SCARST: Schnyder Compact And Regularity Sensitive Triangulation Data Structure

Luca Castelli Aleardi, Olivier Devillers

SCARST: Schnyder Compact And Regularity Sensitive Triangulation Data Structure

Luca Castelli Aleardi* Olivier Devillers†

Abstract

We consider the problem of designing fast and compact solutions for representing the connectivity information of triangle meshes. While traditional data structures (Half-Edge, Corner Table) are fast and user-friendly, they tend to be memory-expensive. On the other hand, compression schemes, while meeting information-theory lower bounds, lack the ability to facilitate rapid navigation within the mesh structure. Compact representations provide an advantageous balance for representing large meshes, enabling a judicious compromise between memory consumption and fast implementation of navigational operations. We propose new representations that are sensitive to the regularity of the graph while still having worst case guarantees. For all our data structures we have both an interesting storage cost, typically 2 or 3 r.p.v. (references/vertex) in the case of very regular triangulations and provable upper bounds in the worst case scenario. One of our solution has a worst case cost of 3.33 r.p.v. which is currently the best-known bound improving the previous 4 r.p.v. [Castelli et al. 2018]. In terms of running time, our representations are slightly slower (factor 1.5 to 4) than classical data structures. In our experiments we compare on various meshes runtime and memory performance of our representations with those of the most efficient existing solutions.

1 Introduction

Planar graphs and 3D surface meshes (which are among the most commonly used representations for discrete approximations of surfaces), have become ubiquitous in various fields such as geometric modeling, computer graphics, and computational geometry. Over the last two decades, the widespread adoption and growing complexity of such graphs have sparked numerous research efforts aimed at efficiently processing and representing these objects. The main focus is on reducing the memory cost for storing the *connectivity*[1] information of the mesh, which is significantly more expensive to store when compared to geometric data.

Mesh representations. Indeed, various hierarchies of representations exist for storing and manipulating meshes, each offering a different balance between ease of use and memory consumption. The choice of which structure to use should rely on the specific needs of the application, on the amount of computational and memory resources and on the type and combinatorial structure of the input graph. For instance, depending on the application one could seek to achieve very good compression rates on average in the case of regular real-world 3D meshes, or to provide worst-case upper bounds for dealing with irregular graphs. Simplicity and efficient navigation are essential features of mesh representations. A straightforward implementation allows for quicker

*LIX, Ecole Polytechnique, Palaiseau, France amturing@lix.polytechnique.fr

†Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, FranceOlivier.Devillers@inria.fr

[1]The connectivity describes the incidence relations between the vertices, edges and faces of the mesh.

processing of common navigational and access operators, improving overall performance. Mesh representations should ideally *preserve the original vertex ordering* of the input graph. This has a twofold advantage. Firstly, several algorithms rely on geometric and other additional data associated with vertices that cannot be arbitrarily rearranged (e.g. when utilizing geometric data structures for proximity queries). On the other hand, many real-world meshes exhibit excellent *vertex locality*, where neighboring vertices tend to have close indices on average. Preserving the vertex ordering in the case of good vertex locality significantly enhances the runtime efficiency by reducing cache misses. The storage cost of a mesh data structure can be measured in *bits per vertex* (b.p.v.) or in *references per vertex* (r.p.v.). A reference can be either a pointer or a vertex (or edge) number of logarithmic size (in the number of vertices). In our results, for short we speak of x r.p.v., but for the detailed results we precise $x \log_2 n + y$ b.p.v..

1.1 Related Works: a Hierarchy of Structures

Explicit data structures. The simplest and most common way of representing the combinatorial structure of surface meshes is to store all incidence relations between vertices, edges and faces. For this reason conventional mesh representations [**?**, **?**, **?**, **?**] tend to be highly redundant in order to allow fast local navigation and data access. They admit pointer-based and array-based implementations, which typically require between 13 and 19 r.p.v. to represent a triangulation.
Compression schemes. From the compression point of view there exist efficient schemes [**?**, **?**, **?**] allowing us to encode triangle meshes with a few bits per vertex or matching asymptotically the information-theory lower bound of 3.245 b.p.v. [**?**]. Sometimes it is possible to exploit the *regularity*[2] of the mesh and to get even better compression rates [**?**]. However, it is worth noting that such encodings are primarily suitable for storage on disks or transmission over networks, as they do not efficiently support local navigation. To access the data, it is necessary to decompress the entire mesh first.
Succinct data structures. On the theoretical side, there exist data structures [**?**, **?**] (called *succinct*) that attain the asymptotic optimal threshold of 3.245 b.p.v. and offer $O(1)$ time navigation within a mesh. While being extremely compact "for n large enough", they suffer from a significant limitation that renders them impractical for real-world use. Their storage bounds typically include a sublinear term $O(n\frac{\log\log n}{\log n})$, which becomes negligible only for very huge datasets.
Compact data structures. Compact data structures decrease the storage cost of explicit structures by reducing the number of stored incidence relations between mesh elements. This can be achieved by performing a convenient re-ordering and/or pairing of the faces and edges and exploiting some combinatorial decompositions. Their storage costs range between 2 and 7 r.p.v. (refer to Table 1): some representations [**?**] exhibit impressive compression rates for very regular 3D meshes (high values of d_6) but achieves bad results for irregular graphs. Because of the simplicity of their implementation and the fact that, in most cases, they preserve the original vertex ordering, compact data structures provide very fast navigation. For instance the computation of the vertex degree requires typically some tens of nanoseconds per query, so they are only 1.5 to 4 times slower than explicit representations (these comparisons assume that the whole graph fits in memory).
Ultra-compact data structures. Combining vertex re-ordering approaches with efficient encodings of variable length references, it is possible to design *ultra compact data structures* whose compression rates are close to the information theory bounds. For instance, the `BELR` [**?**] (*Bit efficient LR)* and `Zipper` [**?**] data structures combine `LR` re-ordering approach with a better

[2] A commonly used measure of mesh regularity is the proportion of degree 6 vertices, denoted by d_6.

Compact data structure	storage cost (r.p.v.)					navigation time		preserves vertex ordering
	lower bound	average (tested meshes) 3D meshes	delaunay	random	upper bound	edge/face navigation	`target` operator	
2D catalogs [**?**]	-	-	-	-	7.67	$O(1)$	$O(1)$	yes
Star vertices [**?**]	7	7	7	7	7	$O(d^\circ)$	$O(1)$	yes
sorted `TRIPOD` [**?**]	6	6	6	6	6	$O(1)$	$O(d^\circ)$	yes
SOT [**?**]	6	6	6	6	6	$O(1)$	$O(d^\circ)$	yes
ESQ [**?**]	4	-	-	-	4.8	$O(1)$	$O(d^\circ)$	yes
SQUAD [**?**]	4	4.14	4.31	4.59	6	$O(1)$	$O(d^\circ)$	yes
LR [**?**]	2	2.27	3.04	6.15	> 12	$O(1)$	$O(1)$	no
Prop. 1=[**?**, Thm 2]	6	6	6	6	6	$O(1)$	$O(d^\circ)$	yes
[**?**, Thm 4]	5	5	5	5	5	$O(1)$	$O(d^\circ)$	yes
[**?**, Thm 5]	4	4	4	4	4	$O(1)$	$O(d^\circ)$	no
SCARST-OS, Thm 2	3	**3**	**3**	**3**	**3**	$O(d_M+d^\circ)$	$O(d^\circ)$	yes
SCARST-OT, Thm 3	3	**3.34**	**3.71**	**3.93**	**5**	$O(1)$	$O(d^\circ)$	yes
SCARST-RS, Cor 5	2	**2**	**2**	**2**	**2**	$O(d_M+d^\circ)$	$O(d^\circ)$	no
SCARST-RT, Thm 4	2	**2.26**	**2.54**	**2.65**	**3.67**	$O(1)$	$O(d^\circ)$	no
SCARST-WC, Thm 6	3	**3.03**	**3.08**	**3.04**	**3.33**	$O(1)$	$O(d^\circ)$	no

Table 1: Comparison between existing compact data structures for triangle meshes. Storage bounds are expressed in terms of *references per vertex* (r.p.v.) and hold for the case of genus 0 closed surfaces. The degree of the accessed vertex is denoted d°, while d_M denotes the maximum vertex degree of its neighbors. We report in columns 3-5 the average storage costs for both synthetic and real-world datasets obtained with our implementations of `SCARST` and `SQUAD` and LR representations. Column 3 reports the average costs obtained on the tested regular 3D meshes (for which $d_6 \geq 40\%$).

encoding of references, decreasing the storage bounds to some tens of bits per vertex. `PEMB` representation enriches the Turan encoding of planar graphs to support constant time navigation and gets even better compression rates: encoding a triangulation requires about 17 b.p.v.. Such storage [**?**, **?**] performances are achieved at the cost of a much slower navigation. For instance, `BELR` is about five times slower than `LR`, and the `PEMB` [**?**] representation is between 15 and 200 times slower than a plain graph data structure depending on the type of query. This makes such structures advantageous only when the whole graph does not fit in main memory with alternative solutions. These structures does not preserve the input vertex ordering.

