Explicit array-based compact data structures for triangulations: practical solutions with theoretical guarantees

Luca Castelli Aleardi, Olivier Devillers

Explicit array-based compact data structures for triangulations: practical solutions with theoretical guarantees

Luca Castelli Aleardi, Olivier Devillers

RESEARCH REPORT
N° 7736
2011
Project-Team Gamble

ISSN 0249-6399 ISRN INRIA/RR--7736--FR+ENG

Explicit array-based compact data structures for triangulations: practical solutions with theoretical guarantees

Luca Castelli Aleardi*, Olivier Devillers†‡§

Project-Team Gamble

Research Report n° 7736 — version 2 — initial version 2011 — revised version November 2017 — 36 pages

Abstract: We consider the problem of designing space efficient solutions for representing triangle meshes. Our main result is a new explicit data structure for compactly representing planar triangulations: if one is allowed to permute input vertices, then a triangulation with n vertices requires at most $4n$ references ($5n$ references if vertex permutations are not allowed). Our solution combines existing techniques from mesh encoding with a novel use of maximal Schnyder woods. Our approach extends to higher genus triangulations and could be applied to other families of meshes (such as quadrangular or polygonal meshes). As far as we know, our solution provides the most parsimonious data structures for triangulations, allowing constant time navigation. Our data structures require linear construction time, and are fast decodable from a standard compressed format without using additional memory allocation. All bounds, concerning storage requirements and navigation performances, hold in the worst case. We have implemented and tested our results, and experiments confirm the practical interest of compact data structures.

Key-words: triangulations, compact representations, mesh data structures, codage de graphes, Schnyder woods

This work is partially supported by ERC under the agreement ERC StG 208471 - ExploreMap.

* LIX, École Polytechnique, France
† Inria, Centre de recherche Nancy - Grand Est, France.
‡ CNRS, Loria, France.
§ Université de Lorraine, France

RESEARCH CENTRE NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Structures de données explicites compactes basées sur des tableaux pour les triangulations

Résumé : Nous nous intéressons là la conception de représentations efficaces pour les maillages triangulaires. Le résultat principal de ce travail est une nouvelle structure de données pour la représentation compacte des triangulations planaires: si on autorise la permutation des sommets du maillages, alors une triangulations à n sommets peut se représenter avec au plus $4n$ références ($5n$ références sont nécessaires, si on ne fait pas de permutation). Notre solution combine des techniques existantes de codage de graphes avec une nouvelle utilisation des Schnyder woods minimaux. Cette approche s'étend aussi au cas de triangulations de genre supérieur et pourrait s'appliquer à d'autres familles de maillages (tels que les maillages quadrangulaires ou polygonaux). A notre connaissance, ce résultat fournit la structure de données la plus compacte pour les triangulations, permettant la navigation dans le maillage en temps constant dans le pire des cas. Le temps de construction de nos représentations est linéaire, et toutes les bornes sont valables dans le pire des cas. Nous avons implémenté et testé nos résultats, et nos expériences confirment l'intérêt pratique des structures de données compactes.

Mots-clés : triangulations, représentations compactes, structures de données pour les maillages, Schnyder woods

1 Introduction

The large diffusion of geometric meshes (in application domains such as geometry modelling, computer graphics), and especially their increasing complexity has motivated a huge number of recent works in the domain of graph encoding and mesh compression. In particular, the *connectivity* information of a mesh (describing the incidence relations) represents the most expensive part (compared to the geometry information): for this reason most works try to reduce the first kind of information, involving the combinatorial structure of the underlying graph. Many works addressed the problem from the *compression* [25, 42, 39, 38] point of view: compression schemes aim to reduce the number of bits as much as possible, possibly close to theoretical minimum bound according to information theory.

For applications requiring the manipulation of input data, a number of explicit (pointer-based) data structures [8, 3, 4, 24] have been developed for many classes of surface and volume meshes. Most geometric algorithms require data structures which are easy to implement, allowing fast navigation between mesh elements (edges, faces and vertices), as well as efficient update primitives. Not surprisingly common mesh representations are redundant and store a not negligible amount of information in order to achieve the prescribed requirements. Standard implementations consume between $13n$ and $19n$ references for storing in main memory a triangulation of n vertices, while compact representations use often less than $6n$ references (see Table 1). Observe that if one requires to encode a planar triangulation without navigation support, then it is possible to obtain in linear time a compressed format [38] using asymptotically at most 3.2451 bits per vertex. In this work we address the problem above (reducing memory requirements) from the point of view of *compact data structures*: the goal is to reduce the redundancy of common explicit representations, while still supporting efficient navigation, and allow good compression rate.

1.1 Existing Mesh Data Structures

Classical data structures in most programming environments do admit *explicit pointer-based* implementations. Each pointer stores at most one reference: pointers allow navigating in the data structure through address indirection, but storing/manipulating service bits within references is not always allowed (this occurs, for instance, for programming languages not allowing pointer arithmetic such as Java, C# or Javascript). Many popular geometric data structures (such as *Quad-edge*, *Winged-edge*, *Half-edge*) fit in this framework. In edge-based representations basic elements are edges (or half-edges): navigation is performed by storing, for each edge, a number of references to incident mesh elements. For example, in the *Half-edge DS* each half-edge stores a reference to the next and opposite half-edge, together with a reference to an incident vertex: which gives 3 references for each of the $6n$ half-edges, for a triangulation having n vertices. If one has to store data associated to triangles (e.g. face normals, face colors, ...) then some additional references are necessary in order to represent the map between edges and triangles.

Compact practical solutions with theoretical guarantees. Several works [10, 41, 31, 2, 12, 29, 26, 27] try to reduce the number of references stored by common mesh representations: this leads to more compact solutions, whose performances (in terms of running time) are still really of practical interest. In this case array-based implementations are sometimes preferred to pointer-based representations, depending on the flexibility of the programming environment. Many data structures (*triangle-based*, *array-based compact half-edge*, *SOT/SQUAD data structures*) make use of the following further assumption: each memory word can store a $\lg n$ bits integer reference,[1] and C bits are reserved as *service bits* (C is a small constant, commonly between

[1] For a mesh with n elements, $\lg n := \lceil \log_2 n \rceil$ bits are necessary to distinguish all the elements. The length w

1 and 4). Moreover, basic arithmetic operations are allowed on references: such as addition, multiplication, floored division, and bit shifts/masks. An interesting general approach is based on the reordering of mesh elements: for example, storing consecutively the half-edges of a face allows us to save 3 references per face (as in *Directed Edge* [10] or in [2], which requires $13n$ references instead of the $19n$ stored by *Half-edge*). Or still, storing edges/faces according to the vertex ordering allows us to implicitly represent the map from edges/faces to vertices. This is one of the ingredients used by the SOT data structure [29], which represents triangulations with $6n$ references. The combination of face pairing and vertex re-ordering also leads to dynamic data structures [13] supporting standard local updates in amortized constant time, and whose space requirements range between $6n$ and $4.8n$ references.

More concise practical solutions Adopting some interesting heuristics one may obtain even more compact solutions [26, 27, 28], requiring better space requirements in practice, but with no theoretical guarantees in the worst case. For instance, the face pairing approach of the $SQUAD$ data structure improves the SOT bounds to about $(4 + c)$ references per vertex: as shown by experiments c is usually a small value (between 0.09 and 0.3 for tested meshes). Another heuristic combines the face and vertex re-ordering with computation of a nearly Hamiltonian ring spanning almost all vertices: as reported in [27], the LR data structure is able to represent a triangulation using between 2.04 and 3.16 references per vertex (results hold for the tested meshes).

Even smaller memory requirements can be achieved making use of difference encoding of references: the bit-efficient version LR [27] consumes between 37 and 90 bits per vertex, while the *Zipper* [28] is able to achieve in average only 12 bits per vertex (all results hold for the tested meshes). Observe that these better compression rates are obtained at the cost of more expensive navigation: as discussed in [27] the standard version of LR has runtime performances comparable to the ones of the Corner Table (when 3D meshes fit in main memory), while the bit-efficient version of LR is five times slower.

One common issue of compact representation is that the whole mesh must be kept in main memory during the pre-processing construction phase. This issue is addressed by *Grouper* [35]: the combination of compact mesh data structures with techniques from streaming meshes for out-of-core computations allows constructing the compact representation on the fly from a compressed format and in a streamable way.

Finally, we observe that the parsimonious use of references may affect the navigation time, for the retrieval of some mesh elements: for example, the access to vertices may require more than $O(1)$ time in the worst case [29, 26, 13, 31]. Table 1 reports some trade-offs between space requirements and navigation performances.

Theoretically optimal solutions. For completeness, we mention that *succinct representations* [18, 7, 37, 16, 15, 43] are successful in representing meshes with the minimum number of bits, while supporting local navigation in worst case $O(1)$ time. They run under the *word-Ram model*, where basic arithmetic and bitwise operations on words of size $O(\lg n)$ are performed in $O(1)$ time. One main idea (underlying almost all solutions) is to reduce the size, and not only the number, of references: one may use graph separators or hierarchical graph decomposition techniques in order to store in a memory word an arbitrary (small) number of tiny references. Typically, one may stores up to $O(\frac{\lg n}{\lg \lg n})$ sub-words of length $O(\lg \lg n)$ each. Unfortunately, the number of auxiliary bits needed by the encoding becomes asymptotically negligible only for very huge graphs, which makes succinct representations of mainly theoretical interest.

of each memory word is assumed to be $\Omega(\lg n)$

Data structure	size (references)	navigation	vertex access	vertex adjacency	dynamic updates
Edge-based data structures [24, 3, 4]	$18n + n$	$O(1)$	$O(1)$	$O(d)$	*yes*
triangle based [8]/Corner Table	$12n + n$	$O(1)$	$O(1)$	$O(d)$	*yes*
Directed edge [10]/Compact half-edge [2]	$12n + n$	$O(1)$	$O(1)$	$O(d)$	*yes*
2D catalogs [12]	$7.67n$	$O(1)$	$O(1)$	$O(d)$	*yes*
Star vertices [31]	$7n$	$O(d)$	$O(1)$	$O(d)$	*no*
TRIPOD [41] or Thm 2	$6n$	$O(1)$	$O(d)$	$O(d)$	*no*
SOT [29]	$6n$	$O(1)$	$O(d)$	$O(d)$	*no*
ESQ [13]	$4.8n$	$O(1)$	$O(d)$	$O(d)$	*yes*
(no vertex reordering) Thm 4	$5n$	$O(1)$	$O(d)$	$O(d)$	*no*
(with vertex reordering) Thm 5	$4n$	$O(1)$	$O(d)$	$O(d)$	*no*
(with vertex reordering) Thm 6	$5n$	$O(1)$	$O(1)$	$O(1)$	*no*

Table 1: Comparison between existing data structures for triangle meshes. All storage and runtime bounds hold in the worst case. The degree of the accessed vertex is denoted d.

