal of Lemma 1. Starting at the root of any suitable spanning tree T, each time we visit for the first time an edge e, we write a "(" if e belongs to T, or a "[" if not. Each time we visit an edge e for the second time, we write a ")" if e belongs to T or a "]" otherwise. We call S the resulting sequence of $2m$ parentheses and brackets, which are enclosed by an additional pair of parentheses and of brackets that represent the root and the outer face, respectively. Ranks of opening parentheses act as node identifiers,

whereas ranks of opening brackets act as face identifiers. Further, positions in S act as edge identifiers: each edge is identified twice, first by an opening parenthesis or bracket, and later by its corresponding closing parenthesis or bracket. In the half-edge model, there is exactly one half-edge per position in S; the half-edge i is precisely the one described at $S[i]$. An edge is identified by its two half-edges. We note that there is no relation between the direction of a half-edge and whether it is an opening or a closing parenthesis or bracket.

Fig. 2b shows the sequence S for the plane embedding of Fig. 2a, starting the traversal at the half-edge $(1, 2)$. Observe that the parentheses of S encode the balanced-parentheses representation of T and the brackets encode the balanced-parentheses representation of the dual spanning tree T^*. In general the representation of a node v, $(\cdots)$ where "(" is the vth opening parenthesis, contains the sequences of nested parenthesis sequences for the children of v in T (e.g., the sequence for node 1 in the figure contains those of the nodes 2 and 7), interspersed with top-level brackets (i.e., brackets not contained in the sequence of a child of v). The opening parenthesis representing v also represents the half-edge with target v (and source u, the parent of v in T); the parenthesis closing that of v represents the half-edge with source v and target u. The top-level brackets represent the other half-edges with source v (e.g., the "]" of I and the two half-edges "[ ]" of J with $v = 1$). Since the brackets also represent the faces of G, if we exchange the roles of brackets and parentheses, the sequence represents the dual graph G^*.

In the succinct representation of Ferres *et al.* [8], the sequence S is stored in three bitvectors, $A[1..2(m+2)]$, $B[1..2n]$, and $B^*[1..2(m-n+2)]$. It holds that $A[i] = 1$ if the ith entry of S is a parenthesis, and $A[i] = 0$ if it is a bracket. Bitvector B stores the balanced sequence of parentheses of S, storing a 0 for each opening parenthesis and a 1 for each closing parenthesis. Bitvector B^* stores the balanced sequence of brackets of S in a similar way. Fig. 2c shows the bitvectors that store the sequence S of Fig. 2b.

Adding support for *rank* and *select* operations on A, B and B^*, and for *open*, *close* and *enclose* (i.e., *parent*) operations on B and B^*, we simulate their support on S, as follows ($match(S, i)$ and $enclose(S, i)$ give the parenthesis or bracket matching or enclosing, respectively, the one at $S[i]$):

$$S[i] = \begin{cases} \text{“(”} & \text{if } A[i] = 1 \wedge B[rank_1(A, i)] = 0, \\ \text{“)”} & \text{if } A[i] = 1 \wedge B[rank_1(A, i)] = 1 \\ \text{“[”} & \text{if } A[i] = 0 \wedge B^*[rank_0(A, i)] = 0, \\ \text{“]”} & \text{if } A[i] = 0 \wedge B^*[rank_0(A, i)] = 1 \end{cases}$$

$$rank_c(S, i) = \begin{cases} rank_0(B, rank_1(A, i)) & \text{if } c = \text{“(”}, \\ rank_1(B, rank_1(A, i)) & \text{if } c = \text{“)”} \\ rank_0(B^*, rank_0(A, i)) & \text{if } c = \text{“[”}, \\ rank_1(B^*, rank_0(A, i)) & \text{if } c = \text{“]”} \end{cases}$$

$$select_c(S, i) = \begin{cases} select_1(A, select_0(B, i)) & \text{if } c = \text{“(”}, \\ select_1(A, select_1(B, i)) & \text{if } c = \text{“)”} \\ select_0(A, select_0(B^*, i)) & \text{if } c = \text{“[”}, \\ select_0(A, select_1(B^*, i)) & \text{if } c = \text{“]”} \end{cases}$$

$$match(S,i) = \begin{cases} select_1(close(B, rank_1(A,i))) & \text{if } S[i] = \text{“(”} \\ select_1(open(B, rank_1(A,i))) & \text{if } S[i] = \text{“)”} \\ select_0(close(B^*, rank_0(A,i))) & \text{if } S[i] = \text{“[”} \\ select_0(open(B^*, rank_0(A,i))) & \text{if } S[i] = \text{“]”} \end{cases}$$

$$enclose(S,i) = \begin{cases} select_1(enclose(B, rank_1(A,i))) & \text{if } S[i] = \text{“(”} \\ select_0(enclose(B^*, rank_0(A,i))) & \text{if } S[i] = \text{“[”} \end{cases}$$

With those primitives, the succinct representation of Ferres et al. [8] supports constant-time operations to navigate the embedding. Precisely, the representation supports the following operations (recall that the index v of the nodes corresponds to their order in the depth-first traversal of the spanning tree T, whereas half-edges i are just positions in S):

$source(i)$**:** the index of the node that is the source of half-edge i;

$first(v)/last(v)$**:** the first/last half-edge whose source is the node v;

$twin(i)$**:** the other half-edge corresponding to the same edge of i;

$next(i)/prev(i)$**:** the next/previous half-edge with the same source of i, in ccw order of the neighbors of v (note that there is one edge incident on v per half-edge with source v).