1.2 Contributions.

We propose SCARST (Schnyder Compact And Regularity Sensitive Triangulation) a new compact data structure (built on the starting representation described in [**?**]-Thm 2) for representing the connectivity of triangle meshes that improve the previous existing worst case bounds. SCARST has several variants, ensuring (or not) a constant navigation time in the worst case, preserving (or not) vertex ordering, being sensitive (or not) to the mesh regularity. All variants have a worst case storage guarantee. Table 1 provide a range for the memory storage and an experimental value, averaging on many practical meshes. More precisely, we design the representations below (they are all provided with upper bounds):

- SCARST-OS (Ordered preserved, constant Storage) and SCARST-OT (Ordered preserved, constant navigation Time) are two variants that preserve the original vertex ordering. SCARST-OS has a cost of 3 r.p.v. and its navigation time is regularity sensitive, while SCARST-OT get constant time to the price of some extra storage that is regularity sensitive. The storage of SCARST-OT is almost always below 4 r.p.v. for both real-world meshes and synthetic graphs, while having an upper bound of 5 r.p.v. in the worst case. On most datasets our data structure achieves better results when compared to `SQUAD` [**?**], the best known compact representation preserving vertex ordering, see Fig. 14.

- SCARST-RS (Reordered, constant Storage) and SCARST-RT (Reordered, constant navigation Time) are two variants that makes use of vertex reordering to further reduce the storage bounds: experimentally SCARST-RT storage is very close to 2 r.p.v. for regular meshes, while we can guarantee a worst-case upper bound of 3.67 r.p.v.. Its compression rates are comparable to the ones of `LR` [**?**], and in some cases even better.

- SCARST-WC (best Worst-Case) is a variant (using vertex reordering) providing $O(1)$ navigation and a 3.33 r.p.v. worst-case upper bound, while being close to 3 r.p.v. on regular meshes. As far as we know this is the best existing upper bound on the storage requirements for such compact data structures. Previous solutions either have larger provable upper bounds in the worst case [**?**, **?**, **?**, **?**] or achieve poor compression rates for several classes of irregular graphs [**?**] (see Fig. 14).

We performed experimental evaluations on a broad spectrum of 2D and 3D triangle meshes, comparing our storage and runtime performances to previous solutions. Our results confirm the practical interest of our representations: while being much more compact than explicit representations, our data structures are still fast, being only between 2× and 4× slower in practice. We also establish new experimental evaluations on the storage bounds of existing compact representations [**?**, **?**]: our results confirm that previous existing solutions [**?**] achieve very good compression rates for regular meshes (with many low degree vertices), but may require large storage in the case of irregular and random graphs.

2 Preliminaries

2.1 Navigation Interface.

We adopt the navigation interface of the *Winged-edge* data structure [**?**] that supports efficient local navigation in a surface mesh (see Fig. 1).

Given an edge $\vec{e} = (v, w)$ (arbitrarily oriented) the incidences between edges and vertices can be performed through the six operators below: $\texttt{Source}(\vec{e})$ and $\texttt{Target}(\vec{e})$ are the two incident vertices, $\texttt{LFront}(\vec{e})$ and $\texttt{LBack}(\vec{e})$ are the two other edges of the triangle to the left of $\vec{e}$, and $\texttt{RFront}(\vec{e})$ and $\texttt{RBack}(\vec{e})$ are edges of the right triangle (see Fig. 1).

It is worth noting that edges have arbitrary orientations and each edge is represented only once (with one of the two possible orientations), conversely to what occurs in the case of the Half-edge structure which stores pairs of half-edges (oriented in opposite direction).

2.2 Schnyder Woods for Planar Triangulations.

Given a planar triangulation (or a genus 0 triangle mesh) with a distinguished *root* face (V_r, V_b, V_g), one can compute a *Schnyder wood* [**?**], which is a partition of the internal edges into three sets

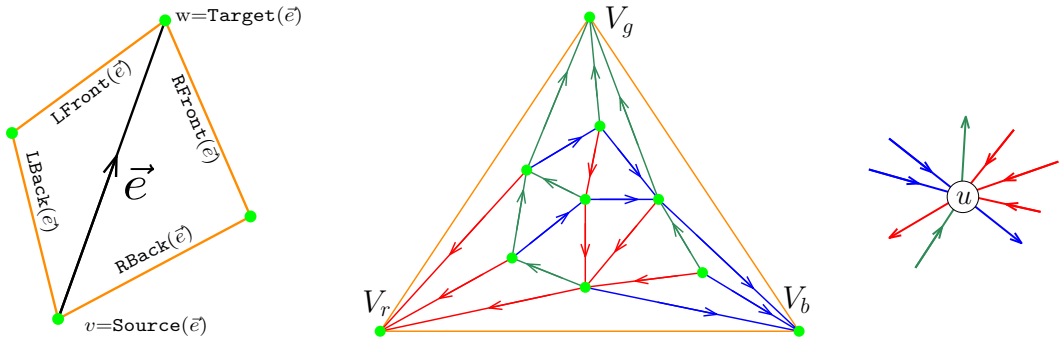

Figure 1: Navigation operators (left) Schnyder wood (center) and Schnyder rule (right)

which are trees T_r, T_b and T_g spanning all internal vertices, rooted at the vertices V_r, V_b and V_g respectively (see Fig. 1-center).

Internal edges are oriented and colored (with 3 colors) in such a way that each inner vertex has exactly three outgoing incident edges (one in each color). We will denote by T_c the tree consisting of edges having color c (where $c \in \{r, b, g\}$), with the convention: $r{+}1 = b$; $b{+}1 = g$; $g{+}1 = r$. The local Schnyder rules state that turning in counterclockwise (ccw) around an internal vertex u we must encounter: the outgoing red edge, a group of incoming green edges if any, the outgoing blue edge, a group of incoming red edges if any, the outgoing green edge, a group of incoming blue edges if any (see Fig. 1).

A Schnyder wood can be computed in linear time by a simple shelling process [**?**]. A given triangulation may admit several distinct Schnyder woods, but there is a unique *minimal* Schnyder wood which is guaranteed to have no *ccw triangles* (triangles whose edges are oriented in counter-counterclockwise direction).

2.3 Castelli&Devillers' Starting Data Structure

For sake of completeness we review the approach adopted by Castelli and Devillers: Thm. 2 in [**?**, Thm. 2] describes a simple and fast compact data structure that represents planar[3] triangulations with 6 b.p.v. Even if we start from the same point as [**?**], we proceed in a different manner to reduce redundancies allowing better performances and sensitivity to the regularity of the triangulation. The complexity proofs need new arguments.

Overview of Thm. 2 in [?]. We first compute an arbitrary Schnyder wood and complete its coloring. We assign colors and orientations to the edges on the root face: edges incident to V_r are red (oriented toward V_r) and edge (V_g, V_b) is blue and oriented toward V_b. Then each edge is uniquely identified by its origin and its color (see Fig. 2).

In the sequel an edge will be denoted $\overrightarrow{v_c}$ for the edge of source v and color $c \in \{r, b, g\}$. Then the idea is for each edge $\overrightarrow{v_c}$ to store the orientation of the four neighboring edges and the source of the two front edges.

For each vertex v and each color c we store three boolean informations: $\texttt{Leaf}[v, c]$ which is true if v is a leaf in T_c, $\texttt{LOrient}[v, c]$ which is true if $\texttt{LFront}(\overrightarrow{v_c})$ is oriented towards $\texttt{Target}(\overrightarrow{v_c})$, and $\texttt{ROrient}[v, c]$ similar on the right. We also store two vertex numbers $\texttt{Source}(\texttt{LFront}(\overrightarrow{v_c}))$ and $\texttt{Source}(\texttt{RFront}(\overrightarrow{v_c}))$. When the orientations are known the colors can be determined using

[3] As done in [**?**, **?**, **?**], we deal in this work with planar graphs (genus 0 meshes): in Section 5 we briefly discuss how to address the case of higher genus graphs.

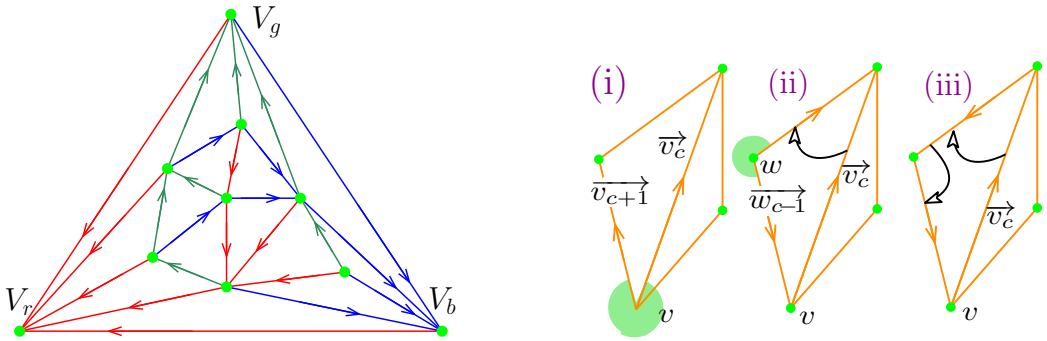

Figure 2: Right:Schnyder wood, coloring external face. Left: Retrieving left back

the Schnyder rule and the back neighbors can be deduced (see Fig. 2). The left back neighbor $\texttt{LBack}(\overrightarrow{v_c})$ of $\overrightarrow{v_c}$ can be retrieved as: (i) $\overrightarrow{v_{c+1}}$, if $\texttt{Leaf}[v, c{-}1]$ is false; (ii) $\overrightarrow{w_{c-1}}$ with $w = \texttt{LFront}(\overrightarrow{v_c})$, if $\texttt{Leaf}[v, c{-}1]$ and $\texttt{LOrient}[v, c]$ are true; (iii) $\texttt{LFront}(\texttt{LFront}(\overrightarrow{v_c}))$, otherwise. In a similar way one can retrieve $\texttt{RBack}(\overrightarrow{v_c})$.