For a more detailed discussion on triangle mesh data structures we refer to [14], while a comprehensive explanation of recent advances in 3D mesh compression can be found in [1, 36].

1.2 Preliminaries

Combinatorial aspects of triangulations. In this work we exploit a deep and strong combinatorial characterization of planar triangulations. A planar triangulation is a simple planar map where every face has degree 3.[2] Triangulations are *rooted* if there is one distinguished *root face*, denoted by (V_0, V_1, V_2), with a distinguished incident *root edge* $\{V_0, V_1\}$. *Inner edges* (and *inner vertices*) are those not belonging to the root face (V_0, V_1, V_2).

As pointed out by Schnyder [40], the inner edges of a planar triangulation can be partitioned into three sets $\mathcal{T}_{\text{red}}$, $\mathcal{T}_{\text{blue}}$, $\mathcal{T}_{\text{black}}$, which are plane trees spanning all inner vertices, and rooted at V_0, V_1 and V_2 respectively. This spanning condition can be derived from a local condition: the inner edges can be oriented in such a way that every inner vertex is incident to exactly 3 outgoing edges, and the orientation/coloration of edges must satisfy a special local rule (see Figure 1).

Definition 1 ([40]). *Let $\mathcal{G}$ be a planar triangulation with root face (V_0, V_1, V_2). A* ***Schnyder wood*** *of $\mathcal{G}$ is an orientation and labeling, with label in* {red, blue, black} *of the inner edges such that the edges incident to the vertices V_0, V_1, V_2 are all incoming and are respectively of color* red, blue, *and* black. *Moreover, each inner vertex v has exactly three outgoing incident edges, one for each color, and the edges incident to v in* counter clockwise *(ccw) order are:*

one outgoing edge colored red,
zero or more incoming edges colored black,
one outgoing edge colored blue,
zero or more incoming edges colored red,
one outgoing edge colored black, *and*
zero or more incoming edges colored blue

(this is referred to as ***local Schnyder condition****).*

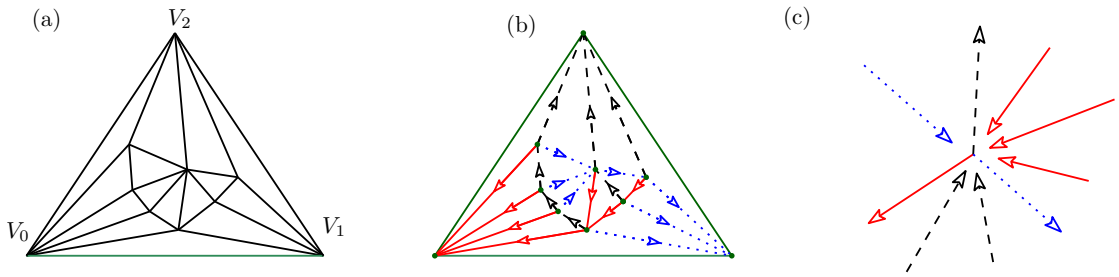


Figure 1: A planar triangulation with 9 vertices (a), endowed with a Schnyder wood (b). Picture (c) local Schnyder condition around inner vertices.

In a triangulation, the edges are originally not oriented, and they get an orientation during the Schnyder wood construction. If an edge e between vertices u and v is oriented from u to v, it will be denoted (u, v) and (v, u) if the edge is oriented in the other direction. If the orientation is unknown the edge will be denoted by $\{u, v\}$. In a Schnyder wood, the inner edges get an orientation, thus we have either $\{u, v\} = (u, v)$ or $\{u, v\} = (v, u)$.

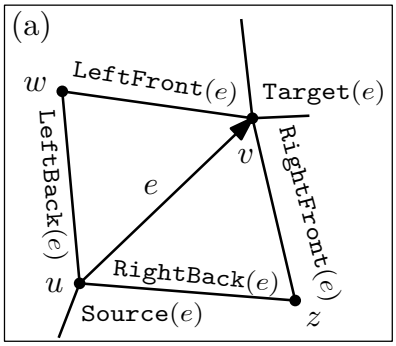
A Schnyder wood of a planar triangulation [40] can be computed in linear time: for the sake of completeness we provide in Appendix a short illustration of the algorithm based on vertex conquests detailed in [9, 33].

Navigational operators.

Here are the operators supported by our representations. Let $e = (u, v)$ be an edge oriented toward v, which is incident to (u, v, w) (its left triangle) and to (v, u, z) (its right triangle), as depicted in Figure 2.

- `LeftBack`(e), returns the edge $\{u, w\}$ (i.e. (u, w) or (w, u)).
- `LeftFront`(e), returns the edge $\{v, w\}$.
- `RightBack`(e), returns the edge $\{u, z\}$.
- `RightFront`(e), returns the edge $\{v, z\}$.

[2] When embedded in the plane, all faces are triangles and the convex hull is also a triangle




```
Degree(u)                    // Enumerating edges incident to u .
{
        e = Edge(u);
        f = e; d = 0;
        do{
                if u =Source(f) f =LeftBack(f) ;
                else            f =RightFront(f) ;
                d = d + 1;
        } while f ≠ e;
        return d;
}
```

Figure 2: (a) Navigational operators supported by our representations.
(b) Enumerating edges incident to a vertex using navigational operators.

- `Source`(e), returns u.
- `Target`(e), returns v.
- `Edge`(u), returns an edge incident to vertex u;
- `Point`(u), returns the geometric coordinates of vertex u.
- `Edge`(f), returns an edge incident to face f;
- $\texttt{Face}_{\texttt{left}}(e)$, returns the face at the left of oriented edge e;
- $\texttt{Face}_{\texttt{right}}(e)$, returns the face at the right of oriented edge e;

The operators above are supported by most mesh representations [10, 3], and allow full navigation in the mesh as required in geometric processing algorithms. Their combination allows us to walk around the edges incident to a given face, or to iterate on the edges incident to a given vertex leading, for instance, to support the operator

- `Adjacent`(u, v), test whether two vertices u and v are adjacent.

This last operator cannot be supported in constant worst case time by common mesh representations: testing the adjacency between vertices is sometimes needed in graph algorithms, and the timing cost is proportional to the degree of the involved vertices.

Overview of our Solution In order to design new compact array-based data structures, we make use of many ingredients: some of them concerning the combinatorics of graphs, and some of them pertaining the design of compact (explicit) data structures. The main steps of our approach are the following:

- as done in [41], we exploit the existence of 3-orientations (edge orientations where every inner vertex has outgoing degree 3) for planar triangulations [40]. This allows storing only two references per edge (corresponding to its `LeftFront` and `RightFront` edges).
- as done in [10, 29, 26], we perform a reordering of cells (edges), to implicitly represent the map from vertices to edges, and the map from edges to vertices;

Combining these two ideas one can easily obtain an array-based representation using $6n$ references: we store the three edges outgoing from a vertex consecutively, so that the retrieval of the missing information involving the `LeftBack` and `RightBack` edges can be efficiently performed exploiting the local Schnyder condition. Our first data structure provides $O(1)$ time navigation between edges and $O(d)$ time for the access to a vertex of degree d: for the sake of completeness, this simple solution will be detailed in Theorem 2.

Additionally, we design a coding scheme that can produce a compressed file of size $4n$ bits (which is just above the theoretical lower bound of 3.24 bits per vertex [38]); our array-based representation can be restored from the compressed file without using extra memory. Our main contribution is to show how to get further improvements and generalizations, using the following ideas:

- we exploit the existence of *maximal* Schnyder woods, without cycles of directed edges oriented in cw direction. And also the fact that, given the partition $(\mathcal{T}_{\text{red}}, \mathcal{T}_{\text{blue}}, \mathcal{T}_{\text{black}})$, the two trees $\mathcal{T}_{\text{red}}$ and $\mathcal{T}_{\text{black}}$, are sufficient to retrieve the triangulation. With these ideas we succeed to store only $5n$ references (Theorem 4).

- we show how to reconstruct our compact representations (Theorems 4 and 2) from a standard compressed format which uses less than $4n$ bits, without extra memory: both the encoding and decoding phases require linear time (Section 3).
- we further push the limits of the previous reordering approach: by arranging the input points according to a given permutation and using a special order on plane trees we are able to use only $4n$ references (Theorem 5);
- Using one extra reference per vertex, we are able to efficiently recover the target of every edge, providing worst case $O(1)$ time access to vertices instead of $O(d)$ (Theorem 6).
- Our compact data structure described in Theorem 6 supports even the `Adjacent` operator in $O(1)$ time on planar triangulations.
- For applications requiring to store data associated to triangles, Theorem 7 shows how to represent the map between edges and triangles using one additional reference per vertex (the retrieval of data associated to triangles is performed in $O(1)$ time).

To our knowledge, these are the best (worst case and guaranteed) upper bounds obtained so far, which improve previous existing results. Finally we mention that our representations can be adapted to deal with more general triangle meshes, by using the reformulations of Schnyder woods proposed for toroidal [22, 20] and genus g [17] triangulations.