The operations are then supported as follows [8]:

- By Lemma 1, the first half-edge of a node v is immediately after the half-edge (u,v) coming from the parent u of v in T (except for the root of T), thus $first(v) = select_{\text{“(”}}(S,v) + 1$. The last half-edge of v is (v,u), the one returning to its parent, so $last(v) = match(S, select_{\text{“(”}}(S,v))$.
- The operation $twin(i)$ is just $match(S,i)$.
- The implementation of $next(i)$ depends on whether the half-edge i belongs to T or not. Specifically, $next(i) = i+1$ unless $S[i] = \text{“(”}$, in which case it is instead $next(i) = match(S,i) + 1$. Analogously, $prev(i) = i - 1$ unless $S[i-1] = \text{“)”}$, in which case it is $match(S, i-1)$.
- Operation $source(i)$ also depends on whether $S[i]$ is a parenthesis or a bracket. In the first case, the half-edge is in T and connects i with its parent. The source is the parent, $rank_{\text{“(”}}(S, enclose(S,i))$, if $S[i] = \text{“(”}$, or the node represented by the twin of i, $rank_{\text{“(”}}(S, match(S,i))$, if $S[i] = \text{“)”}$. On brackets (i.e., $S[i] = \text{“[”}$ or “]”), we must find the lowest node of T containing the bracket. That is, letting $j = select_1(A, rank_1(A,i))$ be the position of the last parenthesis preceding i, the source of i is $rank_{\text{“(”}}(S,j)$ if $S[j] = \text{“(”}$, and $rank_{\text{“(”}}(S, enclose(S, match(S,j)))$ otherwise.

Note that those operations suffice to implement the half-edge interface [33]. As said, each of the $2m$ positions in S identify a half-edge i. Its twin half-edge is $twin(i)$, its source node is $source(i)$, its target node is $source(twin(i))$, and its edge is the pair of positions i and $twin(i)$. The half-edge following i along

its face is $next(twin(i))$, and the one preceding it is $twin(next(i))$. We can take $last(v)$, for example, as the half-edge identifying the node v (which is the edge returning to the parent of v in T). We show in Section 4.1 how to compute $face(i)$, the identifier of the face where i belongs. The dual of operation $last(v)$ (now on brackets) then yields a half-edge identifier for each face.

With the operations described above, we can implement more complex queries in optimal time, such as listing all the incident edges of a node v in constant time per returned element, and listing all the edges or nodes bordering a face given a half-edge of the face, spending constant time per returned element: To list the half-edges with source v in ccw order, we output $i = first(v)$ and move on to $i = next(i)$, until we have listed $last(v)$. To list cw, we go from $last(v)$ to $first(v)$ with $i = prev(i)$. To list nodes instead of half-edges, we list $source(twin(i))$ instead of i. To traverse the frontier of the face of half-edge i in cw order, we list i (or $source(i)$) and move to $i = next(twin(i))$, until returning to i. We use $prev$ instead to list the frontier ccw.

Other operations, such as the degree of a node and checking if two nodes are neighbors, are not supported in constant time. For the degree of a node v, the representation supports any time in $\omega(1)$, whereas for the adjacency test of two nodes u and v, they achieve any time in $\omega(\log m)$.

Finally, Ferres et al. [8, Sec. 5.3] show that their space can be reduced to $3.8m + o(m)$ bits if G has no self-loops, and to $3.58m + o(m)$ if it also has no nodes of degree one, by switching to an entropy-bounded representation of S. The query algorithms designed to run over A, B, and B^* are shown to run transparently on this alternative representation.

Theorem 2 summarizes the results of Ferres et al., with the extension that we also obtain $3.8m + o(m)$ when the graph has no nodes of degree 1. In this case, the pair "( )" cannot appear in S. Ferres et al. [8, Sec. 5.3] analyze the entropy of S when no self-loops exist, which implies that the pair "[ ]" cannot appear in the sequence. They show that the entropy is below 3.8 bits per edge, which is reached when $m = 1.731n$. Let us now exchange the brackets and parentheses in S, so it becomes the sequence S^* of the dual graph. This dual graph has $n^* = m - n + 2$ nodes and $m^* = m$ edges, and its representation has no occurrences of the pair "[ ]". Therefore, the analysis of Ferres et al. applies to S^*, showing that its total entropy is also $3.8m^* = 3.8m$ bits (which occurs when $m^* = 1.731n^*$, i.e., when $m = 1.368n$ in the original graph). This also implies that the total entropy of S is $3.8m$, as it differs from S^* only by a renaming of symbols.

Theorem 2. *An embedding of a connected planar graph with m edges can be represented in $4m + o(m)$ bits, supporting the listing in cw or ccw order of the neighbors of a node and the nodes bordering a face in $O(1)$ time per returned node. One can also find the degree of a node in any time in $\omega(1)$, and check if two nodes are adjacent in any time in $\omega(\log m)$. The space decreases to $3.8m + o(m)$ bits if the graph has no self-loops or has no nodes of degree one, and to $3.58m + o(m)$ bits if it has none of them.*

4. Some Simple Results

As a warm-up exercise, we start with some results that derive easily from previous work [8], but that have not been clearly stated.

4.1. Nodes and Faces Connected by an Edge

We first obtain the nodes connected by a given edge, and its dual, the faces separated by the edge. This trivially answers queries (1.a) and its dual (1.b), (2.a) and its dual (2.b), (3.a) and its dual (3.b), all in constant time.

Note that our half-edge representation, as positions in S, is valid for both G and its dual G^*: by Lemma 1, the spanning tree edges of G, marked with parentheses in S, are exactly the non-spanning tree edges of G^*, and vice versa, the brackets in S are the spanning-tree edges of G^* and the non-spanning tree edges of G. We then define a new operation, $face(i)$, that returns the identifier of the face where the half-edge i belongs. This is solved analogously to $source(i)$, by exchanging the meaning of parentheses and brackets:

- If $S[i]$ is a bracket, then
  - $face(i) = rank_{\text{“[”}}(S, enclose(S, i))$ if $S[i] =$ “[”, and
  - $face(i) = rank_{\text{“[”}}(S, match(S, i))$ if $S[i] =$ “]”.
- If $S[i]$ is a parenthesis, we compute the position $j = select_0(A, rank_0(A, i))$ of the last bracket preceding i; then
  - $face(i) = rank_{\text{“[”}}(S, j)$ if $S[j] =$ “[”, and
  - $face(i) = rank_{\text{“[”}}(S, enclose(S, match(S, j)))$ otherwise.

The following result is then trivial once we can compute operations $source(\cdot)$ and $face(\cdot)$.

