Implementing the Topological Model Succinctly*

José Fuentes-Sepúlveda[1], Gonzalo Navarro[1,3], and Diego Seco[2,3]

[1] Department of Computer Science, University of Chile, Chile
[2] Computer Science Department, Universidad de Concepción, Chile
[3] IMFD — Millennium Institute for Foundational Research on Data, Chile
{jfuentes,gnavarro}@dcc.uchile.cl, dseco@udec.cl

Abstract. We show that the topological model, a semantically rich standard to represent GIS data, can be encoded succinctly while efficiently answering a number of topology-related queries. We build on recent succinct planar graph representations so as to encode a model with m edges within $4m + o(m)$ bits and answer various queries relating nodes, edges, and faces in $o(\log \log m)$ time, or any time in $\omega(\log m)$ for a few complex ones.

1 Introduction

Low-cost sensors are generating huge volumes of geographically referenced data, which are valuable in applications such as urban planning, smart-cities, self-driving cars, disaster response, and many others. Geographic Information Systems (GIS) that enable *capture, modeling, manipulation, retrieval, analysis and presentation* [13] of such data are thus gaining research attention. GIS models can be classified at different levels. For example, on the conceptual level, entity- and field-based approaches exist, whereas on the logical level, vector and raster are the most popular models. In this work we focus on the representation of the geometry of a collection of vector objects, such as points, lines, and polygons.

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 [11]. 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

* Funded by the Millennium Institute for Foundational Research on Data (IMFD), Chile and by European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 690941 (project BIRDS). The first author received funding from Conicyt Fondecyt grant 3170534. The last author received funding from Conicyt Fondecyt regular grant 1170497.

model is more efficient to answer topological queries, which are usually expensive, and thus it is gaining popularity in spatial databases such as Oracle Spatial.

In this work we focus on those topological queries where this model stands out, and show that they can be efficiently answered within very little space. We build on recent results on connected planar graphs [6] in order to provide a succinct-space representation of the topological model ($4m + o(m)$ bits, where m is the number of edges) that efficiently support a rich set of topological queries (most of them in $o(\log\log m)$ time), which include those defined in current standards and flagship implementations. Our main technical result is a new $O(\frac{\log\log m}{\log\log\log m})$ time algorithm to determine if two nodes are neighbors; then many other results are derived via analogous structures and exploiting duality. These results improve upon those of the planar graph representation on which we build [6] (see also that article for a wider coverage of previous work).

2 The Topological Model and Our Contribution

The topological model represents a planar subdivision into adjacent polygons. Hereinafter, we will refer to these polygons as *faces*. A face is represented 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. This eliminates redundancy in the stored geometries and also reduces inconsistencies. In Fig. 1, faces are named with capital letters, A to H, A being the outer face. Face F is defined by the sequence of nodes $\langle 1, 8, 7, 6\rangle$, and edge $(6, 7)$ is shared by faces D and F. Note, however, that a pair of nodes is insufficient in general to name an edge, because multiple edges may exist between two nodes.

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 [1] defines a basic set of primitive operations for the model, which are also implemented in flagship database systems[4]. 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.

Topological queries can be also solved using the geometries, but such approach is computationally very expensive. We propose instead an approach in

[4] `http://postgis.net/docs/Topology.html`

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

Table 1: The queries we consider on the topological model and the best results within succinct space. Our (sometimes partial) contributions are in boldface.

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

Table 1 lists a set of topological queries we consider on the topological model. They 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 `intersects` (1.d and 1.f), `GetNodeEdges` (3.c), and `ST_GetFaceEdges` (3.e).

A preliminary result essentially hinted in previous work [6], Lemma 2, sorts out a number of simple queries (all [123].[ab]) in constant time. Our main result is Lemma 3, which shows how to determine if two given nodes are connected by an edge (1.c) in time $O(\frac{\log\log m}{\log\log\log m})$, adding only $o(m)$ bits to the main structure. The same procedure on the dual graph, Lemma 4, determines in the same time if two given faces share an edge (1.d, a variant of the standard query `intersects`). Another consequence of Lemma 2 is Lemma 5, which extends previous work [6] 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, 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 previous results [6] that count the edges incident on a node (4.a) in any time in $\omega(1)$ to count nodes, edges, or faces incident on a node or bordering a face (4.b).