2 Compactly Representing Triangulations

We first design a simple data structure requiring $6n$ references, which allows performing all navigational operators in worst case $O(1)$ time, and `Target` operator in $O(d)$ time (when retrieving a degree d vertex). This is a preliminary step in describing a more compact solution. Observe that our first scheme achieves the same space bounds as the *Tripod data structure* by Snoeyink and Speckmann [41]. Although both solutions are based on the properties of Schnyder woods, the use of references (between edges) is different: this is one of the features which allow us to make our scheme even more compact in the sequel.

2.1 The First Data Structure: Simple and Still Redundant

Main ideas: the coloring of edges allow an easy matching between vertices and edges of each color to organize edges in three arrays. For each edge we store the leftfront and rightfront edges; the leftback and the rightback edges are retrieved using Schnyder wood rules.

Scheme Description. We firstly compute a Schnyder wood $(\mathcal{T}_{\text{red}}, \mathcal{T}_{\text{blue}}\mathcal{T}_{\text{black}})$ of the input triangulation $\mathcal{G}$ (in linear time, as shown in [40]). We define the tree $\overline{\mathcal{T}}_{\text{red}}$ by adding edges (V_1, V_0) and (V_2, V_0) to $\mathcal{T}_{\text{red}}$; we add the edge (V_2, V_1) to the tree $\mathcal{T}_{\text{blue}}$, as depicted in Figure 3. Thus each edge gets a color and an orientation. Since local Schnyder condition ensures exactly one outgoing edge of each color (except V_0, V_1, V_2) an edge can also be identified by its origin u and its color c and denoted $u_c^{\curvearrowright}$. For each color c and each vertex u, we store two vertex numbers $U_c^{\text{left}}[u]$ and $U_c^{\text{right}}[u]$ and two booleans $T_c^{\text{left}}[u]$ and $T_c^{\text{right}}[u]$ describing the source and color of the two neighboring edges of edge $u_c^{\curvearrowright}$ as detailed below and one boolean $S_c[u]$ indicating the existence of incoming edge at u of color c.

Vertices will be identified by integers $0 \leq i < n$ and edges by their source and color. The three colors are cyclically ordered with operator *next* and *prev* such that $next(\text{red}) = \text{blue}$,

$next$(blue) = black, and $next$(black) = red. Our data structure consists of several arrays of size n:
- three arrays of booleans S_{red}, S_{blue}, and S_{black},
- six arrays of booleans $T^{\text{left}}_{\text{red}}$, $T^{\text{right}}_{\text{red}}$, $T^{\text{left}}_{\text{blue}}$, $T^{\text{right}}_{\text{blue}}$, $T^{\text{left}}_{\text{black}}$, $T^{\text{right}}_{\text{black}}$
- six arrays of vertex indices $U^{\text{left}}_{\text{red}}$, $U^{\text{right}}_{\text{red}}$, $U^{\text{left}}_{\text{blue}}$, $U^{\text{right}}_{\text{blue}}$, $U^{\text{left}}_{\text{black}}$, $U^{\text{right}}_{\text{black}}$, and
- an array P storing the geometric coordinates of the points.

These arrays store the following information (refer to Figure 3) for a vertex u and a color c (the three vertices of the outer face do not have all their outgoing edges and must obey to special rules) :

$S_c[u]$ = `true` if vertex u has incoming edge of color c, `false` otherwise,
$U_c^{\text{left}}[u]$ = `Source(LeftFront(`$u_c^{\nearrow}$`))`, and $U_c^{\text{right}}[u]$ = `Source(RightFront(`$u_c^{\nearrow}$`))`,

Using the coloring rules, these two neighboring edges have only two possible colors: c and $next(c)$ (resp. $prev(c)$) for a left neighbor (resp. right neighbor). Arrays T store that information:

if `Color(LeftFront(`$u_c^{\nearrow}$`))` = c then $T_c^{\text{left}}[u]$ = `true`, else $T_c^{\text{left}}[u]$ = `false`,
if `Color(RightFront(`$u_c^{\nearrow}$`))` = c then $T_c^{\text{right}}[u]$ = `true`, else $T_c^{\text{right}}[u]$ = `false`.

Theorem 2. *Let $\mathcal{G}$ be a triangulation with n vertices. The representation described above requires $6n$ references, while allowing to support* `Target` *operator in $O(d)$ time (when dealing with a degree d vertex) and all other operators in $O(1)$ worst case time.*

Proof. Let us consider operations involving an edge e with source u and target v, whose incident left (resp. right) triangle is (u, v, w) (resp. (u, v, z)).

Operators `Edge`(u) and `Point`(u): to get the index of an edge incident to a given vertex u we simply returns the edge $u_{\text{red}}^{\nearrow}$. The geometric coordinates of vertex u are naturally stored in $P[u]$.

Operator `Source`(e): is a trivial operation since our edges are represented by their source.

Operator `LeftFront`(e): by definition of arrays U and T, `LeftFront`(e) is the edge of source $U^{\text{left}}_{\texttt{Color}(e)}[\texttt{Source}(e)]$ and color `Color`(e) if $T^{\text{left}}_{\texttt{Color}(e)}[\texttt{Source}(e)]$ = `true` and color $next$(`Color`(e)) otherwise.

Operator `LeftBack`(e): We have to distinguish three cases, depending on the color of edges (u, w) and (v, w) (as illustrated in Figure 3):

Case 1: $S_{prev(\texttt{Color}(e))}[\texttt{Source}(e)]$ is false. This case is easy to handle, since (u, w) is the edge with source u and color $next$(`Color`(e)) (in Figure 3(c), there is no incoming blue edge at u, thus `LeftBack`(e) is red and outgoing at u).

Case 2: $S_{prev(\texttt{Color}(e))}[\texttt{Source}(e)]$ is true and $T^{\text{left}}_{\texttt{Color}(e)}[\texttt{Source}(e)]$ is true. Then (w, v) is of color `Color`(e) and (v, w) can be accessed as `LeftFront`(e) and `LeftBack`(e) is the edge with source `Source(LeftFront(`e`))` and color $prev$(`Color`(e)) (there are incoming blue edge at u, thus `LeftBack`(e) is blue. Since `LeftFront`(e) = $\{v, w\}$ is black, it is oriented from w to v and its source is also the searched source of `LeftBack`(e)).

Case 3: $S_{prev(\texttt{Color}(e))}[\texttt{Source}(e)]$ is true and $T^{\text{left}}_{\texttt{Color}(e)}[\texttt{Source}(e)]$ is false, then (v, w) is of color $next$(`Color`(e)) and `LeftBack`(e) is the edge `LeftFront(LeftFront(`e`))`. (as in Case 2

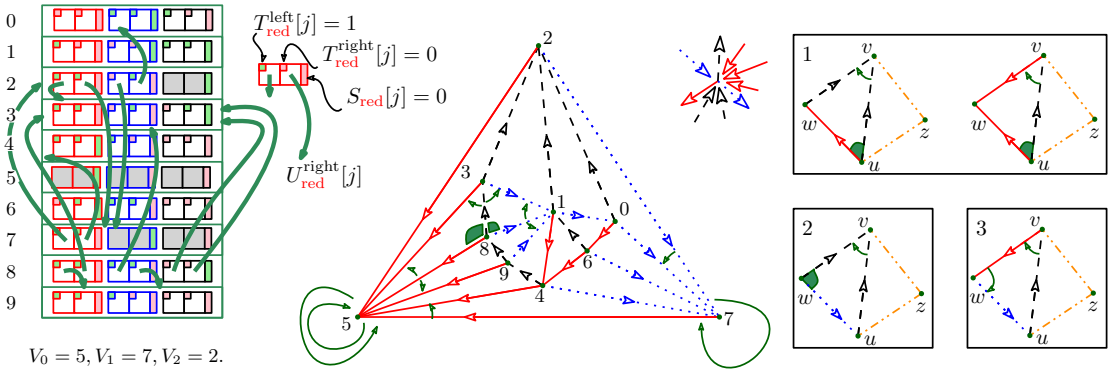


Figure 3: Our first solution. For each vertex we store 6 references, corresponding to the source of the front neighbors of the 3 outgoing edges. Table S, U, T are drawn as an array of size n where booleans are represented by colors (green=`true`) and some vertex indices by arrows. The case analysis of Theorem 2 is illustrated on the right (where edge (u, v) is black).

`LeftBack`(e) is blue, but `LeftFront`$(e) = wv$ is red and oriented from v to w. `LeftBack`(e) is then accessible as `LeftFront(LeftFront`(e)`)`).

Operator `Target`(e): unfortunately we have not stored enough information to return v in $O(1)$ time: the idea is to iteratively turn around vertex v (as described in Figure 2), starting from edge (u, v) in cw direction (or ccw direction) until we get an edge $e' = (v, x)$ having v as source. Then compute `Source`(e') in $O(1)$ time as above, which results in the target of (u, v). This procedures ends after at most $d - 3$ steps (for a vertex v of degree d), since each vertex has 3 outgoing edges.

Operators `RightBack`(e) and `RightFront`(e): observe that the traversal of the right face (u, v, z) incident to (u, v) can be handled in a similar manner as above. Operators `RightBack`(e) and `RightFront`(e) can be deduced by symmetry from operators `LeftBack`(e) and `LeftFront`(e), because of the symmetry of Schnyder woods and of our use of references. □

References Encoding. From a practical point of view, arrays S, T and U can be stored in a single array. Just encode the service bits within the references stored in U, where first k bits of an integer represent the index of a vertex, and the booleans in the other bits of an integer. We can set $k = \lceil \log n \rceil$. We use only less than 2 service bits per integer since we have 9 service bits to associate with 6 vertex indices. Assuming 32 bits integers, we can encode triangulations having up to 2^{30} (1 billion) vertices in 6 arrays of n integers.

2.2 More Compact Solutions, via Maximal Schnyder Woods

Main idea: modify previous scheme to store only one of the two (leftfront and rightfront) neighbors for the blue edges. Again use Schneider woods properties to retrieve missing informations.