Storage and runtime complexity. This structure can be implemented using 6 tables of size n containing vertex numbers: $\mathbf{R}_\mathrm{L}$ (with $\mathbf{R}_\mathrm{L}[v] = \texttt{Source}(\texttt{LFront}(\overrightarrow{v_r}))$) , $\mathbf{R}_\mathrm{R}$, $\mathbf{B}_\mathrm{L}$, $\mathbf{B}_\mathrm{R}$, $\mathbf{G}_\mathrm{L}$, and $\mathbf{G}_\mathrm{R}$, and 9 tables of size n containing booleans $\texttt{Leaf}[., r]$, $\texttt{LOrient}[., r]$, $\texttt{ROrient}[., r]$, $\texttt{Leaf}[., b]$, $\texttt{LOrient}[., b]$, $\texttt{ROrient}[., b]$, $\texttt{Leaf}[., g]$, $\texttt{LOrient}[., g]$, and $\texttt{ROrient}[., g]$. The implementation of the operators $\texttt{Source}(\overrightarrow{v_c})$, $\texttt{LFront}(\overrightarrow{v_c})$, $\texttt{RFront}(\overrightarrow{v_c})$, $\texttt{LBack}(\overrightarrow{v_c})$, and $\texttt{RBack}(\overrightarrow{v_c})$ requires a constant number of accesses to these different tables, while answering $\texttt{Target}(\overrightarrow{v_c})$ needs a number of access bounded by its degree. We get:

Proposition 1 ([?]-Thm 2)**.** *The above structure represents a triangulation of n vertices with $O(1)$ time access to neighboring edges using $9n + 6n \log_2 n$ bits.*

3 SCARST Data Structures

We name our structure SCARST for *Schnyder Compact And Regularity Sensitive Triangulation*. In all cases we use minimal Schnyder woods (with no ccw triangles). We propose several trade-offs between runtime and storage cost. The runtime and/or the storage cost are sensitive to the degree of vertices, i.e. the mesh regularity.

Overview of our approach. We exploit the fact that in a planar (or bounded genus) graph the number of high degree vertices is small: so we distinguish between low and high degree vertices. For an edge whose target vertex w has low indegree in one color (at most three in our implementation), we store only one reference to one of its left (or right) neighboring edges. Retrieving the remaining neighbors (whose reference is not stored) can be performed by turning around w in cw (or ccw) direction, which takes a constant number of steps. For a vertex w having high indegree in color c we store a few *extra* skipping references (depicted as black arrows in our pictures), which allow us to jump and to efficiently traverse the siblings of an edge of color c incoming at w. In our implementation we store the fourth edge turning around w in cw (or ccw) direction. In practice the number of such extra references is small and from counting arguments involving Schnyder woods we derive worst case upper bounds.

Extra references and address indirection. We distinguish between an *essential* reference that exist for all vertices and an *extra* reference that exist only for some (high degree) vertices.

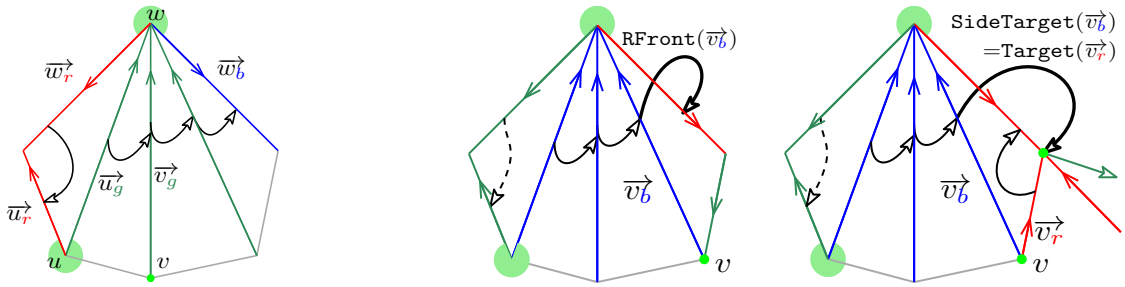

Figure 3: SCARST-OS. Left: green navigation. Right: blue navigation

Retrieving extra references can be done through address indirection. We use the memory word allocated for an essential reference as a pointer to the auxiliary table: in this table we have two corresponding entries, one for the extra reference and the second one for the lost essential reference. So, for storing each extra reference we consume two integer numbers.

3.1 SCARST-OS, a Regularity Sensitive Navigation Time Data Structure

This first structure is pretty similar to the one of Section 2.3, we just remove tables $\mathbf{R}_{\mathrm{R}}$, $\mathbf{B}_{\mathrm{L}}$, $\mathbf{G}_{\mathrm{L}}$. Compared to Section 2.3, $\texttt{Source}(\texttt{RFront}(\overrightarrow{v_r}))$, $\texttt{Source}(\texttt{LFront}(\overrightarrow{v_b}))$, and $\texttt{Source}(\texttt{LFront}(\overrightarrow{v_g}))$ are not stored and we have to retrieve the missing information.

Retrieving $\texttt{LFront}(\overrightarrow{v_g})$ (see Fig. 3). The right front neighbor of $\overrightarrow{v_g}$ is either blue or green, and is explicitly stored. Thus we can iterate the right front operator until we find a blue edge $\overrightarrow{w_b}$. Notice that w is the target of $\overrightarrow{v_g}$. Then we can access to $\texttt{LFront}(\overrightarrow{w_r})$ this edge must be red, since there is no ccw triangle. Let $\overrightarrow{u_r} = \texttt{LFront}(\overrightarrow{w_r})$. Either $u = v$ and the searched edge $\texttt{LFront}(\overrightarrow{v_g})$ is $\overrightarrow{w_r}$. Either $u \neq v$, we can iterate the right front operator starting from $\overrightarrow{u_g}$ until we find $\overrightarrow{v_g}$ and the searched edge $\texttt{LFront}(\overrightarrow{v_g})$ is the previous edge in the iteration.

Retrieving $\texttt{LFront}(\overrightarrow{v_b})$ (see Fig. 3). The right front operator for blue edges is quite similar to the one for green edges. In the case of green edges, we need to access the left front of red edges which is explicitly stored in Table $\mathbf{R}_{\mathrm{L}}$. In the case of blue edges, we need the left front of green edges which is not explicitly stored but retrieved at the previous paragraph.

Actually, for an edge $\overrightarrow{v_b}$ whose right back and front edges are red, $\texttt{RFront}(\overrightarrow{v_g})$ can be obtained as $\texttt{LFront}(\overrightarrow{v_r})$. The space reserved for $\texttt{Source}(\texttt{RFront}(\overrightarrow{v_g}))$ in Table $\mathbf{B}_{\mathrm{R}}$ can be reused to store $\texttt{Target}(\overrightarrow{v_r})$ instead. Thus for a blue edge whose both right edges are red, we have an operator $\texttt{SideTarget}(\overrightarrow{v_b})$. This operator will be used later at Sections 3.3 and 3.4.

Retrieving $\texttt{RFront}(\overrightarrow{v_r})$ (see Fig. 4). For red edges, the situation is symmetrical. The left front neighbor is accessible and the right front one must be retrieved. We access it by turning clockwise around $w = \texttt{Target}(\overrightarrow{v_r})$ until we reach $\overrightarrow{u_r}$ whose right front neighbor is a blue edge $\overrightarrow{w_r}$. Now w is reach and we can continue to turn clockwise around w from $\overrightarrow{w_g}$ until we find again $\overrightarrow{v_r}$, the last seen edge is $\texttt{RFront}(\overrightarrow{v_r})$.

Storage and access time complexity With respect to Section 2.3, we save the storage of 3 vertex numbers per vertex. Unfortunately, this structure does not have a constant time access to the neighboring iterators since we pay a cost proportional to $\mathrm{d}^{\circ}_c(w)$, the indegree of color c of $w = \texttt{Target}(\overrightarrow{v_c})$ to access to one of the two front neighbors of $\overrightarrow{v_c}$.