Lemma 2. *The representation of Theorem 2 can determine in $O(1)$ time the two nodes connected by an edge, and the two faces separated by an edge.*

Proof. Let i be any of the two half-edges of the edge. Then the two nodes corresponding to the edge are $source(i)$ and $source(match(S, i))$. The two faces are $face(i)$ and $face(match(S, i))$.

4.2. Listing Queries

Listing the faces bordering a given face (3.d) can be done as the dual of listing the neighbors of a node (3.c), by exchanging the roles of brackets and parentheses in Theorem 2. Listing the faces incident on a node (3.e) can also be done as a subproduct of Theorem 2. To list the faces ccw, we start from the half-edge $i = first(v)$, list $face(i)$ and advance to $i = next(i)$, until having processed $last(v)$. The process to list the faces cw is analogous. Analogously, given a face identifier x, we can list the nodes found in the frontier of the face (3.f). This query is not exactly the same as in Theorem 2, because there we must start from an edge bordering the desired face.

Lemma 3. *The representation of Theorem 2 suffices to list, given a node u, the faces incident on u in cw or ccw order from its parent in T, each in $O(1)$ time, or given a face x, the nodes in the frontier of x in cw or ccw order from its parent in T^*, each in $O(1)$ time.*

4.3. Counting Queries

Ferres et al. [8] count the number of edges incident on a node u (4.a) in $O(f(m))$ time using $o(m)$ bits, for any $f(m) \in \omega(1)$. A bitvector of length $O(m)$ with $O(m/f(m))$ 1s marks the nodes with degree $f(m)$ or more; this bitvector requires $O(m \log f(m)/f(m)) + o(m)$ bits in compressed form [30]. For nodes with degree below $f(m)$, they traverse the incident edges one by one; for the others, they store the degree explicitly in another bitvector using $O(m \log f(m)/f(m)) \subseteq o(m)$ further bits.

We can similarly count the number of neighboring nodes or faces, with the exception that we can reach several times the same node or face as we traverse the edges incident on a node. Thus, we need time $O(f(m) \log f(m))$ on nodes with degree over $f(m)$ in order to remove repetitions; for higher-degree nodes we store the correct number explicitly. We then obtain $O(f(m) \log f(m))$ time using $o(m)$ bits, which still achieves any time in $\omega(1)$. By building the structure on the dual of G, we can count the number of edges or nodes in the frontier of a face x, as well as the faces sharing an edge with face x (4.b).

5. Deciding if Two Nodes/Faces Share an Edge

We now focus on queries (1.c) and (1.d), which are used in standards and flagship implementations, as described in Section 2.1.

Ferres et al. [8] show how we can determine if two given nodes u and v are connected in any time $f(m) \in \omega(\log m)$. First, they check in constant time if they are connected by an edge of the spanning tree T: one must be the parent of the other. Otherwise, the nodes can be connected by an edge not in T, represented by a pair of brackets. Their idea is to mark in a bitvector $D[1..n]$ the nodes having $f(m)$ or more incident edges (see Fig. 3e for an example). The subgraph G' induced by the marked nodes, where they also eliminate self-loops and multi-edges, has $n' \leq 2m/f(m)$ nodes, because each marked node is the source of at least $f(m)$ half-edges and there are $2m$ half-edges. Since G' is planar and simple, it can have only $m' < 3n' \leq 6m/f(m)$ edges. They represent G' using adjacency lists, which use $o(m)$ bits as long as $f(m) \in \omega(\log m)$. Given two nodes u and v, if either of them is not marked in D, they simply enumerate its neighbors in time $O(f(m))$ to check for the other node. Otherwise, they map both to G' using $rank_1(D)$, and binary search the adjacency list of one of the nodes for the presence of the other, in time $O(\log m) \subseteq o(f(m))$. Bitvector D has $n' \leq 2m/f(m)$ bits set out of $n \leq m + 1$ (this second inequality holds because G is connected), and therefore it can be represented using

$(2m/f(m))\log(f(m)/2)) + O(m/f(m)) + o(m) \subseteq o(m)$ bits while answering *rank* queries in constant time [30].[6]

5.1. Description of our new representation

We will obtain any time $f(m) \in \omega(1)$ by solving the query on G' in a different way. This requires a more complex mapping, however, because now we cannot afford to represent the node identifiers of G' in explicit form within $o(m)$ bits.

In particular, we will not physically remove (all) the unmarked nodes of G to form G'; we just *paint* the unmarked nodes to signal that they can be removed. We do, instead, remove useless edges not in the spanning tree T. More precisely, we start with $G' = G$ and then:

- Paint the low-degree nodes u in gray; say the high-degree ones are black.
- Remove the edges incident on gray nodes u and not belonging to T.
- Remove self-loops and multiple edges, avoiding to remove an edge of T.

The spanning tree T' of G' is in principle identical to T. Note that the only remaining neighbors of gray nodes are connected by edges in T'. In order to obtain the desired space/time performance the gray nodes must be reduced, yet without affecting the traversal order of T' on the remaining nodes. We thus perform the following additional pruning on G' and T':

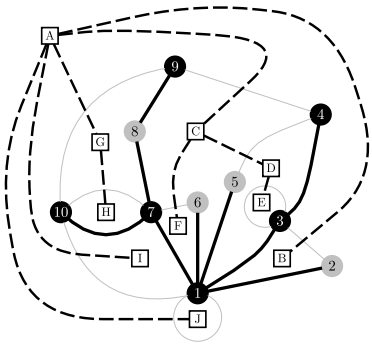
1. Consecutive gray siblings with parent v in T' are merged into one if they are also consecutive neighbors of v in G'. The children lists of the merged nodes are then concatenated.
2. A gray node with only one child that is also gray is removed, and its child is connected to its parent.
3. Gray nodes that are leaves in T' are removed.

Fig. 3a shows the graph G of Fig. 1b painted with gray and black nodes and $f(m) = 3$. Nodes 5 and 6 are merged into one node by pruning rule 1, to then remove node 2 and merged nodes 5 and 6 by rule 3. Self-loops and multiple edges are also removed, obtaining the graph G' of Fig. 3b.