In order to reduce the space requirements, we exploit the existence of a special kind of Schnyder wood, called *maximal*, not containing cw oriented triangles:

Lemma 3 ([9]). *Let $\mathcal{G}$ be a planar triangulation. Then it is possible to compute in linear time a Schnyder wood without cw oriented cycles of directed edges.*

New Scheme. We modify the representation described in previous section: the first step is to endow $\mathcal{G}$ with its maximal Schnyder wood (no cw oriented triangles). Outgoing red and black edges will be still represented with two references each, while we will store only one reference for each outgoing blue edge (different cases are illustrated by Figure 4, top pictures). More precisely, let $e = (u, v)$ be an edge having face (u, v, w) at its left and face (v, u, z) at its right, and let q be the vertex defining the ccw triangle (w, v, q). Our data structure still consists of several arrays of size n:
- three arrays of booleans S_{red}, S_{blue}, and S_{black},
- four arrays of booleans $T^{\text{right}}_{\text{red}}$, T_{blue}, $T^{\text{left}}_{\text{black}}$, $T^{\text{right}}_{\text{black}}$
- one array of colors $T^{\text{toleft}}_{\text{red}}$ that can take three values: {red, blue, black}.
- five arrays of vertex indices $U^{\text{toleft}}_{\text{red}}$, $U^{\text{right}}_{\text{red}}$, U_{blue}, $U^{\text{left}}_{\text{black}}$, $U^{\text{right}}_{\text{black}}$, and
- an array P storing the geometric coordinates of the points.

With respect to the simple solution arrays $T^{\text{left}}_{\text{blue}}$, $T^{\text{right}}_{\text{blue}}$, $U^{\text{left}}_{\text{blue}}$, $U^{\text{right}}_{\text{blue}}$, $U^{\text{left}}_{\text{red}}$, and $T^{\text{left}}_{\text{red}}$ are replaced by T_{blue}, U_{blue}, $U^{\text{toleft}}_{\text{red}}$, and $T^{\text{toleft}}_{\text{red}}$ with slightly different content and meaning. This change is emphasize by the change of names. Roughly, only one neighbor of the blue edge is stored, and for the red edge, we may store the second left neighbor (the second edge turning cw around the target) instead of the first one. For a vertex u the following entries have the same meaning as in the previous solution:

$S_c[u]$ = `true` if vertex u has incoming edges of color c, `false` otherwise,

$U^{\text{right}}_{\text{red}}[u]$ = `Source(RightFront(`$u^{\nearrow}_{\text{red}}$`))`,

$U^{\text{left}}_{\text{black}}[u]$ = `Source(LeftFront(`$u^{\nearrow}_{\text{black}}$`))`, and $U^{\text{right}}_{\text{black}}[u]$ = `Source(RightFront(`$u^{\nearrow}_{\text{black}}$`))`,

if `Color(RightFront(`$u^{\nearrow}_{\text{red}}$`))` = red then $T^{\text{right}}_{\text{red}}[u]$ = `true`, else $T^{\text{right}}_{\text{red}}[u]$ = `false`,

if `Color(LeftFront(`$u^{\nearrow}_{\text{black}}$`))` = black then $T^{\text{left}}_{\text{black}}[u]$ = `true`, else $T^{\text{left}}_{\text{black}}[u]$ = `false`,

if `Color(RightFront(`$u^{\nearrow}_{\text{black}}$`))` = black then $T^{\text{right}}_{\text{black}}[u]$ = `true`, else $T^{\text{right}}_{\text{black}}[u]$ = `false`.

Some other entries are modified with respect to the simple version, if (u, v) is a blue edge, we store only one neighbor depending of the existence of red edges incoming at u:

if $S_{\text{red}}[u]$ then
  $U_{\text{blue}}[u]$ = `Source(LeftFront(`$u^{\nearrow}_{\text{blue}}$`))`
  if `Color(LeftFront(`$u^{\nearrow}_{\text{blue}}$`))` = blue then $T_{\text{blue}}[u]$ = `true`, else $T_{\text{blue}}[u]$ = `false`,
else
  $U_{\text{blue}}[u]$ = `Source(RightFront(`$u^{\nearrow}_{\text{blue}}$`))`,
  if `Color(RightFront(`$u^{\nearrow}_{\text{blue}}$`))` = blue then $T_{\text{blue}}[u]$ = `true`, else $T_{\text{blue}}[u]$ = `false`,

If $\{u, v\}$ is a red edge, instead of always storing its first left neighbor, we store either the first left neighbor, either the second left neighbor which makes a total of three cases, $T^{\text{left}}_{\text{red}}$ can now take three values (see Figure 4-top-right).

if $\{w, v\}$ is red then $T^{\text{toleft}}_{\text{red}}[u]$ = red and $U^{\text{toleft}}_{\text{red}}[u] = w$,

if $\{w, v\}$ is blue and $\{v, q\}$ is red then $T^{\text{toleft}}_{\text{red}}[u]$ = blue and $U^{\text{toleft}}_{\text{red}}[u] = v$,

if $\{w, v\}$ is blue and $\{v, q\}$ is black then $T^{\text{toleft}}_{\text{red}}[u]$ = black and $U^{\text{toleft}}_{\text{red}}[u] = q$,

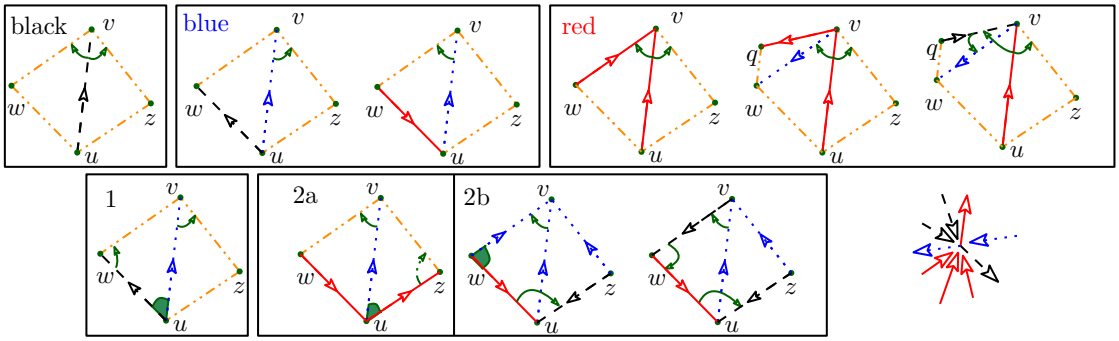


Figure 4: More compact scheme. Neighboring relations between edges are represented by tiny oriented (green) arcs corresponding to stored references, and by filled (green) corners which implicitly describe adjacency relations between outgoing edges incident to a same vertex: because of local Schnyder rule, we do not need to store references between neighboring outgoing edges.

Theorem 4. *Let $\mathcal{G}$ be a triangulation with n vertices. There exists a representation requiring $5n$ references, allowing efficient navigation, as in Theorem 2.*

Proof. We just need to explain how to retrieve the missing `RightFront` or `LeftFront` information. The other operations can be obtained as in the proof of Theorem 2.

We consider an edge e from u to v with incident triangles (v, u, z) on the right and (u, v, w) on the left. We first assume that e is an inner edge (the case of exterior edges is described below).

- If e is black, both `RightFront`(e) and `LeftFront`(e) are directly stored.
- If e is red, `RightFront`(e) is directly stored. `LeftFront`(e) is either directly stored if $T^{toleft}_{\text{red}} \neq$ black or accessible using `RightFront`(e'), where $e' = (q, v)$ is stored, otherwise (Figure 4-top-right).
- If e is blue, one of `RightFront`(e) or `LeftFront`(e) is directly stored and the other can be retrieved as follows (the case analysis below is illustrated by the bottom pictures in Figure 4):

Case 1 $\{u, w\}$ is black (if $S_{\text{red}}[u]$ `=false`) and thus towards w, then `RightFront`(e) is directly stored and `LeftFront`(e) is given by computing `RightFront`(u'_{black}).

Case 2 $\{u, w\}$ is red (if $S_{\text{red}}[u]$ `=true`) and thus towards u, then `LeftFront`(e) is directly stored.

Case 2a $\{u, z\}$ is red (if $S_{\text{black}}[u]$ `=false`) and thus towards z, then `RightFront`(e) can be retrieved as `LeftFront`(u'_{red}).

Case 2b $\{u, z\}$ is black (if $S_{\text{black}}[u]$ `=true`) and thus towards u: since clockwise oriented triangles are forbidden, $\{z, v\}$ is necessarily towards v and thus blue. Edge $\{v, w\}$ can be either black or blue, and is accessible as `LeftFront`(e). Vertex w can be accessed as `Source(LeftFront(LeftFront(`e`)))` if `LeftFront`(e) is black, and as `Source(LeftFront(`e`))` otherwise. Then, we are in the special case where $(w, u) = w'_{\text{red}}$ store a reference to z the source of the second left edge, and $\{z, v\}$ is now accessed as z'_{blue}.

Observe that it is possible to distinguish between cases in $O(1)$ time just reading S and T arrays.