Finally, our solution to determine if a given node is in the frontier of a given face (2.c) is costlier, in $\omega(\log m)$, and that to determine if two given nodes border the same face (1.e) or if two given faces share some node (1.f, a variant of query `intersects`) cost even more, in $\omega(\sqrt{m}\log m)$. The last two solutions build on Lemmas 3 and 4, and we conjecture that their times cannot be easily improved.

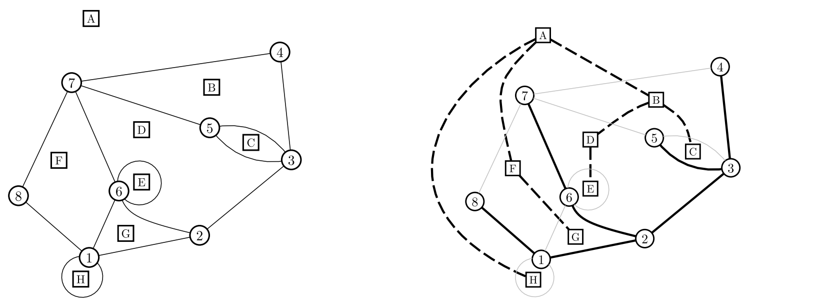
3 Succinct Data Structures

3.1 Sequences and Parentheses

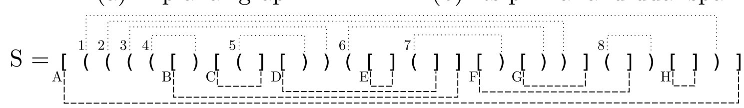
Given a sequence $S[1..n]$ defined over an alphabet of size σ, the operation $rank_a(S,i)$ returns the number of occurrences of the symbol a in the prefix $S[1..i]$, and the operation $select_a(S,i)$ returns the position in S of the ith occurrence of the symbol a. For binary alphabets, $\sigma = 2$, S can be stored in $n + o(n)$ bits supporting *rank* and *select* in $O(1)$ time [3]. 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* [10]. For $\sigma = O(\text{polylog } n)$, S can be represented in $n \log \sigma + o(n)$ bits, still supporting $O(1)$-time *rank* and *select* [5]. Binary sequences can be used to represent balanced parentheses sequences. 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 $j \leq i \leq close(S,j)$. If S is used to represent an ordered tree, we find the parent of the node represented by the opening parenthesis $S[i]$ as $parent(S,i) = enclose(S,i)$. The sequence S can be represented in $n + o(n)$ bits, supporting *open*, *close* and *enclose* in $O(1)$ time [7]. Such representation can be extended to represent k superimposed balanced parenthesis sequences in the same space and time complexities, for any constant k [8, Sec. 7.3].

3.2 Planar Graphs

A planar graph is a graph that can be drawn in the plane without crossing edges. The topology of a specific drawing of a planar graph in the plane is called a *planar embedding*. We use planar embeddings to represent topological models. In particular, we use Turán's representation [12], which can represent any planar embedding of m edges in $4m$ bits. Ferres *et al.* [6] extended Turán's representation with $o(m)$ extra bits in order to support fast navigation, providing the simple and efficient representation of planar embeddings we build on.



(a) A planar graph (b) Its primal and dual spanning trees



(c) The sequence of parentheses and brackets encoding the planar graph

Fig. 1: Example of the succinct planar graph representation of Ferres *et al.* [6].

Given a planar 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 [2]. The edges of T^* correspond to the edges in the dual graph crossing edges in $G \setminus T$. Fig. 1b shows a primal (thick continuous edges) and a dual (thick dashed edges) spanning trees for the planar graph of Fig. 1a. Lemma 1 states a key observation: a depth-first traversal of T induces a depth-first traversal in T^*.

Lemma 1 ([6]). *Consider any planar embedding of a planar graph G, any spanning tree T of G and the complementary spanning tree T^* of the dual of G. Suppose we perform a depth-first traversal of T starting from any node on the outer face of G and always process the edges incident to the node v we are visiting in counter-clockwise order. At the root, we arbitrarily choose an incidence of the outer face in the root and start from the last edge of the incidence in counter-clockwise order; at any other node, we start from the edge immediately after the one to that node's parent. Then each edge not in T corresponds to the next edge we cross in a depth-first traversal of T^*.*

Here, an incidence of the outer face in the root means a place where the root and the outer face are in contact. For instance, in Fig. 1b, the traversal can start at edge $(1,1)$, $(1,2)$, or $(1,8)$, taking node 1 as the root of the spanning tree.