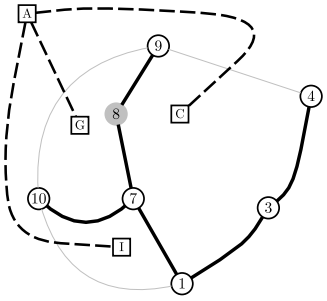
As seen, G' has $n' \leq 2m/f(m)$ black nodes. It has also gray nodes, but by rules (1–3) above, every gray node has a first or a second child in T' that is black (recall that all the edges of a gray node are in T', so rule (1) leaves no consecutive gray children of a gray node in T'). Thus, G' has at most n' gray nodes. Further, since G' is simple and contains at most $2n'$ nodes (black or gray), it contains $m' < 6n'$ edges. The length of the sequence S' representing G' is then $2m' + 4 < 12n' + 4 \leq 24m/f(m) + 4 \in O(m/f(m)) \subseteq o(m)$.

We use an additional bitvector M that identifies with 1s the black nodes of T', in preorder. Therefore, to map the identifier u of a marked node in G (i.e.,

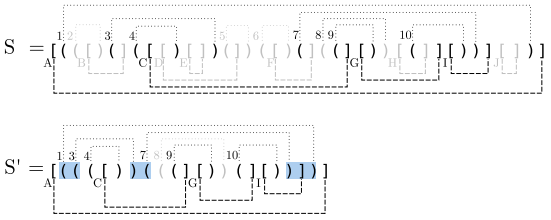
[6] They do not specify how to handle queries of the form (u, u) given that they remove self-loops. We can have a bitvector $L[1..n']$ of size $o(m)$ so that, if $D[u] = 1$, then there is an edge (u, u) in G iff $L[rank_1(D, u)] = 1$.



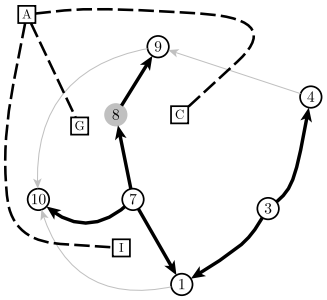
(a) Graph G of Fig. 2a with painted nodes and $f(m) = 3$.



(b) Graph G' with $f(m) = 3$



(c) Parentheses/brackets representation of G'



(d) Oriented graph G'

$$D = 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1$$

$$M = 1\ 1\ 1\ 1\ 0\ 1\ 1$$

$$U = 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0$$

$$V = 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1$$

$$W = 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0$$

$$C = 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0$$

(e) Bitvectors D, M, U, V, W and C

Figure 3: Graph G' used to support query (1.c). **(a)** Graph G of Fig. 2a before applying the pruning rules, with $f(m) = 3$. **(b)** Graph G' after applying all the pruning rules; node 8 is the only gray node that remains. **(c)** The representation of G'. On the top we mark the gray nodes and removed edges in S. On the bottom we show the final sequence S'; we also highlight in blue the range of parentheses and brackets representing the children of node 1. **(d)** Graph G' with oriented edges. **(e)** Bitvectors D, M, U, V, W, and C used to support query (1.c).

$D[u] = 1$), we first compute $u'' = rank_1(D, u)$, which is its preorder position among the black nodes of T'. We then compute $u' = select_1(M, u'')$ to obtain its node identifier in T'. Its opening parenthesis in S' is then at $select_{\text{“(”}}(S', u')$. The length of M is at most $2n' \in o(m)$. An example of the bitvector M can be seen in Fig. 3e.

5.2. Supporting queries

The key idea to determine if (black) nodes u' and v' are connected in G' is that one can orient the edges in a simple planar graph so that every node has outdegree at most 3 [34] (do not confuse with the orientation of half-edges). Once G' is oriented, determining whether (u', v') is an edge of G' requires testing whether at most 6 edges connect u' and v' (i.e., the 3 edges leaving u' and the 3 edges leaving v'). For instance, Fig. 3d shows a possible orientation for the edges of graph G' of Fig. 3b.

The problem then reduces to finding each of the out-edges of a node u' fast. We can focus on the edges not in T', because we always start checking whether (u, v) or (v, u) are in T, and the edges between black nodes of T' also belong to T. We will find in constant time the (at most) 3 brackets that represent out-edges in the top level of the substring $(\cdots)$ of S' that describes u' (we say that those brackets are *marked*). Recall that this substring contains in turn the substrings $(\cdots)$ of the children $u_1, \ldots, u_k$ of u' in T', interspersed with brackets, so "top level" means that the brackets are not inside the substring of any u_j.

We proceed as follows. We define a bitvector U where we traverse T' in preorder and, for each black node u' with k children $u_1, \ldots, u_k$, we add $k+1$ bits. The first bit is 1 iff there are marked brackets between the opening parenthesis of u' and the opening parenthesis of u_1 (or, if u' has no children, between the opening and closing parentheses of u'). For $1 < j \leq k$, the jth of the $k+1$ bits is 1 iff there are marked brackets between the closing parenthesis of u_{j-1} and the opening parenthesis of u_j. Finally, the $(k+1)$th bit is a 1 iff there are marked brackets between the closing parenthesis of u_k and that of u'. Since T' has at most $2n'$ nodes and at most $2n' - 1$ of those are children of some node, the length of U is $< 4n' \in O(m')$. A second bitvector, V, marks with 1s the first of the $k+1$ bits of each node described in U. Finally, we use a third bitvector W of length $O(m')$, where $W[i] = 1$ means that, in a left to right scan of S', the ith opening or closing bracket is marked. For example, the bitvectors U, V and W for the graph of Fig. 3d are shown in Fig. 3e.