This structure uses 3 tables of size n containing vertex numbers: $\mathbf{R}_{\mathrm{L}}$ $\mathbf{B}_{\mathrm{R}}$, and $\mathbf{G}_{\mathrm{R}}$. and the 9 tables of size n containing booleans $\texttt{Leaf}[.,c]$, $\texttt{LOrient}[.,c]$, and $\texttt{ROrient}[.,c]$ for $c \in$

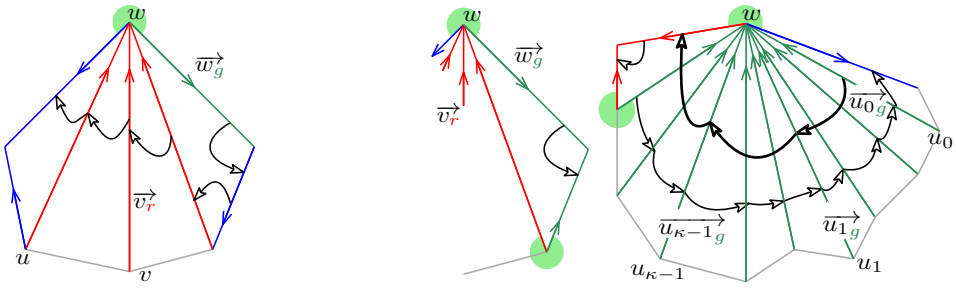

Figure 4: Left: Red navigation to the right. Right: Green navigation (high degree).

$\{r, b, g\}$. Operators $\texttt{Source}(\overrightarrow{v_c})$, $\texttt{Target}(\overrightarrow{v_c})$, $\texttt{LFront}(\overrightarrow{v_c})$, $\texttt{RFront}(\overrightarrow{v_c})$, $\texttt{LBack}(\overrightarrow{v_c})$, and $\texttt{RBack}(\overrightarrow{v_c})$ are implemented as a number of access to these different tables smaller than the degree of $\texttt{Target}(\overrightarrow{v_c})$ (plus $\mathrm{d}^\circ_g(\texttt{Target}(\overrightarrow{w_g}))$ when $c = b$).

Theorem 2. SCARST-OS *represents a triangulation of n vertices with access time to neighboring edges linear in the degree with storage cost $9n + 3n\log_2 n$ bits.*

3.2 SCARST-OT, a Regularity Sensitive Storage Cost Data Structure

Computing $\texttt{LFront}(\overrightarrow{v_g})$ whenever $w = \texttt{Target}(\overrightarrow{v_g})$ has a high indegree $\mathrm{d}^\circ_g(w)$ in the green tree takes more than $O(1)$ time. To reach constant navigation time we store one of the next siblings in clockwise direction. More precisely, we store a reference only when $\mathrm{d}^\circ_g(w) = \delta \geq 4$ and we choose $\kappa = \lfloor \frac{\delta}{3} \rfloor$ children of w in the green tree, $u_0, u_1 \ldots, u_{\kappa-1}$, so that three conditions are verified: (i) $\overrightarrow{u_0}_g$ is the right most child of w, (ii) there are no more than three green edges between $\overrightarrow{u_i}_g$ and $\overrightarrow{u_{i+1}}_g$, and (iii) there are no more than four green edges between $\overrightarrow{u_{\kappa-1}}_g$ and $\overrightarrow{w_r}$ (see Fig. 4).

To compute $\texttt{LFront}(\overrightarrow{v_g})$ we turn ccw around w, following the stored essential reference: at each step we check whether the visited edge has an extra reference going cw, and if so we use it to jump to the left and then continue ccw up to $\overrightarrow{v_g}$. The process is exactly the same for blue edges and symmetrical for red edges.

Retrieving extra references through address indirection. Let η be the number of extra references needed. As before, we need 3 tables of size n containing vertex numbers: $\mathbf{R}_\mathrm{L}$ $\mathbf{B}_\mathrm{R}$, and $\mathbf{G}_\mathrm{R}$; 12 tables of size n containing booleans $\texttt{Extra}[., c]$, $\texttt{Leaf}[., c]$, $\texttt{LOrient}[., c]$, and $\texttt{ROrient}[., c]$ for $c \in \{r, b, g\}$; two tables of size η of vertex numbers $\mathbf{F}[.]$ and $\mathbf{E}[.]$; and one table of size η of booleans $\texttt{ExtraO}(.)$. The fact that an edge, e.g. $\overrightarrow{u_i}_g$, needs an extra reference, is stored in $\texttt{Extra}[u_i, g]$. In this case $\mathbf{G}_\mathrm{R}[u_i]$, instead of storing $\texttt{Source}(\texttt{RFront}(\overrightarrow{u_i}_g))$, contains a number α to represent an indirection. Then $\mathbf{F}[\alpha]$ contains $\texttt{Source}(\texttt{RFront}(\overrightarrow{u_i}_g))$, $\texttt{ExtraO}(\alpha)$ contains a boolean to indicate the orientation and color of the extra neighbor ($\overrightarrow{u_{i+1}}_g$ or $\overrightarrow{w_r}$) we want to store and $\mathbf{E}[\alpha]$ contains its source. Navigation operators are implemented with a constant number of access to these different tables, except $\texttt{Target}(\overrightarrow{v_c})$ which still has a cost bounded by its degree.

Theorem 3. SCARST-OT *represents a triangulation of n vertices with access to neighboring edges in $O(1)$ time and storage cost $12n + \eta + 3n\log_2 n + 2\eta\log_2 n$ bits where η is sensitive to the number of high degree vertices in the triangulation. In the worst case storage remains below $13n + 5n\log_2 n$ bits.*

Proof. It remains to bound η. It is easy to see that $\forall c, \sum_{v \in T_c} \left\lfloor \frac{\mathrm{d}^\circ_T(v)_c}{3} \right\rfloor < \frac{n}{3}$ (Lemma 8 in appendix) and get that the number of extra references per color is less than $\frac{n}{3}$ and thus $\eta <$

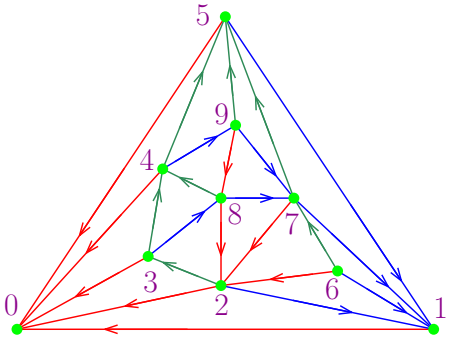

Figure 5: Schnyder wood, red BFS order

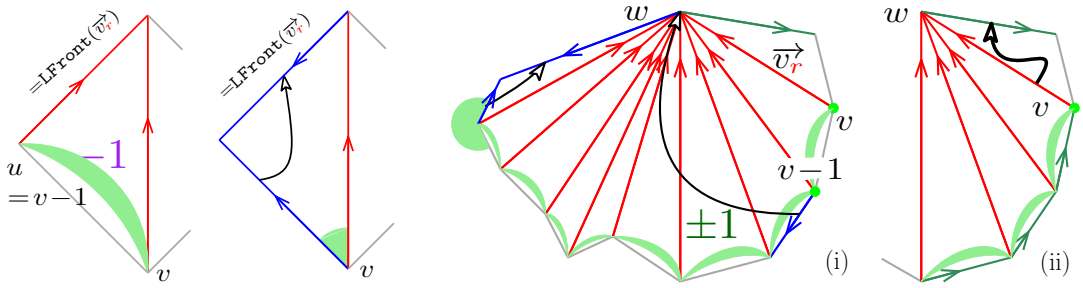

Figure 6: SCARST-RT. Red navigation

$3\frac{n}{3} = n$. This ensures that an indirection to $\mathbf{E}[.]$ fits in Tables $\mathbf{R}_{\mathrm{L}}$, $\mathbf{B}_{\mathrm{R}}$, and $\mathbf{G}_{\mathrm{R}}$ □

3.3 SCARST-RS, SCARST-RT, Reordering the vertices

Making use of vertex re-ordering one can save at least one reference per vertex, which allows us to get below the barrier of 3 references per vertex for many classes of triangle meshes. As in Castelli et al. [**?**], we re-order the vertices in red BFS order so that neighboring red edges can be obtained with basic arithmetic operations. More precisely, we use the minimal Schnyder wood without ccw triangles and we re-order the vertices such that turning ccw around a vertex enumerates its children in the red tree with consecutive indices (see Fig. 5).

We now explain how to retrieve the source of `LFront` of a red edge $\overrightarrow{v_r}$. If the searched edge is a red edge $\overrightarrow{u_r}$, then $u = v - 1$. Otherwise the searched edge is a blue edge $\overrightarrow{w_b}$: since there is no ccw triangle $\texttt{LBack}(\overrightarrow{v_r})$ is the blue edge $\overrightarrow{v_b}$ and $\texttt{LFront}(\overrightarrow{v_r})$ is obtained as $\texttt{RFront}(\overrightarrow{v_b})$. We can now remove Table $\mathbf{R}_{\mathrm{L}}$ (see Fig. 6).