The compact representation [12, 6] 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 "[" otherwise. 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 by an opening parenthesis or bracket, and also by its corresponding closing parenthesis or bracket. Fig. 1c shows the sequence S for the planar graph of Fig. 1b, starting the traversal at the edge $(1, 2)$. Observe that the parentheses of S encode the balanced-parentheses representation of T and the brackets the balanced-parentheses representation of the dual spanning tree T^*. In the succinct representation of Ferres *et al.* [6], 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.

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^*, the succinct representation of Ferres *et al.* [6] supports constant-time operations to navigate the embedding. Precisely, the succinct representation supports $first(v)/last(v)$ (the position in S of the first/last visited edge of the node v), $mate(i)$ (the position in S of the other occurrence of the ith visited edge), $next(i)/prev(i)$ (the position of the next/previous edge after visiting the ith edge of a node v in counter-clockwise order), and $node(i)$ (the index of the source node when visiting the ith edge). Notice that the index v of the nodes corresponds to their order in the depth-first traversal of the spanning tree T, whereas the index i of a visited edge is just a position in S (i.e., each edge is visited twice). According to Lemma 1, the first visited edge of a node v is the edge immediately after the edge to the parent of v in T (except for the root of T), thus $first(v) = select_1(A, select_0(B, v)) + 1$. The implementation of $last(v)$ is similar. The operation $mate(i)$ is transformed to an *open* operation if $S[i]$ is a closing parenthesis or bracket (i.e., $\hat{B}[rank_{A[i]}(A, i)] = 1$): $mate(i) = select_{A[i]}(A, open(\hat{B}, rank_{A[i]}(A, i)))$, or to a *close* operation otherwise (i.e., $\hat{B}[rank_{A[i]}(A, i)] = 0$): $mate(i) = select_{A[i]}(A, close(\hat{B}, rank_{A[i]}(A, i)))$, where $\hat{B} = B$ if $A[i] = 1$ and $\hat{B} = B^*$ if $A[i] = 0$. The implementation of $next(i)$ depends on whether the ith visited edge belongs to T or not. Specifically, $next(i) = i + 1$ unless i is an opening parenthesis (i.e., $A[i] = 1$ and $B[rank_1(A, i)] = 0$), in which case it is instead $next(i) = mate(i) + 1$; $prev(i)$ is analogous. Operation $node(i)$ also depends on whether $S[i]$ is a parenthesis or a bracket. In the first case ($A[i] = 1$), $node(i) = rank_0(B, enclose(B, rank_1(A, i)))$ if $B[rank_1(A, i)] = 0$ and $node(i) = rank_0(B, open(B, rank_1(A, i)))$ otherwise. On brackets ($A[i] = 0$), $node(i) = rank_0(B, rank_1(A, i))$ if $B[rank_1(A, i)] = 0$, otherwise $node(i) = rank_0(B, enclose(B, open(B, rank_1(A, i))))$.

With the operations described above, we can implement more complex operations in optimal time, such as listing all the incident edges (and the corresponding neighbor nodes) of a node v in constant time per returned element, and listing all the edges or nodes bordering a face given an edge of the face, spending constant time per returned element. 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, $degree(v)$, the representation supports any time in $\omega(1)$, whereas for the adjacency test of two nodes u and v, $neighbor(u, v)$, they achieve any time in $\omega(\log m)$. In Section 4 we give an $O(\frac{\log\log m}{\log\log\log m})$-time solution for $neighbor(u, v)$, and introduce several other operations in Section 5. Theorem 1 summarizes the results of Ferres *et al.*

Theorem 1 ([6]). *An embedding of a connected planar graph with m edges can be represented in $4m + o(m)$ bits, supporting the listing in clockwise or counter-clockwise order of the neighbors of a node and the nodes bordering a face in $O(1)$ time per returned node. Additionally, one can find the degree of a node in any time in $\omega(1)$, and check the adjacency of two nodes in any time in $\omega(\log m)$.*