To find the out-neighbors of a node u', we first find its area $U[p..p']$, with $p = select_1(V, u')$ and $p' = select_1(V, u' + 1) - 1$. We now find the (up to 3) positions $p_i \in [p..p']$ where $U[p_i] = 1$, with $p_i = select_1(U, rank_1(U, p-1) + i)$, stopping when $p_i > p'$. For each of those p_i, we must search the area of brackets that lie between the $(p_i - p)$th and the $(p_i - p + 1)$th children of u'. Let u' have k children. The area between the 0th and the 1st children refers to $S'[select_{\text{“(”}}(S', u') + 1..select_{\text{“(”}}(S', u' + 1) - 1]$. The area between the ith and the $(i+1)$th children, for $1 \leq i \leq k$, refers to $S'[match(S', child(S', u', i)) + 1..child(S', u', i+1)]$, where we extend the operation *child* to operate on the

parentheses of the sequence S' as follows:

$$child(S', u', i) = select_1(A', child(B', select_0(B', u'), i)),$$

where we assume S' is represented with bitvectors A', B', and $(B^*)'$. Finally, the area between the kth and the $(k+1)$th children of u' refers to $S'[match(S', child(S', u', k)) + 1..match(S', u') - 1]$.

Let $S'[q..q']$ be any such area of S', which is composed of only brackets. We map q and q' to W with $r = rank_0(A', q)$ and $r' = r + (q' - q)$, so we must enumerate the (up to 3) 1s in $W[r..r']$. Those are $r_i = select_1(W, rank_1(W, r - 1) + i)$, stopping when $r_i > r'$. The corresponding marked brackets are $q_i = r_i + (q - r)$. Each position $S'[q_i]$ corresponds to an edge that must be tested to see if it is incident on v', in constant time with query (2.a). We then analogously check if the up to 3 out-neighbors of v' are incident on u'.

We thus solve query (1.c) with $o(m)$ extra bits of space and $O(f(m))$ time, for any $f(m) \in \omega(1)$.

Lemma 4. *The representation of Theorem 2 can be enriched with $o(m)$ bits so that we can determine whether two nodes are connected in any time in $\omega(1)$.*

It is natural, if two nodes are connected, to ask for one edge connecting them. This is trivial in our case when the edge belongs to T. Otherwise, we aim to retrieve the positions $S[b..b']$ of a pair of brackets that connect our mapped nodes u' and v'. To do this, we enrich our structure with bitvector C, which tells which face identifiers of G (i.e., ranks of opening brackets) survive in G' after the pruning process. See the bottom of Fig. 3e for an example. Once we find that u' and v' are neighbors connected by the edge with opening bracket at $S'[x] =$ "[" and closing bracket at $S[x'] =$ "]", we have that the opening bracket number $b' = rank_{\text{“[”}}(S', x)$ connects them in G'. We then identify the edge in G with $b = select_1(C, b')$, and $twin(b)$, in $O(1)$ additional time. The length of bitvector C is less than m and it has less than m' 1s, thus it can be represented in $O(m \log f(m)/f(m)) + o(m) \subseteq o(m)$ bits [30].

5.3. Determining Adjacency of Faces

By exchanging the interpretation of parentheses and brackets, the same sequence S represents the dual G^* of G, where the roles of nodes and faces are exchanged. We can then use the same solution of Lemma 4 to determine whether two faces are adjacent (1.d). We do not explicitly store the sequence S^* representing G^*, since we can simulate it using S. We do, instead, build a structure on S^* analogous to the one we built on S, creating sequence $(S^*)'$ and its auxiliary bitvectors. This time, the input to the query are the ranks of the opening brackets representing both faces (i.e., node identifiers in G^*). We then solve query (1.d).

Lemma 5. *The representation of Theorem 2 can be enriched with $o(m)$ bits so that we can determine whether two faces are adjacent in any time in $\omega(1)$.*

6. Determining Incidence of a Face in a Node

We now consider the problem of, given a node u and a face x, determine whether x is incident on u (2.c). As a motivation for this query, consider a city map where nodes are street intersections, edges are the street segments between nodes, and faces are blocks. An algorithm for traversing from a given intersection towards a given block needs to determine, at any new node we reach, whether the node touches the desired face. Various traversal algorithms exist for planar graphs, with applications to city maps and network routing [35, 36].

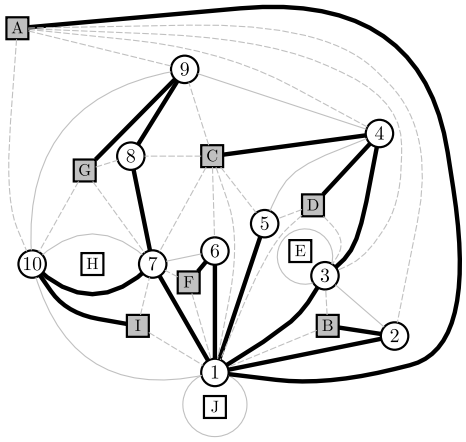
Since with Lemma 3 we can list each face incident on u in constant time, or each node bordering x in constant time, we can use a scheme combining those of Lemmas 4 and 5: If u has less than $f(m)$ neighbors, we traverse them looking for x. Otherwise, if x has less than $f(m)$ bordering nodes, we traverse them looking for u. We now show how to handle the remaining case.

We define a graph $G^{\#}$ where we add additional nodes representing *selected* faces of G, that is, those having at least $f(m)$ nodes in their frontier. The queries that are not handled by enumeration in G will become a node neighbor query on $G^{\#}$, and solved as in Lemma 4. The graph $G^{\#}$, which will have $O(m)$ edges, will not be represented directly.

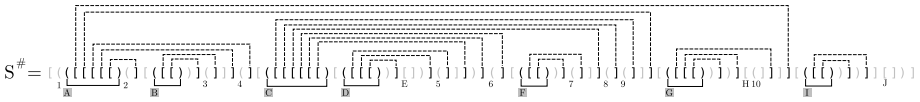
Concretely, $G^{\#}$ adds to G a new node $v(x)$ per selected face x, as well as new edges connecting $v(x)$ with all nodes in the frontier of x. Note $G^{\#}$ is planar because we can draw $v(x)$ inside the face x. There are at most $2m/f(m)$ selected faces x, because each has at least $f(m)$ edges in its frontier and each edge is in the frontier of two faces. The graph $G^{\#}$ contains $n' \leq n + 2m/f(m) = O(m)$ nodes and $m' \leq 3m$ edges (each selected face limited by j edges of G adds j new edges in $G^{\#}$, and each edge of G limits two faces).