In Section 3.2, we have added an additional reference for $\overrightarrow{v_r}$ when $w = \texttt{Target}(\overrightarrow{v_r})$ has a high red indegree. We can also save most of these additional references. If $\texttt{RFront}(\overrightarrow{v_r})$ is a red edge $\overrightarrow{u_r}$ then u is obtained easily as $u = v + 1$. Otherwise $\texttt{RFront}(\overrightarrow{v_r})$ is $\overrightarrow{w_g}$ and in order to retrieve it we have two cases:
(i) if $v - i$ for $i \in [0, k)$ is a green leaf then w is obtained as $\texttt{SideTarget}(\overrightarrow{(v-i)_b})$. This is true $\forall k < \mathrm{d}^{\circ}_r(w)$, but to have a constant access time we use $k = 3$.
(ii) Otherwise, we still have to use an extra reference to store w.

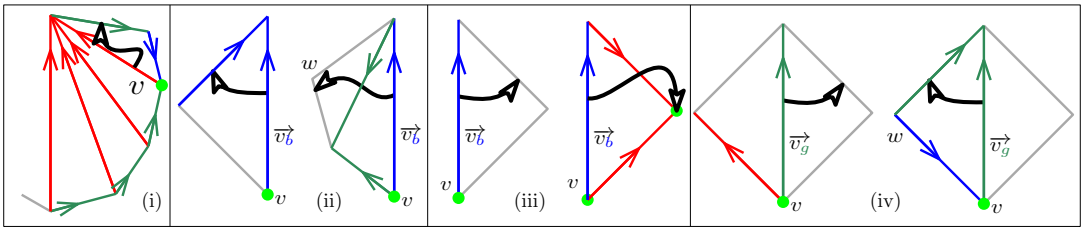

Figure 7: SCARST-WC. Definition of references: (i) red extra (ii) left blue (iii) right blue (iv) green

Storage and access time complexity With respect to the previous solution, we use exactly the same tables except $\mathbf{R}_{\mathrm{L}}$ saving one reference per vertex.

Theorem 4. SCARST-RT *represents a triangulation T of n vertices with constant access time to neighboring edges with storage cost $12n+\eta+2n\log_2 n+2\eta\log_2 n$ bits where $\eta \in [0,n)$ is sensitive to the number of high degree vertices in the triangulation. In the worst case storage remains below $12.83n + 3.67n\log_2 n$ bits.*

Proof. We have to bound η. We need an extra red reference when a vertex has at least 4 children in the red tree and three ofthem are internal nodes in the green tree. Such a situation should not arise too often, to get a conservative bound, we can permute the color so that the green tree is the one with the bigger number of leaves, in such a case the number of extra red references cannot be bigger than $\frac{n}{6}$ (Lemma 9 in appendix). The number of extra references for the blue and green tree is $\frac{n}{3}$ each, as in the previous section and $\eta \leq \frac{5}{6}n$. □

SCARST-OS can be viewed as a simplified version of SCARST-OT. In the same way, we can simplify SCARST-RT, forgetting the extra references to the price of an access time to some neighboring edges of $\overrightarrow{v_c}$ proportional to the degree of relevant vertices around $\overrightarrow{v_c}$.

Corollary 5. SCARST-RS *represents a triangulation T of n vertices with access time to neighboring edges proportional to the degree and with storage cost $9n + 2n\log_2 n$ bits.*

3.4 SCARST-WC, Improving Worst Case Storage Cost

Our last data structure achieves the best worst case bounds known so far. As in the previous section, we use a minimal Schnyder wood and vertices are numbered in BFS order of the red tree. For each blue edge we store exactly two references, one at left and one at right, and exactly one reference for each green edge. Some red edges need an extra reference as detailed below (see Fig. 7). The storage and navigation for red edges are almost identical to previous section. A reference to $\texttt{RFront}(\overrightarrow{v_r})$ is stored only if v is the rightmost child of some vertex w with $\mathrm{d}^\circ_r(w) \geq 4$ and v, $v-1$, and $v-2$ are internal nodes of T_g. Navigation is performed as in the previous section (see Fig. 8).

Two references are stored for each blue edge. On the right side, $\texttt{RFront}(\overrightarrow{v_b})$ is stored except if both $\texttt{RFront}(\overrightarrow{v_b})$ and $\texttt{RBack}(\overrightarrow{v_b})$ are red. In the latter case $\texttt{Target}(\overrightarrow{v_r})$ is stored and $\texttt{RFront}(\overrightarrow{v_b})$ is retrieved as $\overrightarrow{(v-1)}_r$. On the left side, if $\texttt{LFront}(\overrightarrow{v_b})$ is blue, then a reference to it is stored, otherwise, $\texttt{LFront}(\overrightarrow{v_b})$ is green, a reference to $\overrightarrow{w_\beta} = \texttt{RFront}(\texttt{LFront}(\overrightarrow{v_b}))$ is stored. A new table of boolean $\texttt{LROrient}(.)$ gives the color β: green or blue. In the latter case, $\texttt{LFront}(\overrightarrow{v_b})$ can

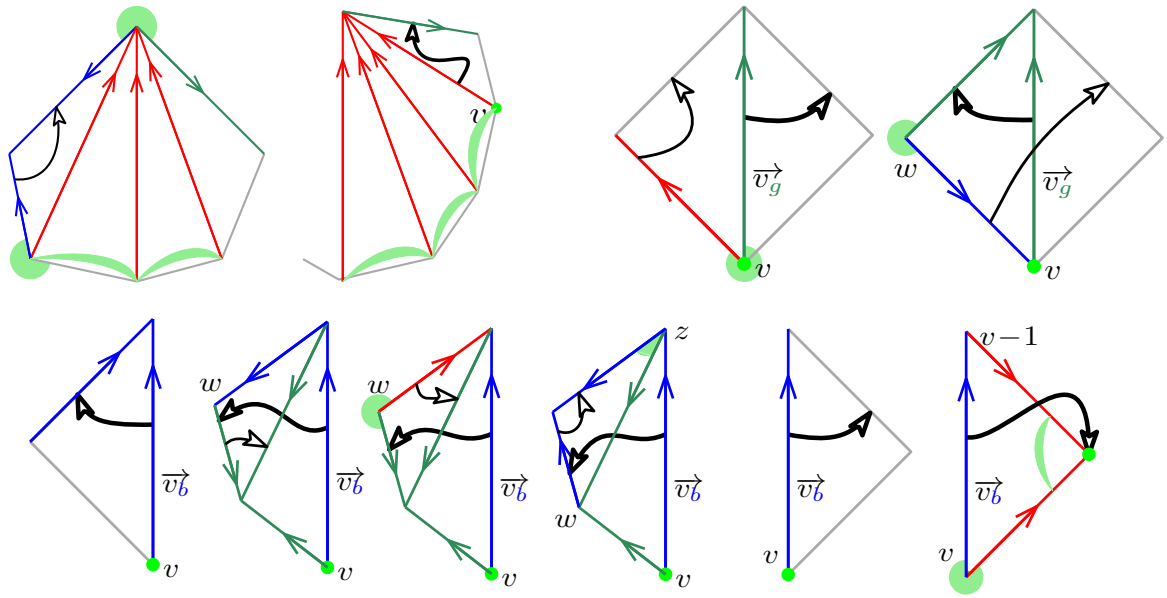

Figure 8: SCARST-WC. Red, green, and blue navigation

be retrieved, either as $\texttt{LFront}(\overrightarrow{w_g})$ if $\beta = g$ and $\texttt{LBack}(\overrightarrow{w_g})$ is blue, $\texttt{RFront}(\overrightarrow{w_r})$ if $\beta = g$ and $\texttt{LBack}(\overrightarrow{w_g})$ is red, or as $\overrightarrow{z_g}$, where $z = \texttt{Source}(\texttt{RFront}(\overrightarrow{w_b}))$ otherwise (see Fig. 8).

For a green edge, we store a single reference. If $\texttt{LBack}(\overrightarrow{v_g})$ is red, then $\texttt{RFront}(\overrightarrow{v_g})$ is stored and $\texttt{LFront}(\overrightarrow{v_g})$ is retrieved as $\texttt{RFront}(\overrightarrow{v_r})$. If $\texttt{LBack}(\overrightarrow{v_g})$ is blue, then $\texttt{LFront}(\overrightarrow{v_g}) = \overrightarrow{w_g}$ is stored. In such a case $\texttt{LBack}(\overrightarrow{v_g}) = \overrightarrow{w_b}$ and we are in the special case where the left reference of $\overrightarrow{w_b}$ is exactly $\texttt{RFront}(\overrightarrow{v_g})$ (see Fig. 8).