3.3 Obtaining the Nodes and Faces of an Edge

Before presenting our main results, we show how to obtain the nodes connected by a given edge, and its dual, the faces separated by the edge. These results are somewhat implicit in the preceding work [6], but we prefer to present them clearly here. They trivially answer 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 edge representation, as positions in S, is valid for both G and G^* (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^*). The two nodes corresponding to an edge i in G are obtaining analogously to operation $node(i)$: if i is a parenthesis ($A[i] = 1$), then $p \leftarrow rank_1(A, i)$ is its position in B. If it is closing ($B[p] = 1$), we set $p \leftarrow open(B, p)$. The two nodes are then $rank_0(B, p)$ and $rank_0(B, enclose(B, p))$. On brackets ($A[i] = 0$), we find two positions in B, $p_1 \leftarrow rank_1(A, i)$ and $p_2 \leftarrow rank_1(A, mate(i))$. If any is a closing parenthesis ($B[p_1] = 1$ or $B[p_2] = 1$), we take its parent, $p_1 \leftarrow enclose(B, open(B, p_1))$ and/or $p_2 \leftarrow enclose(B, open(B, p_2))$. Finally, the answers are the resulting nodes, $rank_0(B, p_1)$ and $rank_0(B, p_2)$. The identifiers of the two faces divided by the edge are obtained almost with the same formulas, replacing the meaning of 0 and 1 in A, and using B^* instead of B.

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

4 Determining if Two Nodes are Connected

Ferres *et al.* [6] 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 $f(m)$ neighbors or more. 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 at least $f(m)$ edges are incident on each marked node and each of the m edges can be incident on at most 2 nodes. Since G' is planar and simple, it can have only

$m' < 3n' \leq 6m/f(m)$ edges.[5] 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 $O(\log m) = o(f(m))$. 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 $(2m/f(m))\log(f(m)/2)) + O(m/f(m)) + o(m) = o(m)$ bits while answering *rank* queries in constant time [10].

In order to improve this time, we apply the idea for more than one level. This requires a more complex mapping, however, because only in the last level we can afford to represent the node identifiers in explicit form. The intermediate graphs, where we cannot afford to store a renumbering of nodes, will be represented using an extension of the idea of a sequence of parentheses and brackets, in order to maintain the order of the node identifiers.

Concretely, let us call $G_0 = G$ the original graph of $n_0 = n$ nodes and $m_0 = m$ edges, and $S_0[1..2(m_0+2)] = S[1..2(m+2)]$ its representation using parentheses and brackets. A bitvector $D_0[1..n_0]$ marks which nodes of G_0 belong to $G_1 = G'$. When a certain node u is removed from G_0 to form G_1, we also remove all its edges, which are of two kinds:

- Not belonging to the spanning tree T. These are represented by a pair of brackets $[\cdots]$, opening and closing, which are simply removed from S_0 in order to form S_1.
- Belonging to the spanning tree T. These are *implicit* in the parent-child relation induced by the parentheses. By removing the parentheses of u we remove the node, but this implicitly makes the children of u to be interpreted as new children of v, the parent of u in T. To avoid this misinterpretation, we replace the two parentheses of u by angles: $(\cdots)$ becomes $\langle\cdots\rangle$.

In order to obtain the desired space/time performance, the angles must be reduced to the minimum necessary. In particular, we enforce the following rules:

1. Elements under consecutive angles are grouped inside a single one: $\langle X\rangle\,\langle Y\rangle$ becomes $\langle X\,Y\rangle$.
2. An angle containing only one angle is simplified: $\langle\,\langle X\rangle\,\rangle$ becomes $\langle X\rangle$.
3. Angles containing nothing disappear: $\langle\,\rangle$ is removed.

As seen, G_1 contains $n_1 \leq 2m_0/f(m)$ nodes and, since it contains no multiple edges, $m_1 < 8m_0/f(m)$ edges. Its representation, S_1, then contains $2n_1$ parentheses and $2(m_1 - n_1 + 2)$ brackets. It also contains angles, but by rules 2 and 3, each angle pair contains at least one distinct maximal pair of parentheses[6], and

[5] In fact, they do not specify how to handle queries of the form (u, u) given that they remove self-loops. They could leave one self-loop around each node that has one or more, and the bound would be $m' \leq 4n' \leq 8m/f(m)$. We do this in our extension.