A spanning tree $T^{\#}$ for $G^{\#}$ is built by extending the spanning tree T of G with leaves $v(x)$ that represent selected faces x. An example of $G^{\#}$ and $T^{\#}$ is shown in Fig. 4a. Consider the traversal of G that defines the spanning trees T and T^*. Let $(u, v) \notin T$ be the edge in the frontier of a selected face x where the traversal of T^* first reaches the face x, that is, when edge (y, x) is added to T^* for some face y (e.g., the edge $(4, 9)$ in Fig. 4a). Right after visiting the edge (u, v), we add a new leaf node $v(x)$, as a child of u, to $T^{\#}$ (e.g., the edge $(4, C)$ in Fig. 4a). We also add the edges (x, w) to the graph $G^{\#}$ for the other nodes w in the frontier of face x; those edges will not belong to $T^{\#}$.

To generate the sequence $S^{\#}$ representing $G^{\#}$ we traverse $T^{\#}$ starting from the edge that connects the outer face with the starting node that generates the sequence S. After that, the traversal follows the same order of T. When we reach an original selected face x (which in $G^{\#}$ is partitioned into j triangles), we will visit the edge $(u, v(x)) \in T^{\#}$ right after $(u, v) \notin T^{\#}$. We will then traverse the other $j - 1$ edges incident on $v(x)$, none of which is in $T^{\#}$, and all of which are visited for the first time because we had not entered face x before. Therefore, the "[" that represents (u, v) in $S^{\#}$ will be immediately followed by $(\ [^{j-1})$, and then the normal layout of T will follow. Those opening brackets will be closed later along the traversal. Fig. 4b shows the sequence $S^{\#}$ obtained after traversing the spanning tree $T^{\#}$ of Fig. 4a, starting from the edge $(1, A)$.



(a) Graph $G^{\#}$ with $f(m) = 3$



(b) Sequence $S^{\#}$ generated from the graph of Fig. 4a

Figure 4: Graph $G^{\#}$ used to support query (2.c). **(a)** Graph $G^{\#}$ with $f(m) = 3$. The selected faces are painted in gray and the spanning tree $T^{\#}$ is represented with thick edges. Gray edges are those not in $T^{\#}$: solid edges are in T^* and dashed edges are those created to be incident to the selected faces. **(b)** Sequence $S^{\#}$ obtained from the traversal of $T^{\#}$. The parentheses and brackets added to those in S are in black, so S is the gray subsequence of $S^{\#}$.

For instance, face C is bounded by the nodes 1, 5, 4, 9, 8, 7 and 6, which are represented by ([[[[[[ ) in Fig. 4b.

This implies that every opening bracket in S representing a selected face x, is immediately followed in $S^{\#}$ by an opening parenthesis corresponding to the node $v(x)$ we created for the face. That is, selected faces x in S are in the same order of added nodes $v(x)$ in $S^{\#}$. Further, the nodes of G are in the same order in S and $S^{\#}$. We exploit this correspondence to map nodes and faces from G to nodes of $G^{\#}$ by using the following bitvectors:

- The same bitvector $D[1..n]$ of Section 5, where $D[u] = 1$ iff node u of G has degree at least $f(m)$.
- A bitvector $E[1..m - n + 2]$, where $E[x] = 1$ iff face x of G has at least $f(m)$ nodes (or edges) in its frontier.
- A bitvector R where $R[x'] = 1$ iff node x' of $G^{\#}$ is one of the nodes $v(x)$ we added to G in order to form $G^{\#}$.

If $D[u] = 1$, we map it to node $u' = select_0(R, u)$ in $G^{\#}$ (note that all the nodes in G appear in $G^{\#}$). If $E[x] = 1$, we map it to node $x' = select_1(R, rank_1(E, x))$ (only the selected faces in G appear as new nodes in

$G^\#$). Bitvectors D, E, and R are of length $O(m)$ and have $O(m/f(m))$ 1s, so they can be represented within $O(m \log f(m)/f(m)) + o(m) \subseteq o(m)$ bits [30].

If u has degree at least $f(m)$ and x is limited by at least $f(m)$ nodes, we map node u and face x to nodes u' and x' in $G^\#$ as explained, and determine if they are neighbors. Note that, since x has at least $f(m)$ nodes in its frontier, node x' has degree at least $f(m)$. Node u' also has degree at least $f(m)$ in $G^\#$, because it had in G and it can only get further neighbors in $G^\#$. We then build on $S^\#$ the structures of Lemma 4, without explicitly representing $S^\#$, so that we map u' and x' to the reduced sequence $(S^\#)'$ and solve the query in there. This adds $o(m)$ bits of space and completes the query in any time in $\omega(1)$.

Lemma 6. *The representation of Theorem 2 can be enriched with $o(m)$ bits so that, given a node u and a face x, it answers whether u is in the frontier of x in any time in $\omega(1)$.*

7. Determining Indirect Connections

We finally consider the most complex queries, (5.a) and (5.b), which correspond to a variant of the query `ST_Touches` and its dual, as mentioned in Section 2.1.

To handle these queries, we reuse the idea of selecting a subgraph where the query cannot be solved in time $O(f(m))$ and storing a suitable speed-up structure for those cases. This time, however, the idea leads to a much higher time complexity. We later obtain constant time by relaxing the space requirement to $O(m)$ bits.