Storage and access time complexity. In this structure we use 3 tables of size n containing vertex numbers: $\mathbf{B}_\text{L}$, $\mathbf{B}_\text{R}$, and $\mathbf{G}$; the usual 9 tables of size n containing booleans $\texttt{Leaf}[.,c]$, $\texttt{LOrient}[.,c]$, $\texttt{ROrient}[.,c]$ $c \in \{r,b,g\}$; 2 other tables of size n containing booleans $\texttt{Extra}[v,r]$ storing the existence of an extra reference for $\overrightarrow{v_r}$ and $\texttt{LROrient}(v)$ storing the orientation of $\texttt{RFront}(\texttt{LFront}(\overrightarrow{v_b}))$ when needed; 2 tables of size η of vertex numbers $\mathbf{F}[.]$ and $\mathbf{E}[.]$ for extra red references; and finally one table of size η of booleans $\texttt{Extra0}(.)$. Navigation operators require a constant number of accesses to these different tables, except $\texttt{Target}(\overrightarrow{v_c})$ which remains degree dependant.

Theorem 6. *The above structure represents a triangulation T of n vertices with constant access time to neighboring edges with storage cost $11n + \eta + 3n\log_2 n + 2\eta\log_2 n$ bits where $\eta \in [0,n)$ is sensitive to the number of high degree vertices in the triangulation. In the worst case storage remains below $11.33n + 3.33n\log_2 n$ bits.*

Proof. As in proof of Theorem 4, $\eta \leq \frac{n}{6}$ using a suitable choice of colors. □

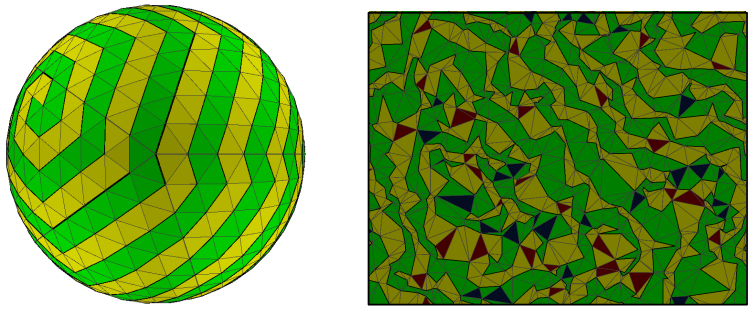

Figure 9: Laced-ring computed by our implementation of the *Ring-Expander* procedure of `LR`

4 Experimental Results

We have written `Java` implementations of our data structures[4] and compared their performances to previous works on a wide collection of datasets. In order to obtain fair runtime comparisons, we have written array-based implementations of several mesh representations: *Half-edge* (`HE`, 19 r.p.v.), the *Compact Half-edge* [**?**] (13 r.p.v.) and *Winged-edge* (19 r.p.v.) which are edge-based data structures, as well as *Corner Table* (`CT`, 13 r.p.v.), `SOT` (6 r.p.v.) and `SQUAD` which are all face-based. For `SQUAD` we follow the well detailed description in [**?**]: we use the *Matching&Pairing* procedure to compute a matching between vertices and faces and a pairing of triangles into quadrangles. For the evaluation of `LR` we have implemented its construction phase:[5] its *Ring-Expander* procedure (whose pseudo-code is provided in [**?**]), computes in linear time a quasi-hamiltonian of the graph (see Fig. 9). We also have evaluated the structure of Thm. 4 in [**?**] (whose code is publicly available).

4.1 Experimental Setting and datasets.

All our tests are performed on a laptop Latitude 7410 equipped with an Intel Core i7-10610U CPU (1.80 GHz, with 8MB of L3 cache), running under Ubuntu 22.04 and with `Java` 18 64-bit. We run our experiments on a single core allocating 4GB of RAM (we use the option `-server -Xmx4G` for the JVM). Our tests involve real-world meshes from the `aim@shape` and the `Thingi10K` repositories as well as synthetic datasets of various size and type including grid-like meshes, `delaunay` triangulations (of random points in a disk), `stacked` and `random` planar triangulations (see Appendix B.1 and B.2 for more details on datasets and implementations).

4.2 Runtime Performances.

As in previous works, we evaluate the timings by comparing the runtime performances for the procedures `degree`, `adjacent` and `normal` that perform a traversal of the edges incident to a given vertex (`adjacent` and `normal` also involves the computation of the `Target`(v) operator; `normal` requires a few numerical calculations) and `bfs` (performing a BFS graph traversal from

[4]Datasets, runnable programs and source code are available via this anonymous `Google Drive folder`.

[5]This allows a direct comparison of storage performances but not on runtimes. An indirect comparison of the runtimes of SCARST and `LR` is also possible, since the runtimes of `LR` have been evaluated and compared to `CT` in [**?**]: in the case of regular 3D meshes `CT` and `LR` achieve very close running times. To our knowledge the source code of `LR` is not publicly available.

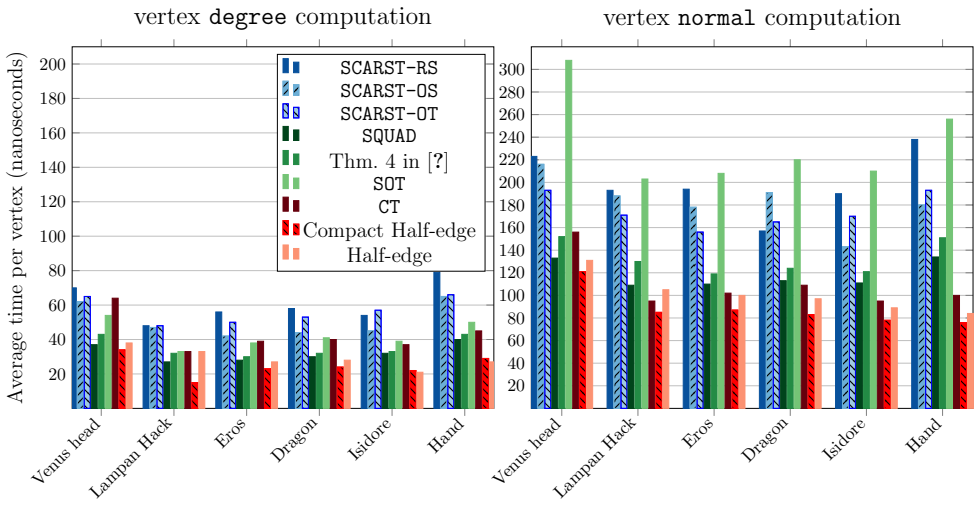

Figure 10: Average navigation timings for real-world 3D meshes.

a random seed). Fig. 10, 11 and 12 report a comparison of the average timings per query (the average is computed over 200 runs of our navigational operators). For the `degree` and `normal` operators vertices are accessed sequentially according to the original vertex ordering of the input mesh: the vertex orderings coincide for all data structures, except the one described by our Cor. 5. In the case of the `adjacent` operator we select randomly $10K$ pairs of adjacent vertices (*real edges*) and $10K$ pairs of non adjacent vertices.

Discussion. As one would expect, our data structures are in most cases slower than explicit data structures (or some previous compact representations) being much more compact. According to our results we loose a factor between 1.2 and 3.8 with respect to uncompressed storage, depending on the type of query and the class of tested graphs. It is worth noting that SCARST-OS achieves most of the time better runtimes than SCARST-OT despite the fact that the worst case $O(1)$ time is not guaranteed, in practice the computation of vertex degrees has a global linear complexity when iterating over all vertices. SCARST-OT becomes more advantageous only when the number of high degree vertices is not negligible, which occurs for irregular graphs (e.g. `random` triangulations, see Fig. 11). Observe that SCARST-RS is almost always slower than the others, which is due to two facts: because of its compactness the retrieval of service bits is more involved and, more important, the original vertex ordering is lost, which decreases the runtime performances for graphs having good vertex locality.

Observe that SCARST data structures are slower of `SOT` and `SQUAD` (between 1.2 and 2.1 times) when processing vertices in a sequential manner (as for `degree` operator) but become much more competitive, from both the storage and runtimes point of view, when accessing the data in a random manner (`adjacent` operator) or performing non local graph traversals (`bfs` operator), as illustrated in Fig. 12.

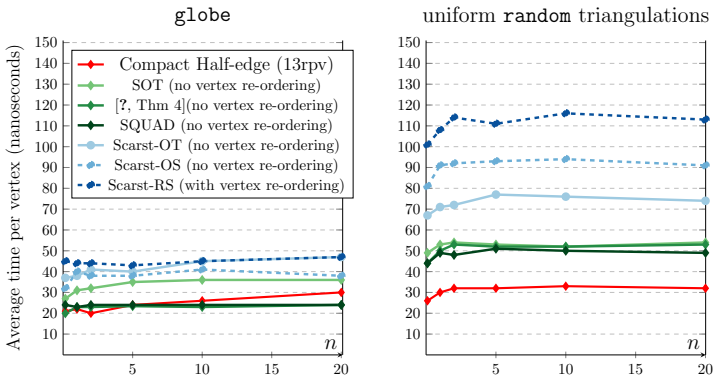

Figure 11: Comparison of timings on synthetic datasets of various size (vertex `degree`). We plot the average runtime performance as a function of the mesh size (millions of vertices).

Construction phase.