[6] Not brackets: a top-level bracket inside angles would correspond to an edge inciding on the removed node, and thus must have been removed when forming S_1.

thus there are at most $2n_1$ angles. The length of S_1 is then $2(n_1 + m_1 + 2) < 20m_0/f(m) + 4$.

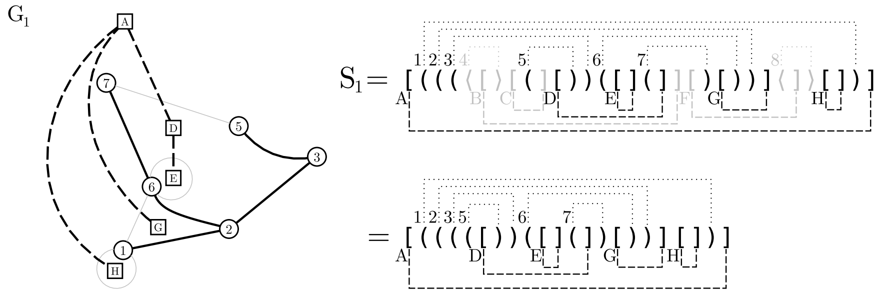
We represent S_1 using an array $A_1[1..2(n_1 + m_1 + 2)]$ over an alphabet of size 3 (to distinguish brackets = 0, parentheses = 1, and angles = 2), and the projected balanced sequences $B[1..2n_1]$ of parentheses, $B^*[1..2(m_1 - n_1 + 2)]$ of brackets, and $B^-[1..2n_1]$ of angles. We can then support constant-time *rank* and *select* on A_1 using $o(m_1)$ extra bits [5], and *open*, *close*, and *enclose* on B, B^*, and B^- also using $o(m_1)$ extra bits. Thus we can support operations $mate(\cdot)$ and $node(\cdot)$ on S_1 in constant time, just as described in Section 3.2.

In order to determine if u_1 and v_1 are neighbors in G_1 we may visit the neighbors of u_1: We sequentially traverse the area between the parentheses of u_1, $S_1[p..p'] = (\cdots)$, starting from $p \leftarrow p + 1$, analogously as the neighbor traversal described in Section 3.2. If we see an opening parenthesis, $S[p] =$ "(", we skip it with $p \leftarrow mate(p) + 1$ because we are only checking for neighbors via brackets (we already know that the nodes are not neighbors via edges in T). If we see an opening angle, $S[p] =$ "$\langle$", we also skip it with $p \leftarrow mate(p) + 1$ because this encloses eliminated nodes and no top-level brackets of u_1 can be enclosed in those angles, as explained. If we see a bracket, $S[p] =$ "[" or $S[p] =$ "]", we find its mate, $j = mate(i)$, then the node containing it, $v = node(j)$, and check if $v = v_1$. Note that a bracket cannot lead us to an eliminated node, because brackets of eliminated nodes were effectively removed from S_1. This procedure takes time proportional to the number of neighbors of u_1 in G_1: although we may spend time in traversing angles, by rule 1 above, every angle we skip is followed by a non-angle or by the final closing parenthesis $S[p'] =$ ")".

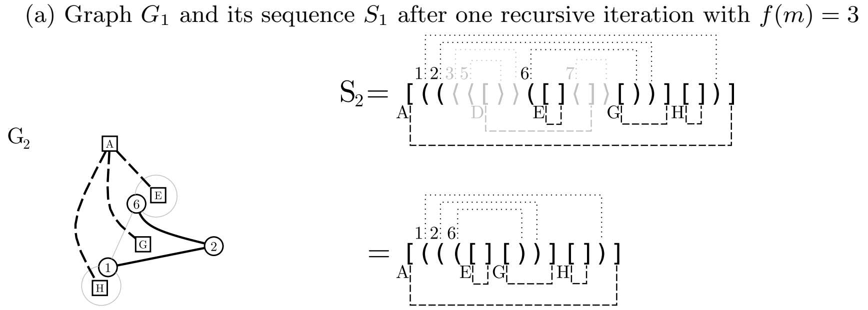
Our construction does not end in G_1, however. We repeat the construction process in G_1, so that G_2 is the subgraph of G_1 induced by its nodes with $f(m)$ incident edges or more. We continue for $k(m)$ iterations, obtaining the sequences $S_0, \ldots, S_{k(m)-1}$ and the graph $G_{k(m)}$. In $G_{k(m)}$, we store the neighbors of each node in a perfect hash table. Fig. 2 shows the resulting graphs G_1 and G_2 after applying two recursive calls over the planar graph of Fig. 1.