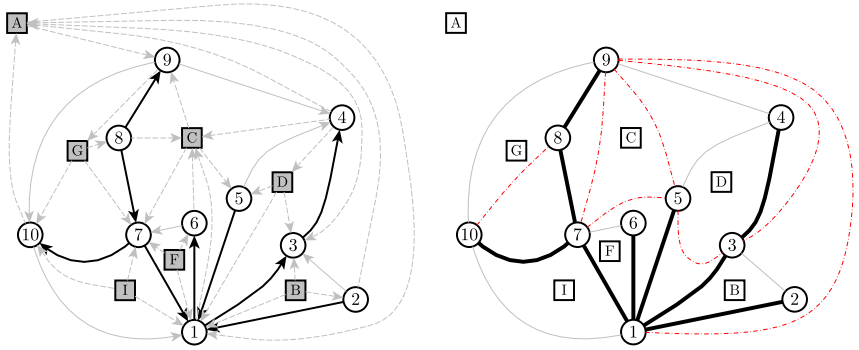
Let us first consider determining if two nodes are in the border of the same (unknown) face. Given two nodes u and v, if either has less than $f(m)$ neighbors we can traverse its incident faces one by one and, for each face x, use Lemma 6 to determine if x is incident on the other node in time $\omega(1)$. For all the pairs of nodes (u, v) where both have $f(m)$ neighbors or more, we store a binary matrix telling whether or not they lie on the same face. This requires $(2m/f(m))^2$ bits, which is $o(m)$ for any $f(m) \in \omega(\sqrt{m})$. Thus we can solve query (5.a) and, by duality, query (5.b), in any time in $\omega(\sqrt{m})$.

Lemma 7. *The representation of Theorem 2 can be enriched with $o(m)$ bits so that, given two nodes or two faces, it answers in $O(f(m))$ time whether they share a face or a node, respectively, for any $f(m) \in \omega(\sqrt{m})$.*

If we want to know the identity of the shared face (or, respectively, node), this can be stored in the matrix, which now requires $O((m/f(m))^2 \log m)$ bits. We can then reach any time in $\omega(\sqrt{m \log m})$.

7.1. Constant Times with $O(m)$ Bits of Space

As explained, those operations requiring $\omega(1)$ time in Table 1 automatically become $O(1)$ if we use $f(m) \in O(1)$. In exchange, the space becomes $(4 + \epsilon)m$ bits for any desired constant $\epsilon > 0$. We now show that queries (5.a) and (5.b)



(a) Graph $G^{\#}$ with oriented edges

(b) Graph $G^{\$}$

Figure 5: Graphs $G^{\#}$ and $G^{\$}$ used to support query (5.a). **(a)** Graph $G^{\#}$. Solid edges represent edges of the original graph G (Fig. 1b), and dashed ones represent edges connecting faces with their bordering nodes. The spanning tree T of the original graph is in black. **(b)** Graph $G^{\$}$. Red edges represent the new edges connecting nodes linked by two out-edges of the node induced by their common face in $G^{\#}$.

can also be solved in $O(1)$ time if we raise the space to $O(m)$ bits.[7] Let us focus on the query of Lemma 7 (5.a); we then obtain (5.b) by duality. Our solution is inspired in a (non-compact) data structure for constant-time bounded shortest distance queries on planar graphs [39].

Consider the graph $G^{\#}$ of Section 6, with the following changes:

- We start with a copy of G and then remove self-loops and multiple edges (not those that belong to T). Self-loops and multiple edges are irrelevant for the query (5.a).
- We select *all* the remaining faces of G.
- We represent $G^{\#}$ explicitly (so we use $O(m)$ bits of space).
- We orient the edges of $G^{\#}$ as in Section 5. Since $G^{\#}$ is simple, the out-degree of every node can be made at most 3.

If nodes u and v are both in the frontier of some face mapped to node x in $G^{\#}$, then the following configurations of the edge orientations are possible in $G^{\#}$: $u \to x \to v$, $u \to x \leftarrow v$, $u \leftarrow x \leftarrow v$, and $u \leftarrow x \to v$. The first three are easily verified in constant time using the structures of Section 6. For example, for $u \to x \to v$ we check the (up to) 3 out-neighbors x_i of u and, for those x_i that are faces in G (which is verified in bitvector R), we check their (up to) 3 out-neighbors $v_{i,j}$, to see if $v_{i,j} = v$.

The difficult case is $u \leftarrow x \to v$, because we should start our quest from the unknown node x. Fortunately, inside the face x represents in G, there can be

[7] In the conference version [28], we incorrectly conjectured that this problem was intersection-hard [37, 38] even using $O(m \log m)$ bits of space.

at most 3 nodes that are out-neighbors of x in $G^{\#}$. We then create another extended version of the original graph G (still without self-loops and multiple edges), which we call $G^{\$}$, where we additionally connect those 3 nodes inside each face x, thereby drawing a triangle inside the face (thus $G^{\$}$ is planar). Thus, there is a configuration $u \leftarrow x \rightarrow v$ iff (u, v) is one of those edges of the triangles we have added. Fig. 5b shows an example for the graph $G^{\$}$, based on the oriented edges of Fig. 5a. For instance, the cases $9 \leftarrow A \rightarrow 1$ and $5 \leftarrow C \rightarrow 7$ cause the insertion of the edges $(1, 9)$ and $(5, 7)$, as shown in Fig. 5b. It is clear that $G^{\$}$ has $O(m)$ edges and that we can define a spanning tree $T^{\$}$ for it that is identical to that of T, letting all the added edges be not in $T^{\$}$. We then represent $G^{\$}$ as in Lemma 4 (with $f(m) \in O(1)$) and verify the configuration $u \leftarrow x \rightarrow v$ by asking if u and v share an edge in $G^{\$}$. Note that this query can return an edge that belongs to G, but in this case it is also true that u and v border the same face. By also considering duality, we have the following result.

Lemma 8. *The representation of Theorem 2 can be enriched with $O(m)$ bits so that, given two nodes or two faces, it answers in $O(1)$ time whether they share a face or a node, respectively.*

In the graph $G^{\#}$ we add inside each face of G one new node, as well as one new edge per edge limiting the face. As each edge of G limits two faces, we add in total $m - n + 2$ new nodes and $2m$ new edges, so $G^{\#}$ has up to $m + 2$ nodes and $3m$ edges. Its Turán's representation then requires $12m$ bits. In addition we need its bitvector R, which adds other $m + 2$ bits. The impact of the graph $G^{\$}$, on the other hand, can be reduced by (1) maintaining only the edges of the spanning tree and (2) adding the triangles only on the faces x limited by at least $f(m) \in O(1)$ nodes. This retains the constant time and limits the edges to $n + O(m/f(m))$. In this way, $G^{\$}$ requires $(2n + \epsilon)n$ further bits, for any desired constant $\epsilon > 0$. In total, the representation of Lemma 8 requires at most $(15 + \epsilon)m$ further bits of space. This can be reduced to $(13 + \epsilon)m$ by representing $G^{\$}$ using the techniques of Section 5, so that it shares the main structure with G and the added edges are seen only on the $O(m/f(m))$ faces that have at least $f(m)$ neighbors.