The case of an exterior edge (u, v) incident to the outer face is dealt as follows. If $e = (V_1, V_0)$ or $e = (V_2, V_0)$ then the navigation operators are performed as in Thm 2. If $e = (V_2, V_1)$ then `RightFront` and `RightBack` are performed as described above for the case of inner edges: observe that the result of `RightFront`(u'_{blue}) is stored by construction in $U_{\text{blue}}[u]$. Finally, the two remaining cases are dealt in a special manner: the `LeftFront` and `LeftBack` operators are defined to return v'_{red} and u'_{red} respectively. □

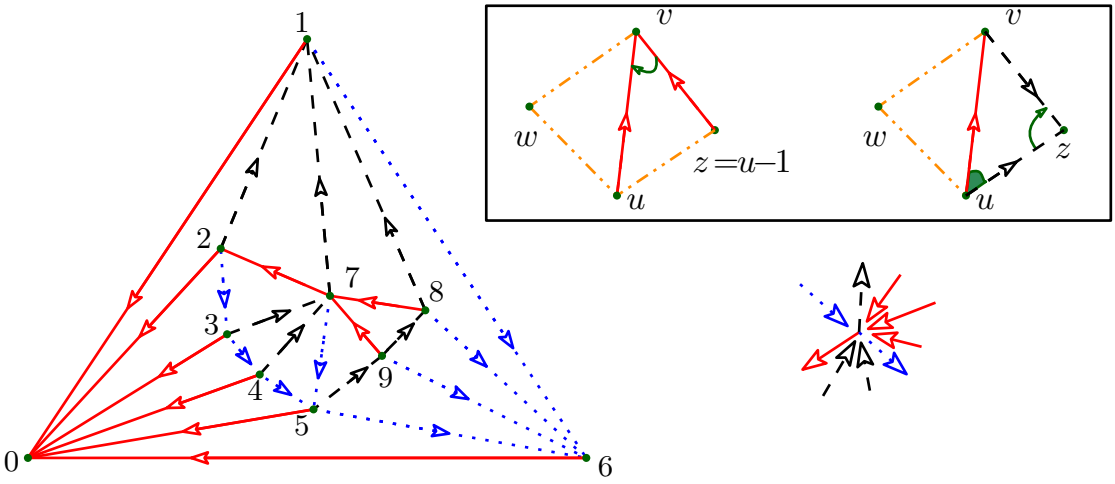


Figure 5: A planar triangulation (endowed with its maximal Schnyder wood) whose vertices are labeled according to a BFS traversal of tree $\overline{\mathcal{T}}_{\text{red}}$ (right). On the right are shown all cases involved in the proof of Theorem 5: we now store only one reference for red edges, since most adjacency relations between edges in $\overline{\mathcal{T}}_{\text{red}}$ are implicitly described by the BFS labels.

2.3 Further Reducing the Space Requirement

Main idea: order the vertices according to the red tree such that some leftfront or rightfront neighbors for the red edges can be found using arithmetic operations instead storing references. Schneider woods properties are still used to retrieve missing informations.

Allowing the exploitation of a permutation of the input vertices (reordering the vertices according to a given permutation), we are able to save one more reference per vertex.

In particular, we need an ordering such that the vertices having the same parent vertex u in $\overline{\mathcal{T}}_{\text{red}}$, will have consecutive numbers (when traversed turning cw around u from u'_{black}): a simple breadth-first search traversal of $\overline{\mathcal{T}}_{\text{red}}$ computes such an ordering (denoted *BFS*). Another possible choice could be the DFUDS ordering used in [5].

Scheme description. We first compute a maximal Schnyder wood of $\mathcal{G}$, and perform a BFS traversal of $\overline{\mathcal{T}}_{\text{red}}$ starting from its root V_0: as $\overline{\mathcal{T}}_{\text{red}}$ is a spanning tree of all vertices of $\mathcal{G}$, we obtain a vertex labeling such that, for every vertex $v \in \mathcal{G}$, the children of v in $\overline{\mathcal{T}}_{\text{red}}$ have consecutive labels (as illustrated in Figure 5). We then reorder all vertices (their associated data) according to their BFS label, and we store entries in table T accordingly. This allows us to save one reference per vertex: we do not store a reference to `RightFront` for edges in $\overline{\mathcal{T}}_{\text{red}}$, which leads us to not store tables $T^{\text{right}}_{\text{red}}$ and $U^{\text{right}}_{\text{red}}$, and retrieve the corresponding information using the ordering of vertices as explained below. All other tables are exactly as in the previous section. We can state the following result (the case analysis is partially illustrated by pictures in Figure 5):

Theorem 5. *Let $\mathcal{G}$ be a triangulation with n vertices. If one is allowed to permute the input vertices (their associated geometric data) then $\mathcal{G}$ can be represented using $4n$ references, supporting navigation as in previous representations.*

Proof. Compared to the representation of Theorem 4 we lose the information concerning `RightFront` for red edges (which was previously explicitly stored in $T^{\text{right}}_{\text{red}}$ and $U^{\text{right}}_{\text{red}}$). An important remark is that, given a red edge $e = (u, v)$, we retrieve its right siblings in $\overline{\mathcal{T}}_{\text{red}}$ just using its BFS label.

More precisely, $\texttt{RightFront}(u^{\nearrow}_{\text{red}})$ is either red or black. If it is red, then it is $(u-1)^{\nearrow}_{\text{red}}$, it is possible to decide if this is the case by checking if $u^{\nearrow}_{\text{red}} = \texttt{LeftFront}((u-1)^{\nearrow}_{\text{red}})$. In the other case, edge $\{u, z\}$ must also be black because clockwise oriented triangles are forbidden, and thus $\texttt{RightFront}(u^{\nearrow}_{\text{red}})$ can be obtained as $\texttt{LeftFront}(u^{\nearrow}_{\text{black}})$. □

2.4 All Operations in $O(1)$ Worst Case Time

We can further exploit the redundancy in our representation in order to improve the computational cost of the navigation: both `Target` and `Adjacent` operators can be supported in worst case constant time. The solution relies on a very simple idea: we add a new table storing explicitly a reference to the target vertex of $u^{\nearrow}_{\text{blue}}$ or $u^{\nearrow}_{\text{black}}$. We also use a service bit to store the parity of the depth of a given vertex in the red tree, and we store the target of $u^{\nearrow}_{\text{red}}$ in $U^{\text{left}}_{\text{red}}$ when the information usually stored there is redundant and can be retrieved by other means. More precisely, we have the following theorem:

Theorem 6. *If one is allowed to permute input points, then there exists a compact representation requiring* $5n$ *references which supports all navigation operators (including* `Target` *and* `Adjacent`*) in worst case* $O(1)$ *time.*

Proof. The data structure of Theorem 5 is modified and extended as follows:
— $U^{\text{left}}_{\text{red}}[u]$ stores $\texttt{Target}(u^{\nearrow}_{\text{red}})$.
Notice that when $\texttt{LeftFront}(u^{\nearrow}_{\text{red}})$ is red, $\texttt{Source}(\texttt{LeftFront}(u^{\nearrow}_{\text{red}}))$ and $\texttt{Target}(u^{\nearrow}_{\text{red}})$ are the same and the content of $U^{\text{left}}_{\text{red}}[u]$ is the same as before.
— A new array of bits Y is created with $Y[u] = \texttt{true}$ if u has even depth in the red tree, $Y[u] = \texttt{false}$ otherwise.
— A new array W storing vertex indices is created. If $S_{\text{red}}[u] = \texttt{false}$ (that is u is a leaf of the red tree), then $W[u] = \texttt{Target}(u^{\nearrow}_{\texttt{Color}(\texttt{LeftFront}(u^{\nearrow}_{\text{blue}}))})$.
If $S_{\text{red}}[u] = \texttt{true}$ (u is not a leaf), then $W[u] = \texttt{Target}(u^{\nearrow}_{c})$, where c = black if $Y[u] = \texttt{true}$ (the depth of u in the red tree is even), and c = blue otherwise.

`Target` operator. We now explain how the `Target` operator can be implemented for each color, and how the `LeftFront` operator is modified for red edges (see Figure 6); the implementations of the other operators remain unchanged.
— $\texttt{LeftFront}(u^{\nearrow}_{\text{red}})$:
If $T^{\text{left}}_{\text{red}}[u] = \textit{true}$ then $\texttt{LeftFront}(u^{\nearrow}_{\text{red}})$ is $(u+1)^{\nearrow}_{\text{red}}$ else $\texttt{LeftFront}(u^{\nearrow}_{\text{red}})$ is stored in $U^{\text{left}}_{\text{red}}[u]$.
— $\texttt{Target}(u^{\nearrow}_{\text{red}})$: the target vertex v is directly stored in $U^{\text{left}}_{\text{red}}[u]$.
— $\texttt{Target}(u^{\nearrow}_{\text{blue}})$.
If $S_{\text{red}}[u] = \texttt{false}$ and $T^{\text{left}}_{\text{blue}}[u] = \texttt{true}$, the result is directly stored in $W[u]$.
If $S_{\text{red}}[u] = \texttt{false}$ and $T^{\text{left}}_{\text{blue}}[u] = \texttt{false}$, the result is $\texttt{Source}(\texttt{LeftFront}(u^{\nearrow}_{\text{blue}}))$.
If $S_{\text{red}}[u] = \texttt{true}$ and $Y[u] = \texttt{false}$, the result is directly stored in $W[u]$.
If $S_{\text{red}}[u] = \texttt{true}$ and $Y[u] = \texttt{true}$ and $T^{\text{left}}_{\text{blue}}[u] = \texttt{false}$,
the result is $\texttt{Source}(\texttt{LeftFront}(u^{\nearrow}_{\text{blue}}))$.
If $S_{\text{red}}[u] = \texttt{true}$ and $Y[u] = \texttt{true}$ and $T^{\text{left}}_{\text{blue}}[u] = \texttt{true}$,
the result is $\texttt{Target}(\texttt{LeftFront}(u^{\nearrow}_{\text{blue}}))$.

Notice that at the last line, we need the `Target` of a blue edge whose source w has odd depth: the result is thus directly stored in W if w is not a leaf of the red tree or computable otherwise.
— $\texttt{Target}(u^{\nearrow}_{\text{black}})$ implementation is similar to the one of $\texttt{Target}(u^{\nearrow}_{\text{blue}})$, up to one forbidden case since there is no cw triangle:
If $S_{\text{red}}[u] = \texttt{false}$ and $T^{\text{left}}_{\text{blue}}[u] = \texttt{false}$, the result is directly stored in $W[u]$.