The algorithm to determine if $u_0 = u$ and $v_0 = v$ are neighbors, once we check that none is a child of the other in T, is then as follows. If $D_0[u_0] = 0$, we traverse the neighbors of u_0 as described (the top-level sequence, S_0, does not contain angles, though), to see if v_0 is mentioned. This takes time $O(f(m))$ because u_0 has less than $f(m)$ neighbors. Otherwise, if $D_0[v_0] = 0$, we proceed analogously with v_0, in time $O(f(m))$. Otherwise, both nodes are mapped to G_1, to $u_1 = rank_1(D_0, u_0)$ and $v_1 = rank_1(D_0, v_0)$, and we continue similarly with u_1 and v_1 in G_1. If, after $k(m)$ steps, we arrive at $G_{k(m)}$ without determining if they are neighbors, we look for $v_{k(m)}$ in the perfect hash table of the neighbors of $u_{k(m)}$, in constant time. Overall, the query time is $O(k(m) + f(m))$.

As for the space, G_i has $n_i \leq 2m_{i-1}/f(m)$ nodes and $m_i < 4n_i \leq 8m_{i-1}/f(m)$ edges (because G_i has no multiple edges for all $i > 0$), and thus $m_i < m \cdot (8/f(m))^i$ and $n_i \leq (1/4)m \cdot (8/f(m))^i$. The length of S_i is then less than $2(n_i + m_i + 2) < (5/2)m \cdot (8/f(m))^i + 4$. The previous expression, summed over all $1 \leq i < k(m)$, yields a total length for all $S_1, \ldots, S_{k(m)-1}$ below



(a) Graph G_1 and its sequence S_1 after one recursive iteration with $f(m) = 3$



(b) Graph G_2 and its sequence S_2 after two recursive iterations with $f(m) = 3$

Fig. 2: Intermediate planar graphs and their sequences to support the operation *neighbor*$(\cdot,\cdot)$. Symbols in light-gray represent deleted elements.

$20m/(f(m)-8)+4k(m) = O(m/f(m)+k(m))$, for any $f(m) \geq 9$. Since the S_i have constant-size alphabets, they can be represented within $O(m/f(m)+k(m))$ bits, with the constant-time support for *rank*, *select*, *open*, and *close*. On the other hand, the explicit representation of $G_{k(m)}$ requires $O(m_{k(m)} \log m) = O(m \log m \cdot (8/f(m))^{k(m)})$ bits. For all this space to be $o(m)$ we need that $k(m) = o(m)$, $f(m) = \omega(1)$, and $(f(m)/8)^{k(m)} = \omega(\log m)$.

The choice $f(m) = k(m) = \max(9, \frac{(1+\epsilon)\log\log m}{\log\log\log m})$, for any constant $\epsilon > 0$, yields a time complexity in $O(\frac{\log\log m}{\log\log\log m})$ and an extra space in $o(m)$ bits.

If we wish to retrieve the positions $S[b..b']$ of a pair of brackets that connect u and v, when the edge does not trivially belong to T, we enrich our structure with bitvectors $C_0, \ldots, C_{k(m)-1}$, where $C_i[1..m_i - n_i + 2]$ tells which face identifiers (i.e., ranks of opening brackets) survive in G_{i+1}. Once we find, in some G_i, that u_i and v_i are neighbors connected by the edge $S_i[p..p'] = [\cdots]$, we have that the opening bracket number $b_i = rank_{\text{``[''}}(S_i, p) = rank_0(B^*, rank_0(A_i, p))$ connects them in G_i. We then identify the edge in G_{i-1} with $b_{i-1} = select_1(C_{i-1}, b_i)$, and continue upwards until finding the answers, $b = b_0$ and $b' = mate(b)$, all in $O(k(m))$ additional time.