We plot in Fig. 13 the running time and memory usage of the construction phase of the SCARST-OT and SCARST-RS data structures for the `Delaunay` datasets (SCARST-OS achieves similar performances and is omitted). The timing cost (left plots) consists of two parts: the computation of the Schnyder wood and the computation of references stored by our data structures (we use the Compact Half-edge representation for storing the input graph). Our results confirm the linear behaviour of the construction procedure: once the input graph is loaded in main memory, our construction algorithm is extremely fast, being able to process about $1.56M$ vertices per second. The right plots show the peaks of the memory consumption during the construction phase (we distinguish two phases: the computation of the Schnyder wood and the data structure initialization). The memory requirements for processing a Delaunay triangulation with $20M$ vertices never exceed $2.5GB$, including the cost of geometric coordinates, for both SCARST-OT and SCARST-RS. According to our experiments these performances are similar to the ones of `SQUAD`.

4.3 Storage Comparison

The charts in Fig. 14 show the storage performances of our data structures compared to the ones of `LR` [**?**] and `SQUAD` [**?**] on both synthetic and real-world datasets. The results in Fig. 14 clearly confirm that our data structures are sensitive to the vertex degree distribution: the storage costs decrease to the lower bounds reported in Table 1 as d_6 tends to 100%. It is worth noting that our SCARST representations achieve compression rates which are always well below the upper bounds established in Section 3.

Structures preserving vertex ordering. SCARST-OT preserve vertex ordering and allow $O(1)$ time edge/face navigation as `SQUAD` does. The storage of SCARST-OT is almost always below 4 r.p.v. which makes SCARST-OT between 10% and 25% more compact than `SQUAD` in practice. The only exception are grid-like meshes such as `Lampan Hack`, having roughly half of the vertices

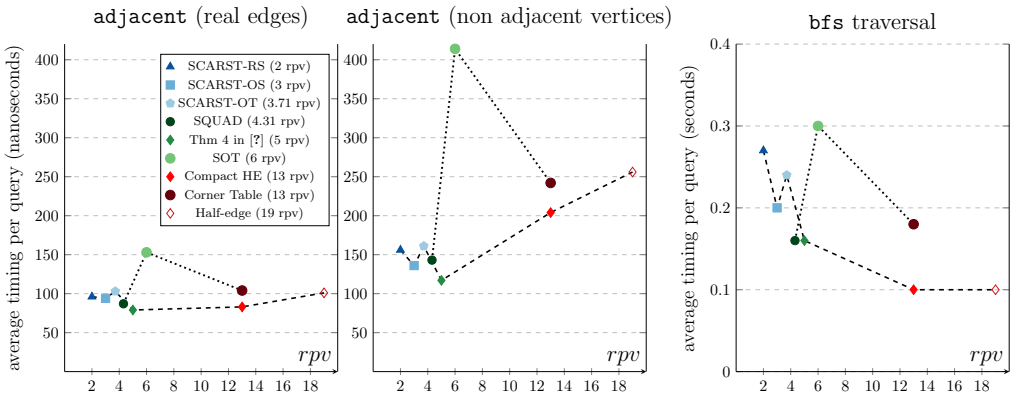

Figure 12: Trade-off between storage and runtime (vertices are accessed at random): we run the `adjacent` operator on the `delaunay1M` graph. Left (resp. right) plot reports the average timings for the case of real edges (resp. pair of non adjacent vertices) in the graph. Results are expressed as a function of the storage cost. The dashed line (resp. dotted line) connects edge-based (resp. triangle-based) representations.

of high degree. SCARST-OS only requires 3 r.p.v., which make it at least 25% more compact than `SQUAD` (and for some datasets even more).

Structures not preserving vertex ordering. Concerning structures reordering the vertices, SCARST-RT and SCARST-WC have similar compression rates compared to `LR` while providing worst case provable guarantees, and performing much better in some cases. More precisely, our experiments confirm the intuition that for regular 3D meshes (with $d_6 \geq 50\%$), `LR` achieves in practice very good compression rates ranging between 2 and 2.5 r.p.v..

But we have observed that in the case of irregular meshes (few degree 6 vertices and many separating triangles) the storage cost of `LR` increases significantly. According to our experiments it is more than 6 r.p.v. for random triangulations and even worst for some graphs (e.g. for stack triangulation we got more than 12 r.p.v.), which makes `LR` far less compact than SCARST (a few more details on the storage of `LR` can be found in Appendix B.3).

5 Concluding Remarks

Topology extensions. The results stated here and in [**?**, **?**, **?**, **?**] hold for planar[6] triangulations. We mention that it could be possible to adapt our data structures to deal with genus 1 triangulations making use of toroidal Schnyder woods [**?**]. In the higher genus case one can planarize the surface, cutting the graph along non contractible cycles of length $O(\sqrt{n})$: as mentioned in [**?**] this approach is suitable for dealing with bounded genus, since the number of duplicated vertices remains asymptotically negligible.

Sensitivity to regularity. Observe that the number of additional skipping references (e.g. ccw pointers in Fig. 4) can be modified in order to obtain different trade-offs between running time and storage cost (the bounds depending on η in our theorems will change accordingly). Our algorithms are sensitive to the regularity of the triangulation. These sensitivity appears in the complexities through the parameter η for which we give bounds in the worst case scenario. These

[6]It is worth noting that a large proportion of real-world 3D meshes are planar: among the 5806 manifold meshes of the `Thingi10K` repository, 2474 do have genus 0.

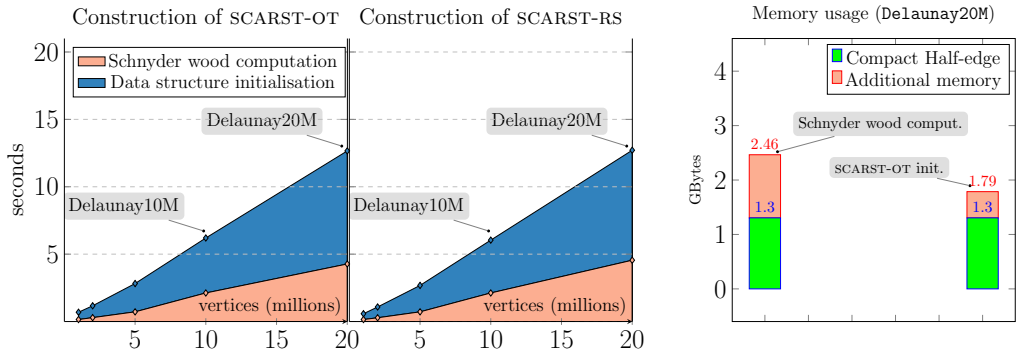

Figure 13: Runtime cost (left) and memory consumption (right) of the construction phase.

bounds can be improved if we have some knowledge of the input graph: this occurs for instance for some classes of planar triangulations (such as Delaunay triangulations of random points or uniform planar triangulations) for which the vertex distribution can be estimated asymptotically.

Streamable decompression from compressed format. Castelli et al. [**?**, Section 3] describes a procedure to construct the compact representation of Proposition 1 directly from a compressed format of size $4n$ bits without the usage of extra storage. This can be adapted to our SCARST-OS and SCARST-OT data structures.

A Combinatorial Lemmas

Let $\mathrm{d}^\circ_T(v)$ denote the indegree of a vertex v in a rooted tree T. Let $N_{=d}(T)$ and $N_{\geq d}(T)$ denote the number of vertices of T whose indegree is, respectively, exactly d and at least d. For short the number of leaves of a rooted tree is denoted by $L(T) = N_{=0}(T)$.

Lemma 7. *For all positive integer d, we have*

$$L(T) > (d-1)N_{\geq d}(T).$$

Proof. The total indegree must be the same as the total outdegree. In a rooted tree the outdegree is 1 for all nodes except for the root (outdegree 0). Since the number of nodes of the tree is $\sum_{i=0}^{\infty} N_{=i}(T)$ we have:

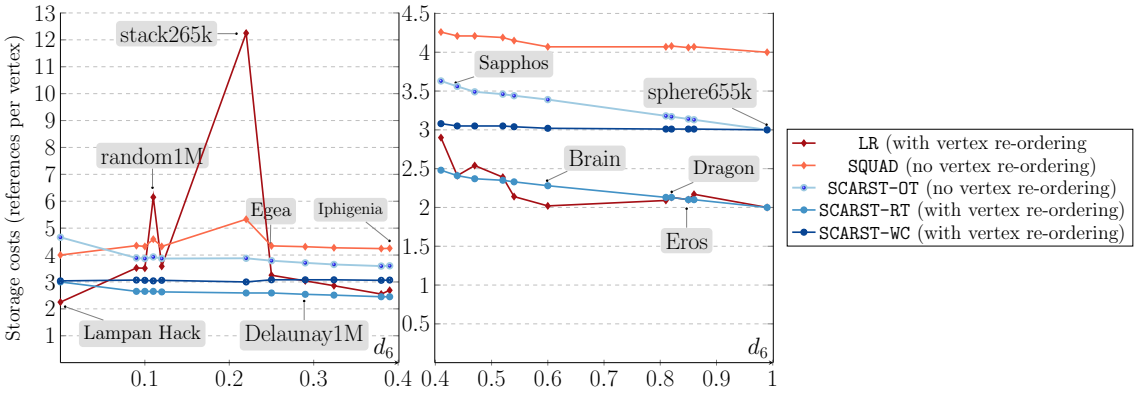

Figure 14: Comparison of storage results (expressed in terms of r.p.v. as function of d_6). (left) irregular and synthetic graphs ($d_6 < 40\%$); (right) 3D real-world regular meshes ($d_6 \geq 40\%$).

$$\begin{aligned}
\Big(\sum_{i=0}^{\infty} N_{=i}(T)\Big) - 1 &= \sum_{i=0}^{\infty} i \cdot N_{=i}(T) \\
0 &= 1 + \sum_{i=0}^{\infty} (i-1) \cdot N_{=i}(T) \\
N_{=0}(T) &= 1 + \sum_{i=2}^{d-1} (i-1) \cdot N_{=i}(T) + \sum_{i=d}^{\infty} (i-1) \cdot N_{=i}(T) \\
L(T) &> 0 + 0 + (d-1) \cdot N_{\geq d}(T).
\end{aligned}$$

□

Lemma 8. *For a rooted tree T and $k \in \mathbb{N}^+$:* $\sum_{v \in T} \left\lfloor \frac{\mathrm{d}^\circ_T(v)}{k} \right\rfloor < \frac{n}{k}$.