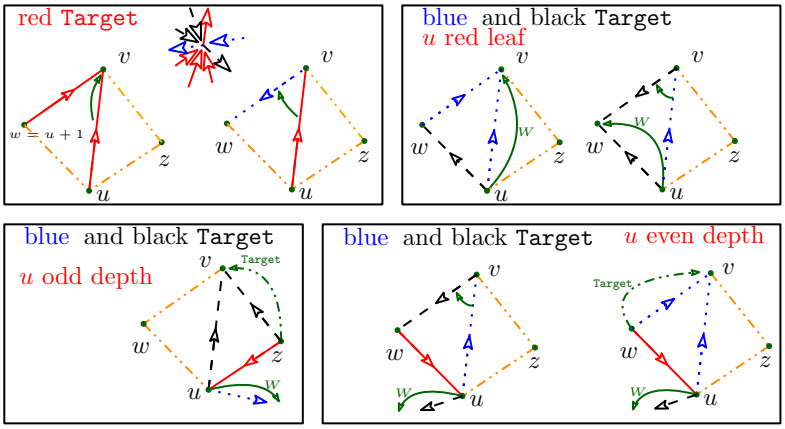


Figure 6: Implementation of the `Target` operator in $O(1)$ time, with $5n$ storage.

If $S_{\text{red}}[u]$ = `false` and $T^{\text{left}}_{\text{blue}}[u]$ = `true`, the result is `Source(LeftFront(`$u^{\nwarrow}_{\text{blue}}$`))`.
If $S_{\text{red}}[u]$ = `true` and $Y[u]$ = `true`, the result is directly stored in $W[u]$.
If $S_{\text{red}}[u]$ = `true` and $Y[u]$ = `false`, the result is `Target(RightFront(`$u^{\nwarrow}_{\text{black}}$`))`.

`Adjacent` operator. The constant time cost of `Target` directly leads to a very simple and $O(1)$ time implementation of the `Adjacent`(u, v) operator. We first get the three target vertices of the 3 edges outgoing from u. These three neighbors of u are retrieved by computing `Target(`$u^{\nwarrow}_{\text{black}}$`)`, `Target(`$u^{\nwarrow}_{\text{blue}}$`)` and `Target(`$u^{\nwarrow}_{\text{red}}$`)`: we check whether one of them does coincide with vertex v. Similarly, we retrieve three neighbors of vertex v by computing `Target(`$v^{\nwarrow}_{\text{black}}$`)`, `Target(`$v^{\nwarrow}_{\text{blue}}$`)` and `Target(`$v^{\nwarrow}_{\text{red}}$`)`, and we check whether one these vertices does coincide with vertex u.

Since in a triangulation endowed with a Schnyder wood each (inner) vertex has outdegree 3, we know that there exists an edge $\{u, v\}$ if and only if one of the 6 tests described above returns a positive answer.

□

2.5 Representing faces

One limitation of the encoding schemes presented at the previous sections concerns the fact that our data structures are edge-based and do not allows us to explicitly represent the triangle faces: this could be a desirable requirement for some geometric processing applications.

In some situation, various kinds of data related to triangles need to be stored (such as face colors, face normals, ...): for this purpose common mesh data structures store, in addition to the incidence relations between edges and vertices, the map between faces and edges. For instance, common implementations of the half-edge data structure [32] store for each half-edge a reference to the incident face, and for each face they store a reference toward one incident half-edge: this requires $(2 + 6)n = 8n$ references that must be added to the $19n$ references of the basic implementation of the half-edge data structure.

In the case of our compact data structures we can represent the map between faces and edges in a more concise way, exploiting again the structural properties of Schnyder woods. In order to construct the mapping from edges to triangles, we first endow the triangulation with its maximal Schnyder wood. Then we match some edges with the triangle that lies at their left. Observe

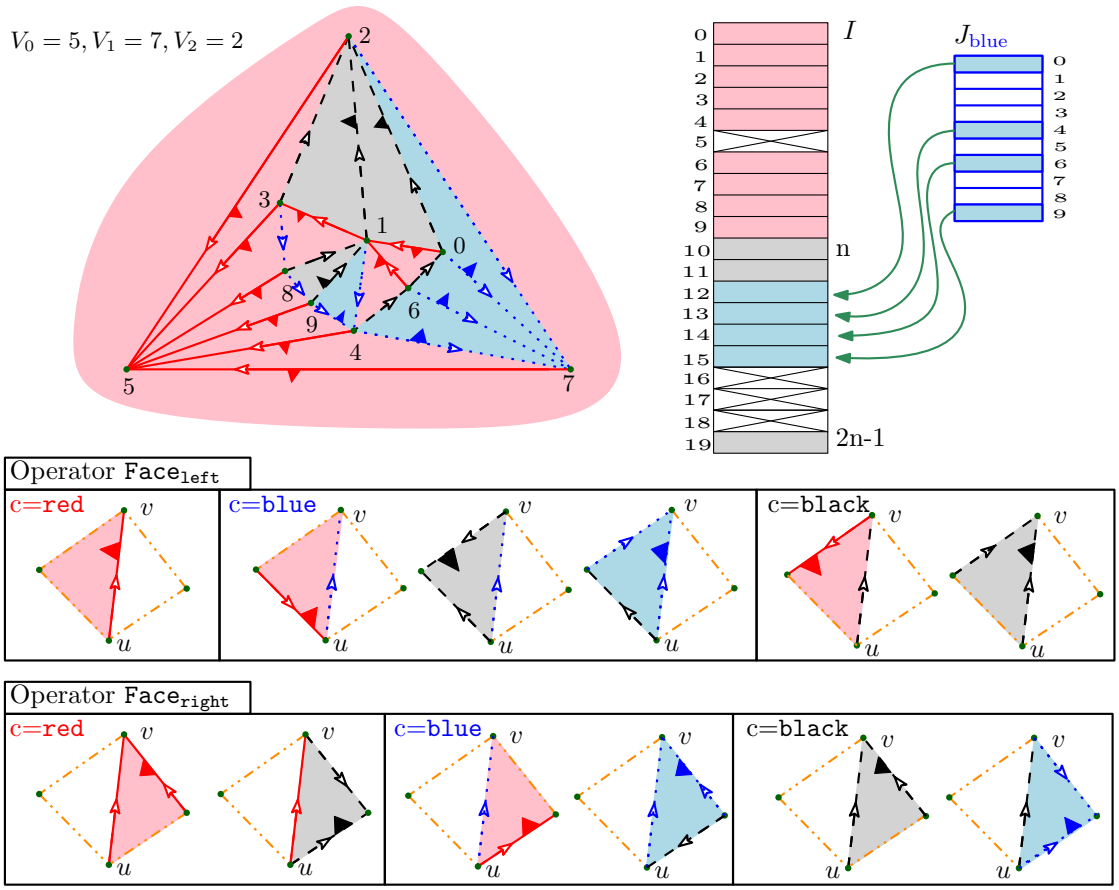


Figure 7: Mapping from edges to triangles (indicated by small triangles). Bottom pictures illustrate the case analysis of Theorem 7.

that, since there are no cw oriented cycles of directed edges, each triangle is at the left to at least one edge.

First, we match all $(n-1)$ red edges with the triangles at their left and call these triangles "red triangles" (see Fig. 7). Second, we match the black edges having a non-red triangle at their left with their left face: these are called "black triangles". Finally, the remaining triangles are called "blue triangles"and must be at the left of one blue edge, and are matched with this blue edge.

The idea is to index the triangles using numbers from 0 to $2n-1$ in the following way: (i) The triangles matched with a red edge get the index of the source of the red edge (all red edges are matched with a triangle). Observe that each red triangle is matched to exactly one red edge: the local property of Schnyder woods ensures that in a triangle incident to two red edges both edges must be oriented toward the same vertex. (ii) The triangles matched with a black edge get the index of the source of the black edge plus n (some black edges remain unmatched) (iii) The triangles matched with a blue edge get the index of the source of an unmatched black edge that must be stored in some auxilliary table.

Theorem 7. *The data structures of Theorems 2, 4, and 5 can be extended to store information*

in triangles using one additional reference per vertex. Access to the information is done in $O(1)$ time.

Proof. A triangle of color c is represented by its corresponding edge $u_c^{\nwarrow}$ in the above matching and denoted $u_c^{\triangle}$. In addition to the input data associated to faces (e.g. face colors or normals), we make use of an additional array that represents the mapping from blue edges to blue triangles. More precisely, we have:
- one array of size n of face indices J_{blue} and
- one array of size $2n$ of face informations I (four entries are unused).

The information stored in the array I can be retrieved with the following rules:

$$\begin{aligned}&\text{data associated to } u^{\triangle}_{\text{red}} \text{are stored in } I[u]\\ &\text{data associated to } u^{\triangle}_{\text{black}} \text{ are stored in } I[n+u]\\ &\text{data associated to } u^{\triangle}_{\text{blue}} \text{ are stored in } I[n+J_{\text{blue}}[u]]\end{aligned}$$

We now explain how to retrieve the indices of the two faces incident to a given edge ($\texttt{Face}_{\texttt{left}}$ and $\texttt{Face}_{\texttt{right}}$ operators, see Figure 7).