The lengths of all bitvectors, for $i > 0$, is $|D_i| + |C_i| = m_i + 2$, so they add up to $o(m)$. For D_0 and C_0, note that they have n_1 and m_1, both in $O(m/f(m))$, 1s out of $n \leq m+1$ or $m-n+2 \leq m$, respectively. Therefore, they can be represented in $O(m \log(f(m))/f(m)) + o(m)$ bits [10]. We thus solve query (1.c).

Lemma 3. *The representation of Theorem 1 can be enriched with $o(m)$ bits so that we can determine whether two nodes are connected in time $O(\frac{\log\log m}{\log\log\log m})$.*

5 Other Results Exploiting Analogies and Duality

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 3 to determine whether two faces are adjacent (1.d). We do not explicitly store the sequence S^* representing G^*, since we can operate it using S. Instead, we build a structure on S^* analogous to the one we built on S, creating sequences $S_1^*, \ldots, S_{k(m)-1}^*$, $D_0^*, \ldots, D_{k(m)-1}^*$, $C_0^*, \ldots, C_{k(m)-1}^*$, and the final explicit dual graph $G_{k(m)}^*$, so as to determine, within the same space and time complexities, whether two faces of G share an edge, and retrieve one of these edges. This time, the input to the query are the ranks of the opening brackets representing both faces (i.e., node identifiers in G^*).

Lemma 4. *The representation of Theorem 1 can be enriched with $o(m)$ bits so that we can determine whether two faces are adjacent in time $O(\frac{\log\log m}{\log\log\log m})$.*

Listing related nodes or faces. 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 1. Listing the faces incident on a node (3.e) can also be done as a subproduct of Theorem 1. For each edge e incident on u, obtained in counter-clockwise order, we obtain the faces e divides using Lemma 2. This lists all the faces incident on u, in counter-clockwise order, with the only particularity that each face is listed twice, consecutively. 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 1, because there we must start from an edge bordering the desired face.

Lemma 5. *The representation of Theorem 1 suffices to list, given a node u, the faces incident on u in counter-clockwise order from its parent in T, each in $O(1)$ time, or given a face x, the nodes in the frontier of x in clockwise order from its parent in T^*, each in $O(1)$ time.*

Determining incidence of a face in a node. Given a node u and a face x, the problem is to determine whether x is incident on u (2.c). Since with Lemma 5 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 3 and 4: 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. Otherwise, we search for (u, x) in a perfect hash table where we store all the faces y (bounded by $f(m)$ or more nodes) incident on nodes v (having $f(m)$ or more neighbors).

To see that this hash table contains $O(m/f(m))$ elements, consider the bipartite planar graph $G^+(V^+, E^+)$ where $V^+ = V \cup F$ (F being the faces of our original graph $G(V, E)$) and $E^+ = \{(u, x), x \in F$ is incident on $u \in V$ in $G\}$. G^+ is planar because it can easily be drawn from an embedding of G, by placing the nodes $x \in F \subseteq V^+$ inside the face x of G and drawing its edges without having them cut. Note that the nodes $u \in V$ preserve their degree in G^+, whereas the degree of nodes $x \in F$ is the number of edges bordering their corresponding face in G. Therefore G^+ has $n^+ = |V| + |F| = m + 2$ nodes (as per Euler's formula $|F| = m - n + 2$) and $m^+ = 2m$ edges (one per edge limiting each face, so each edge of G contributes twice). If we remove from G^+ all the nodes (of either type) connected with less than $f(m)$ neighbors, and remove multiple edges, each surviving edge corresponds precisely with an entry (v, y) of our perfect hash table. By the same argument used in Section 4, at most $4m/f(m)$ nodes survive and, since the reduced graph has no multiple edges, at most $4 \cdot (4m/f(m)) = O(m/f(m))$ edges survive. Hence, we obtain extra space $o(m)$ by choosing any $f(m) \in \omega(\log m)$.

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

Counting neighbors. Ferres *et al.* [6] count the number of edges incident on a node u (4.a) in $O(f(m))$ time using $O(m \log f(m)/f(m))$ bits. For nodes with degree below $f(m)$, they traverse the neighbors; for the others, they store the degree explicitly. Neighboring nodes or faces can be counted similarly, except that we can reach several times the same node or face. Thus, we need time $O(f(m) \log f(m))$ on nodes with degree below $f(m)$ 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 $O(m \log f(m)/f(m))$ bits, which still achieves any time in $\omega(1)$ in $o(m)$ bits. By building the structure on the dual of G, we count the number of edges, nodes, or faces in the frontier of a face x (4.b).