Proof. It comes easily $\sum_{v \in T} \left\lfloor \frac{\mathrm{d}^\circ_T(v)}{k} \right\rfloor \leq \sum_{v \in T} \frac{\mathrm{d}^\circ_T(v)}{k} = \frac{1}{k} \sum_{v \in T} \mathrm{d}^\circ_T(v) = \frac{(n-1)}{k} < \frac{n}{k}$. Notice that this bound is tight when, e.g., all leaves are children of the root. □

Given a Schnyder wood (T_r, T_b, T_g), we denote $K_r^{(d)}$ the number of vertices w such that $\mathrm{d}^\circ_{T_r}(w) \geq d+1$ and such that at least d children of w are not a leaf of T_g.

Lemma 9. *Given a Schnyder wood on n vertices, such that the green tree has more leaves than the red tree, we have $K_r^{(d)} \leq \frac{1}{2d}n$.*

Proof. A vertex to be counted can be associated with d internal nodes of the green tree, as a consequence T_g has at least $dK_r^{(d)}$ internal nodes. Thus

$$\begin{aligned} dK_r^{(d)} &\leq N_{\geq 1}(T_g) \\ n - dK_r^{(d)} &\geq n - N_{\geq 1}(T_g) \\ &\geq L(T_g) \qquad \text{total size of } T_g \text{ is } n \\ &\geq L(T_r) \qquad T_g \text{ minimizes } \sharp\text{leaves} \\ &\geq dN_{\geq d+1}(T_r) \qquad \text{by Lemma 7} \\ &\geq dK_r^{(d)}. \end{aligned}$$

□

B Implementation and datasets

B.1 Datasets

The datasets tested in our work include several classes of meshes:

- 3D real-world meshes: they are taken from the `aim@shape` and the `Thingi10K` repositories;
- synthetic grid-like meshes: `globe` and `sphere`;
- `delaunay` triangulations of random points in a disk (generated by the `CGAL` library): we first re-arranged the points using Hilbert spatial sorting to get good vertex locality for better runtime performances;
- irregular graphs: `stacked` triangulations and `random` planar triangulations (obtained with an uniform random sampler [**?**]).

Fig. 15 reports some mesh statistics. As suggested in previous works [**?**] the proportion d_6 of degree 6 vertices is a good parameter for measuring the regularity of a mesh, we also report the maximum vertex degree d_{max}. Our experiments suggest that number of separating triangles (denoted t_s) and its normalized value (t_s/n), are also interesting parameters for the storage evaluation of some encoding schemes. As a measure of the vertex locality we also report the *average gap profile*, defined by $\hat{\xi}(G) = (\sum_{e=(u,v)\in E} |u-v|)/|E|$, where E is the set of edges of a graph G and $|u-v|$ is the absolute value of the difference of the indices of u and v.

B.2 SCARST: implementation details

All the variants of SCARST encode incidence relations between mesh elements and other informations (e.g. edge orientations) using several tables storing either integer numbers or boolean values. For the sake of compactness, as done in previous compact data structures, we store booleans as *service bits* within 32-bits words, using the remaining part for encoding integer references. In our implementation we prefer to represent edges with integer indices in $[0, 3n)$, instead of handling an edge $\overrightarrow{v_c}$ as a pair (v, c) (where vertex indices are on $\log_2 n$ bits). This choice makes our implementation slightly faster since performing some basic operations is straightforward: the

Graphs	n	d_6	d_{max}	$\hat{\xi}$	t_s	t_s/n
Lampan Hack	309506	0.1%	192	525	0	0
Pumpkin head	197257	39%	15	6951	1356	0.0068
Hand	1.68M	41%	12	25896	12	7×10^{-6}
Sapphos	140864	43%	11	351	1469	0.0104
Venus head	239874	44%	28	12486	500	0.0020
Arc triomphe	174066	47%	23	548	310	0.0017
Brain	396451	60%	10	675	212	0.0005
Isidore	1.1M	81%	32	14203	1292	0.0011
Dragon	655980	82%	25	18778	888	0.0013
random20M	20M	12%	65	5789	7410266	0.370
stack797K	797164	22%	12288	3.4×10^5	797160	0.999
Delaunay20M	20M	29%	326	3965	228505	0.011
sphere10M	10.4M	99.9%	6	3.7×10^6	0	0
globe20M	20M	99.9%	4473	4472	0	0

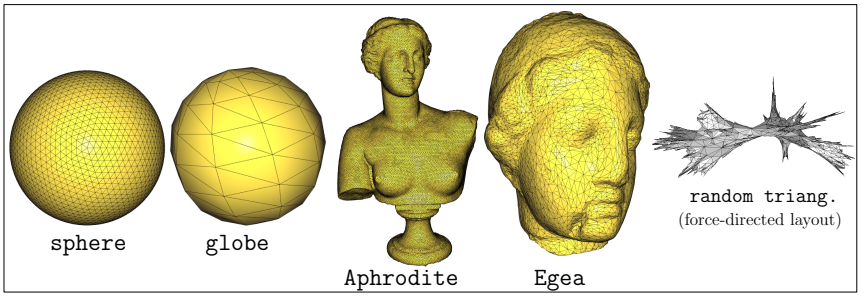

Figure 15: Mesh statistics for a selection of the tested meshes.

index of edge $\overrightarrow{v_c}$ is simply $3 * v + c$ (where $c \in \{0, 1, 2\}$), while the source and color of an edge of index e are given by $e/3$ and $e\%3$ respectively. As we make use of a bounded and small number of service bits per word (between 3 and 6 depending on the data structure), we can represent very large meshes. For instance, for the representation of SCARST-OT we need 4 service bits per edge: this allows us to process triangulations having up to $2^{28} \simeq 268$ million edges (equivalently up to 89 million vertices). In the case of huge datasets, when the size constraint is an issue, we can reduce the number of service bits and store booleans in separate tables. According to our preliminary tests, the runtime performance is slightly worst in this case.

B.3 LR: an overview

The compactness of `LR` [**?**] is obtained by re-ordering vertices and triangles according to the traversal of a so-called *ring* (an edge cycle) computed by the *Ring-Expander* procedure (see Figure 9). The map from vertices to triangles and the neighboring relations between adjacent triangles can be implicitly encoded by this vertex/face re-ordering.

In the case of regular meshes (large majority of low degree vertices and few separating triangles) the *ring* is quasi-hamiltonian and the number of isolated vertices (not covered by the cycle) is small. The set of faces is split into two subsets that resemble to triangle strips: the number of bifurcation triangles (not having any incident edge on the ring) is really small, as illustrated in Fig. 9. The storage requirement of `LR` strongly depends on the number of edges captured in the *ring* and on the number of bifurcation triangles, and can be computed according to a formula given in Section 8 of [**?**]: in principle, when bifurcation triangles and isolated vertices are very few, only one reference per triangle suffices to retrieve in constant time all mesh incidence relations between faces and vertices. This is why, at least for regular 3D meshes having $d_6 \geq 50\%$, `LR` achieves in practice very good compression rates ranging between 2 and 2.5 r.p.v.. As done in [**?**] in all our experiments we take the best results over 50 runs with random seeds, and perform the post-processing *wart skipping* optimization.

In the case of irregular graphs the number of isolated vertices and bifurcation triangles is not negligible[7] and the storage cost increases significantly. According to our experiments the storage performance of `LR` both depends on d_6 and on the number of separating triangles. This also explains the poor compression rates of `LR` in the case of uniform random triangulations (more than 6 r.p.v.) and stack triangulations (for which we got more than 12 r.p.v.).

It is worth noting that the problem above could also impact the storage performances of `BELR` and `Zipper` in the case of irregular graphs, since they combine the re-ordering approach based on the hamiltonian cycle with an efficient encoding of variable length references.

[7] The existence of separating triangles is a strong obstruction to the existence of hamiltonian cycles in planar triangulations [**?**].