$\texttt{Face}_{\texttt{left}}$ operator The $\texttt{Face}_{\texttt{left}}$ operator can be implemented as follows for an edge $u_c^{\nwarrow}$: — If c = red
$\texttt{Face}_{\texttt{left}}(u^{\nwarrow}_{\text{red}})$ is $u^{\triangle}_{\text{red}}$
— If c = black
If $\texttt{LeftFront}(u)$ is red then $\texttt{Face}_{\texttt{left}}(u^{\nwarrow}_{\text{black}})$ is given by $\texttt{Source}(\texttt{LeftFront}(u))^{\triangle}_{\text{red}}$
else $\texttt{Face}_{\texttt{left}}(u^{\nwarrow}_{\text{black}})$ is $u^{\triangle}_{\text{black}}$
— If c = blue
If $\texttt{LeftBack}(u)$ is red then $\texttt{Face}_{\texttt{left}}(u^{\nwarrow}_{\text{blue}})$ is $\texttt{Source}(\texttt{LeftBack}(u))^{\triangle}_{\text{red}}$
else If $\texttt{LeftFront}(u)$ is black then $\texttt{Face}_{\texttt{left}}(u^{\nwarrow}_{\text{blue}})$ is $\texttt{Source}(\texttt{LeftFront}(u))^{\triangle}_{\text{black}}$
else $\texttt{Face}_{\texttt{left}}(u^{\nwarrow}_{\text{blue}})$ is simply $u^{\triangle}_{\text{blue}}$

$\texttt{Face}_{\texttt{right}}$ operator The $\texttt{Face}_{\texttt{right}}$ operator can be implemented as follows for an edge $u_c^{\nwarrow}$: — If c = red
If $\texttt{RightFront}(u)$ is red then $\texttt{Face}_{\texttt{right}}(u^{\nwarrow}_{\text{red}})$ is $\texttt{Source}(\texttt{RightFront}(u))^{\triangle}_{\text{red}}$
else $\texttt{Face}_{\texttt{right}}(u^{\nwarrow}_{\text{red}})$ is $u^{\triangle}_{\text{black}}$ (recall that cw oriented triangles are forbidden).
— If c = black
If $\texttt{RightFront}(u)$ is black then $\texttt{Face}_{\texttt{right}}(u^{\nwarrow}_{\text{black}})$ is $\texttt{Source}(\texttt{RightFront}(u))^{\triangle}_{\text{black}}$
else $\texttt{Face}_{\texttt{right}}(u^{\nwarrow}_{\text{black}})$ is $\texttt{Source}(\texttt{RightBack}(u))^{\triangle}_{\text{blue}}$
— If c = blue
If $\texttt{RightBack}(u)$ is red then $\texttt{Face}_{\texttt{right}}(u^{\nwarrow}_{\text{blue}})$ is $\texttt{Source}(\texttt{RightBack}(u))^{\triangle}_{\text{red}}$
else $\texttt{Face}_{\texttt{right}}(u^{\nwarrow}_{\text{blue}})$ is $\texttt{Source}(\texttt{RightFront}(u))^{\triangle}_{\text{blue}}$

Finally, observe that the two operators above can be performed in worst case $O(1)$ time since their implementation does not involve the $\texttt{Target}$ operator.

$\texttt{Edge}$ operator The $\texttt{Edge}$ operator for a face $u_c^{\triangle}$ trivially returns $u_c^{\nwarrow}$.

To initialize J_{blue} we need an extra array of booleans K of size n (Array I can be used for this purpose since I is not yet initialized at that step). Array K is initialized at $\texttt{true}$ at the beginning. Then in a first step: for each black edge $u^{\nwarrow}_{\text{black}}$ if $\texttt{Face}_{\texttt{left}}(u^{\nwarrow}_{\text{black}}) = u^{\triangle}_{\text{black}}$ we set $K[u] = \texttt{false}$. In a second step: for each blue edge $u^{\nwarrow}_{\text{blue}}$ we set $J[u] = v$ such that $K[v] = \texttt{true}$ and we set $K[v] = \texttt{false}$. □

Notice that if storing information in faces is not needed, it is still possible to design a procedure to iterate over all faces in constant time per face without using additional storage. A simple solution consists in performing a DFS traversal of the dual graph, avoiding to traverse the edges of the red tree: this is similar to the traversal performed to compute the binary encoding string that wil be described in Section 3.1.

3 Decoding the Triangulation from a Compressed Format

A main issue common to many compact data structures [13, 12, 26, 29, 27], is that an explicit representation of the entire mesh must be kept in main memory during the whole construction phase: this is needed, for example, to process vertices and edges (or faces), which must be re-numbered according to a prescribed mesh traversal. This preliminary construction phase can greatly increase the overall memory requirements, especially for very huge meshes: in addition to the space storage of the compact data structure to construct (between $4n$ and $8n$ references for most compact representations), one has to use between $13n$ and $19n$ references for an explicit standard representation (such as *Corner Table* or *Half-edge*), and a few additional memory references for the implementation of the mesh processing (typically, a graph traversal). To address this issue, one could take advantage of the existence of various and efficient compressed formats for triangle meshes, which have been designed in order to store a triangulation on disk or to send it on the network.

In this section we show how to save our array-based compact representations in a compressed format so that we can reconstruct on the fly our structure in linear time, without any extra memory cost and in a streamable fashion. The first construction of our representation still need an explicit representation, in particular for the computation of the Schnyder wood. We make use of a compressed format for triangle meshes which is so far a standard tool in the domains of graph encoding and graph enumeration. This format allows coding/decoding a planar triangulation of size n with less than $4n$ bits. The encoding scheme, originally designed for the planar case [30], relies on the combinatorial properties of Schnyder woods (or the related *canonical orderings* structure) and provides a bijection between the set of all Schnyder woods of a given planar triangulation and pair of non-crossing Dyck paths [6]. More recently this scheme has been generalized for dealing with genus g triangulations [17].

3.1 Encoding Scheme

For the sake of completeness, we provide a concise overview of the encoding scheme for planar triangulations (a more detailed presentation can be found in [6]).

We start with a planar triangulation with n vertices, endowed with an arbitrary Schnyder wood, and we will produce a binary encoding of length $4n-3$, obtained as follows. First construct a (standard) balanced parenthesis word encoding the combinatorics of the red tree: just perform a depth-first traversal of of $\overline{\mathcal{T}}_{\text{red}}$ starting from its root V_0 (in ccw order). Each time a new vertex is reached during this traversal a symbol (is produced; while each time the traversal goes back to a parent, a symbol) is produced. This procedure gives a word of $2n$ bits, called the red word in the sequel. Then we construct a black word, that stores in unary representation the number of incoming black edges (the incoming black degree of a vertex) for all vertices in $\mathcal{G}$. These numbers (one for each vertex) are separated by a '0' symbol: vertices are naturally ordered according to the depth-first traversal of the red tree (as illustrated by the left picture in Fig. 8). Observe that the combination of the red and black words, together with the local Schnyder condition, represent the red tree enriched with the information concerning the locations of starting and ending places of black edges (depicted as broken black arrows in the middle picture of Fig. 8).

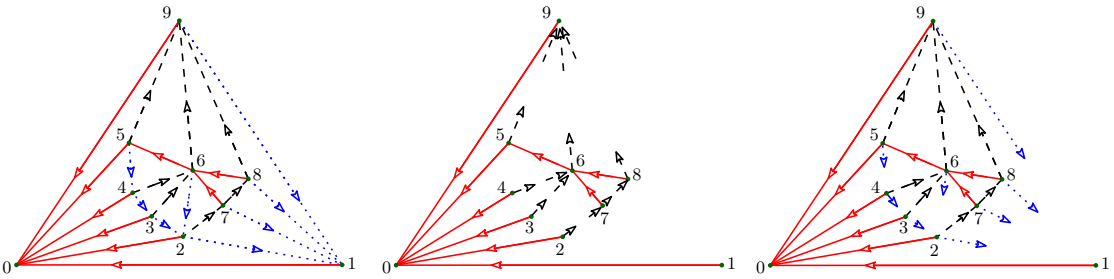


Figure 8: Encoding/decoding phase. A triangulation (endowed with a Schnyder wood orientation) is encoded by a pair of binary words, encoding respectively the red tree $\overline{\mathcal{T}}_{\text{red}}$ and the tails of black edges (center). The information concerning blue edges is redundant and can fully retrieved by applying the local Schnyder condition (right).

The size of the final encoding can be easily evaluated, as follows. The number of 1 bits in the black word matches the number of black edges in the tree $\mathcal{T}_{\text{black}}$: recall that $\mathcal{T}_{\text{black}}$ has $n-2$ vertices and thus $n-3$ edges, which leads to a total length of $2n-3$ symbols, as the black word contains a 0 bit for each vertex. The red word has two parentheses per vertex, that is a length of $2n$. The total length of the encoding, concatenation of the black and red words, is thus $4n-3$.

The string below corresponds to the encoding of the triangulation in Figure 8 (this is the same triangulation of the example in Figure 3, where vertices are re-ordered according to a depth-first traversal of $\overline{\mathcal{T}}_{\text{red}}$ in ccw order):

(( ) ( ) ( ) ( ) ((( ) ( ))) ( )) 00000011010101110

For the sake of clarity, we can decorate this code by vertex indices:

$(_0\ (_1\ )_1\ (_2\ )_2\ (_3\ )_3\ (_4\ )_4\ (_5\ (_6\ (_7\ )_7\ (_8\ )_8\ )_6\ )_5\ (_9\ )_9\ )_0\ 0_0\ 0_1\ 0_2\ 0_3\ 0_4\ 0_5\ 110_6\ 10_7\ 10_8\ 1110_9$.

In the example above the black word encodes the in-black-degrees of vertices, which are respectively 0, 0, 0, 0, 0, 0, 2, 1, 1, 3.

3.2 Decoding Phase

The decoding procedure is in two steps. At the first step, we use the `readNextRedSymbol()` to read the red word, and the operator `readBlackInDegree()` to retrieve the incoming black degree of vertices (stored in the black word). The linear scan of the red word allows the construction of the red tree by inserting the vertices (numbered according to a depth-traversal) into a red stack: in this way we fill all red columns of array T and U. In parallel, we can read the black word which allows us to fill the black columns of array T and U. No extra memory is required for an explicit storage of the red stack: as the blue columns are not involved during this first step, one blue column can be used to provide an array-based implementation of such a stack.

Finally, in a second step that will be detailed later, the array of vertices is scanned retrieving the information about blue edges.

Constructing Red and Black Trees. More precisely, the red tree is entirely defined by the red word. By the coloring rule, we know that a red edge from u to v is such that `LeftFront` is either a red edge incoming at v or a blue edge outgoing at v. Symmetrically, `RightFront` is either a red edge incoming at v or a black edge outgoing at v. At the first look, you can skip

the call to ConstructBlackTree in the algorithm below and get a program that creates the red columns of T. Observe that when the red tree is traversed in depth first order, the source of a black edge is always visited before its target.