8. Conclusions

We built on a recent extension [8] of Turán's representation [6] for plane embeddings so as to support a rich set of topological queries within succinct space, $4m + o(m)$ bits for an m-edge embedding. Though it exceeds the asymptotically optimal space of $3.58m + o(m)$ bits, this representation is particularly attractive to handle the topological model because it regards the graph and its dual symmetrically, thereby enabling a number of queries relating nodes, edges, and faces. We actually reach the $3.58m + o(m)$ bits of space if the graph has no self-loops and no nodes of degree one.

Starting with an improved solution to determine if two nodes are neighbors, we exploit analogies and duality to support most of the operations in any time

in $\omega(1)$. We then relax our space requirements to $O(m)$ bits, showing that in this case we can represent variants of the graph that allow us support all the desired queries (on the original graph) in $O(1)$ time.

An interesting challenge is whether we can support bounded distance queries (bounded meaning that only distances up to some constant k are distinguished) efficiently and within $O(m)$ bits of space. For $k = 2$, a relatively obvious variant of Lemma 7 yields any time in $\omega(\sqrt{m})$ within $4m+o(m)$ bits of space. We cannot use an analogous to the $O(m)$-bits construction of Lemma 8 to obtain constant time, however, because the resulting graph $G^{\$}$ could be non-planar. Kowalik and Kurowski [39] show that this query can be solved in constant time using $O(m \log m)$ bits, that is, with a classical non-compact representation. They use the same idea of orienting the edges and are left with the hard subproblem of the configuration $u \leftarrow x \rightarrow v$, for which we built $G^{\$}$. They handle this case by adding the edges (u, v) explicitly, which in general make the graph non-planar. They show, however, that the resulting graph is the union of a constant number of planar graphs, which can then be queried one by one. Our problem to obtain $O(m)$ bits from this idea is how to track the node identifiers across those planar graphs, which can have very different spanning trees.

References

[1] G. D. Lozzo, A. D'Angelo, F. Frati, On planar greedy drawings of 3-connected planar graphs, Discrete Computational Geometry 63 (1) (2020) 114–157.

[2] G. Navarro, Compact Data Structures: A practical approach, Cambridge University Press, Cambridge, UK, 2016.

[3] L. Castelli Aleardi, O. Devillers, G. Schaeffer, Succinct representation of triangulations with a boundary, in: Proc. 9th International Conference on Algorithms and Data Structures (WADS), 2005, pp. 134–145.

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

[5] P. Bose, E. Y. Chen, M. He, A. Maheshwari, P. Morin, Succinct geometric indexes supporting point location queries, ACM Transactions on Algorithms 8 (2) (2012) 10:1–10:26.

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

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

[8] L. Ferres, J. Fuentes-Sepúlveda, T. Gagie, M. He, G. Navarro, Fast and compact planar embeddings, Computational Geometry Theory and Applications (2020) 101630.

[9] M. Worboys, M. Duckham, GIS: A Computing Perspective, 2nd Edition, CRC Press, Boca Raton, FL, USA, 2004.

[10] M. O. Scholl, Spatial Databases with Application to GIS, Morgan Kaufmann, San Francisco, CA, USA, 2002.

[11] ISO/IEC 13249-3:2016. Information technology – Database languages – SQL multimedia and application packages – Part 3: Spatial, Tech. rep. (2016).

[12] OpenGIS Simple Features Specification For SQL, Tech. rep. (1999).
URL `https://www.ogc.org/standards/sfs`

[13] E. Clementini, J. Sharma, M. J. Egenhofer, Modelling topological spatial relations: Strategies for query processing, Computers & Graphics 18 (6) (1994) 815–822.

[14] V. Gaede, O. Günther, Multidimensional access methods, ACM Comput. Surv. 30 (2) (1998) 170–231.

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

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

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

[19] 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: ICALP, LNCS 1443, 1998, pp. 118–129.

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

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

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

[23] W. Schnyder, Embedding planar graphs on the grid, in: Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1990, pp. 138–148.

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

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

[26] W. T. Tutte, A census of planar triangulations, Canadian Journal of Mathematics 14 (1962) 21–38.

[27] K. Yamanaka, S.-I. Nakano, A compact encoding of plane triangulations with efficient query supports, Inform. Process. Lett. 110 (18-19) (2010) 803–809.

[28] J. Fuentes-Sepúlveda, G. Navarro, D. Seco, Implementing the topological model succinctly, in: Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE), 2019, pp. 499–512.

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

[30] R. Raman, V. Raman, S. Satti, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, ACM Transactions on Algorithms 3 (4) (2007).

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

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

[33] D. E. Muller, F. P. Preparata, Finding the intersection of two convex polyhedra, Theoretical Computer Science 7 (2) (1978) 217–236.

[34] M. Chrobak, D. Eppstein, Planar orientations with low out-degree and compaction of adjacency matrices, Theoretical Computer Science 86 (2) (1991) 243–266.

[35] E. Kranakis, H. Singh, J. Urrutia, Compass routing on geometric networks, in: Proc. 11th Canadian Conference on Computational Geometry (CCCG), 1999.

[36] P. Bose, P. Morin, I. Stojmenovic, J. Urrutia, Routing with guaranteed delivery in ad hoc wireless networks, Wireless Networks 7 (6) (2001) 609–616.

[37] H. Cohen, E. Porat, Fast set intersection and two-patterns matching, Theoretical Computer Science 411 (40-42) (2010) 3795–3800.

[38] M. Patrascu, L. Roditty, Distance oracles beyond the Thorup-Zwick bound, SIAM Journal on Computing 43 (1) (2014) 300–311.

[39] L. Kowalik, M. Kurowski, Oracles for bounded-length shortest paths in planar graphs, ACM Transactions on Algorithms 2 (3) (2006) 335–363.