6 More Expensive Solutions

We left for the end other solutions that are likely impractical compared to using brute force, but that nevertheless have theoretical value. These more expensive solutions also encompass some more sophisticated queries not included in Table 1.

Determining if two nodes border the same 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 $\omega(\log m)$. 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 share a face. This requires $(2m/f(m))^2$ bits, which is $o(m)$ for any $f(m) = \omega(\sqrt{m})$. Thus we can solve query (1.e) and, by duality, query (1.f), in any time in $\omega(\sqrt{m}\log m)$.

Lemma 7. *The representation of Theorem 1 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 $f(m) \in \omega(\sqrt{m}\log m)$.*

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 $\omega(\sqrt{m}\log^{3/2} m)$.

Determining if two nodes/faces are connected with the same node/face. Given two nodes u and v, if either has less than $f(m)$ neighbors we can traverse its neighbors w and, using Lemma 3, determine if w is a neighbor of v. This takes $O(f(m) \cdot \frac{\log\log m}{\log\log\log m})$ time. 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 share a neighbor. By duality, we can tell if two faces share edges with the same face.

Lemma 8. *The representation of Theorem 1 can be enriched with $o(m)$ bits so that, given two nodes or two faces, it answers in $O(f(m))$ time whether they are connected with a node or a face, respectively, for any $f(m) \in \omega(\sqrt{m} \cdot \frac{\log\log m}{\log\log\log m})$.*

As before, to know the identity of the shared node or face, the time raises to $f(m) \in \omega(\sqrt{m} \cdot \frac{\sqrt{\log m}\log\log m}{\log\log\log m})$.

7 Conclusions

We built on a recent extension [6] of Turán's representation [12] for planar graphs to support queries on the topological model in succinct space. Starting with an improved solution to determine if two nodes are neighbors, we exploit analogies and duality to support a broad set of operations, most in time $O(\frac{\log\log m}{\log\log\log m})$.

One remaining challenge is the the support for the standard query `intersects` (whether two given faces touch each other). If this is interpreted as the faces sharing an edge, then this is query (1.d), which we solve in time $O(\frac{\log\log m}{\log\log\log m})$. If, instead, it suffices with the faces sharing a node, this is query (1.f), which we solve in any time in $\omega(\sqrt{m}\log m)$. We conjecture that this second interpretation is intersection-hard [4, 9], and thus no significant improvement can be expected even if using non-compact space.

References

1. ISO/IEC 13249-3:2016. Information technology – Database languages – SQL multimedia and application packages – Part 3: Spatial. Tech. rep. (2016)
2. Biggs, N.: Spanning trees of dual graphs. J. Comb. Theory B **11**(2), 127–131 (1971)

3. Clark, D.R.: Compact PAT Trees. Ph.D. thesis, Univ. of Waterloo, Canada (1996)
4. Cohen, H., Porat, E.: Fast set intersection and two-patterns matching. Theor. Comp. Sci. **411**(40-42), 3795–3800 (2010)
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. **3**(2), article 20 (2007)
6. Ferres, L., Fuentes-Sepúlveda, J., Gagie, T., He, M., Navarro, G.: Fast and compact planar embeddings. Comput. Geom. Theor. App. (2018), to appear. Available at `https://arxiv.org/abs/1610.00130`. Preliminary version in *Proc. WADS'17*
7. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. **31**(3), 762–776 (2001)
8. Navarro, G.: Compact Data Structures: A practical approach. CUP (2016)
9. Patrascu, M., Roditty, L.: Distance oracles beyond the Thorup-Zwick bound. SIAM J. Comput. **43**(1), 300–311 (2014)
10. Raman, R., Raman, V., Satti, S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Alg. **3**(4) (2007)
11. Scholl, M.O.: Spatial Databases with Application to GIS. MK (2002)
12. Turán, G.: On the succinct representation of graphs. Discr. Appl. Math. **8**(3), 289–294 (1984)
13. Worboys, M., Duckham, M.: GIS: A Computing Perspective. CRC, 2nd edn. (2004)