We present the algorithm as recursive for the ease of comprehension, but notice that this function does not use any implicit memory stack, we have made all memory explicit in the red and black stacks. See Figures 9 and 10. Notice that a vertex is not simultaneously in the two stacks (except from the top) and thus one blue array has enough memory to implement the two stacks.

Constructing the Blue Tree When considering the red and black trees together, one obtains a planar map (an embedded graph whose faces are homeomorphic to a disk) where only blue edges are missing (as depicted in the rightmost picture of Figure 8). In the standard case where one is provided with a complete representation of such a map (as in [30, 6]) retrieving the location of blue edges is straightforward: the source vertex of a blue edge is known by applying the local Schnyder rule, and edge destinations can be recovered by performing a facial walk of each red/black face. In our case such a solution cannot be applied: after the first decoding step the red and black trees are recovered, but only a partial knowledge of the red/black map is available.

Our strategy is slightly different, and consists of iteratively discovering and visit in cw order the blue edges incident to a given vertex x (this procedure is performed independently for each vertex). The code of the procedure `constructBlueTree` computing blue edges is illustrated by Figure 11. The code enumerates the blue edges with target x clockwise around x. Let (u, x) be a blue edge, we are searching $\{v, x\}$ the next edge cw around x. There are four cases depending of the color of $\{u, v\}$ (red or black) and of of the color of $\{x, v\}$ (blue or black). This can be done using Schnyder rules as described by Figure 11. The three vertices $V_0 = 0$, $V_1 = 1$, and $V_2 = n - 1$ are treated in a special manner.

Decoding for $5n$ Version The above `ConstructBlueTree` decoding algorithm produces the $6n$ version of the data structure described in Section 2.1. To produce the $5n$ version of Section 2.2, the algorithm `ConstructRedTree` remains almost unchanged, except that $U_{\text{red}}^{\text{left}}$ is replaced by $U_{\text{red}}^{to\text{left}}$ and $T_{\text{red}}^{\text{left}}$ is replaced by $T_{\text{red}}^{to\text{left}}$ (taking the values red or blue instead of `true` or `false` respectively). Then, we have to modify the function `ConstructBlueTree` as described in `ConstructBlueTree5n` (Figure 12) to update the left red references in the relevant way, and to store only one of the two blue references.

3.3 Streamable Encoding Scheme

As described above, the first decoding phase (construction of red and black trees), requires a parallel reading of the two black and red words, or to perform a linear scan of the whole encoding in two passes: to first construct the red tree, and then to recover black edges (with a second linear scan). In practice, this can be a limitation in the case of streaming applications. In order to avoid this problem, and to make our data structure fully streamable, we can slightly modify our encoding scheme, by interleaving the symbols in the red and black words.

More precisely, the bits in the red and black words can be mixed in a single binary word as follows. We start with the encoding of the red tree, where (and) symbols are replaced by 0 and 1 bits respectively. Let us assume we are visiting and encoding a vertex v: just after the 0_v bit corresponding to the first visit of v, we encode the black in-degree of v by writing a block consisting of d 1 bits followed by a 0 bit. The mixed code corresponding to the example of Figure 8 is given below

$0_0\ 0_0\ 0_1\ 0_1\ 1_1\ 0_2\ 0_2\ 1_2\ 0_3\ 0_3\ 1_3\ 0_4\ 0_4\ 1_4\ 0_5\ 0_5\ 0_6\ 110_6\ 0_7\ 10_7\ 1_7\ 0_8\ 10_8\ 1_8\ 1_6\ 1_5\ 0_9\ 1110_9 1_9\ 1_0.$

```
// N is a global counter to index vertices, initialized at 0
// before the first call, 0 is pushed in the red stack, and first ( already read.
ConstructRedTree()
u = top( red stack ) ;                    // Construct tree rooted at u
d = readBlackInDegree();
ConstructBlackTree(u, d);
readNextRedSymbol();
if " ) " ;                                        // u has no red child
 then
    S_red[u] = false;
else
    S_red[u] = true;
    N = N + 1; v = N ;                          // first red child of u
    U^left_red[v] = u; T^left_red[v] = false;
    LastRedEdge=false;
    while LastRedEdge == false do
        push(v, red stack);
        ConstructRedTree() ;                          // tree rooted at v
        v=pop(red stack); u=top(red stack);
        readNextRedSymbol();
        if " ( " then
            N = N + 1; w = N;                       // next red child of u
            U^left_red[w] = v; T^left_red[w] = true;
            U^right_red[v] = w; T^right_red[v] = true;
            v = w;
        else
            LastRedEdge = true ;                      // to exit the loop
        end
    end
    // v is the last red child of u
    U^right_red[v] = u; T^right_red[v] = false;
    if u == 0 then
        // V_0 does not have outgoing black edge
        U^right_red[v] = 1; T^right_red[v] = true;
    end
end
if u > 1 then
    // V_0 and V_1 do not have outgoing black edges
    push(u, black stack);
end
```

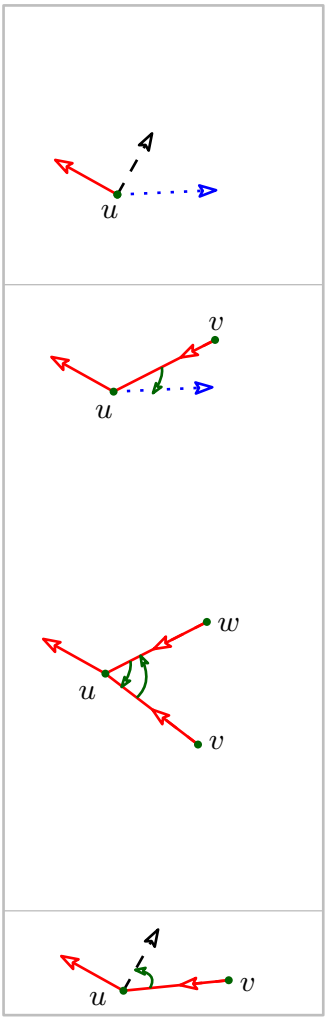


Figure 9: Decoding algorithm: recursive construction of the red and black trees in parallel.

```
ConstructBlackTree(u, d)
if d == 0 ;                                   // u has no black child
 then
    S_black[u] = false;
else
    S_black[u] =true;
    d = d − 1; pop(x, black stack) ;          // first black child
    U^left_black[x] = u; T^left_black[x] = false;
    while d > 0 do
        y = x; d = d − 1; pop(x, black stack);
        U^left_black[x] = y; T^left_black[x] = true;
        U^right_black[y] = x; T^right_black[y] = true;
    end
    U^right_black[x] = u; T^right_black[x] = false ;   // last black child
end
```

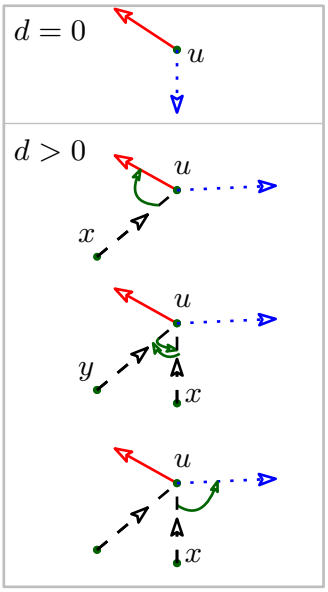


Figure 10: This procedure performs the construction of the children of a vertex u having d incoming black edges in $\mathcal{T}_{\text{black}}$.

where we provide colors and subscripts just to help intuition. It is easy to distinguish between red and black symbols, since during the decoding phase we perform a linear scan of this mixed word, by taking into account the Schnyder local rule. We start by reading the first three red symbols, mixed to two black 0 bits, which corresponds to the first edge (V_0, V_1) that has not incoming black edges. For any other vertex u (different from V_0 and V_1), when after visiting its outgoing red edge we turn around u in ccw order, we may encounter a (possibly empty) sequence of black incoming edges.
– The red 0_u symbol must be followed by a block of black bits: a sequence (possibly empty) of black 1 bits, ended by a black 0_u bit, encoding the (possibly empty) group of incoming black edges.
– So, the black 1 symbol is always followed by a black bit.
– The red 1_u symbol must by followed red bit. This is either a red 0_v symbol (if u has another sibling vertex v in $\mathcal{T}_{\text{red}}$) or a red 1_x symbol (if x, the parent of u, has no other children).
– The black 0_u symbol is followed by a red bit: either a 1_u bit if u has no children, or a 0_v bit otherwise, where v is the first child of u.

Thus with a simple linear scan the `readNextRedSymbol` and `readBlackInDegree` operators can be supported on the mixed encoding described above, allowing the construction of the red and black trees simultaneously.

4 Experimental Results

Settings and datasets. We have written Java implementations of the processing algorithms and compact data structures presented in this work.[3] We performed tests on a wide collection

[3] A pure Java implementation of our algorithms and data structures, as well as input meshes in compressed format, are available at `www.lix.polytechnique.fr/~amturing/software.html`.

```
ConstructBlueTree()
for x = 2 to N − 2 do
    if U^right_red[x] == U^left_black[x] and T^right_red[x] ≠ T^left_black[x] then
        // x has no blue child
        S_blue[x] = false;
    else
        S_blue[x] = true;
        if T^right_red[x] ;                          // RightFront is red or black?
         then
            u = U^right_red[x];
        else
            z = U^right_red[x];
            u = U^right_black[z] ;
        end
        // (u, x) is the first blue edge towards x
        U^right_blue[u] = x; T^right_blue[u] = false;
        LastBlueEdge = false;
        while LastBlueEdge==false do
            // {v, x} will be the next edge cw around x;
            if S_red[u] ;                                        // (v, u) is red
             then
                v = u + 1 ;                              // first red child of u
                if U^left_black[x] = v and T^left_black[x] = false then
                    U^left_blue[u] = x;T^left_blue[u] = false;        // (u, x) last blue edge
                    LastBlueEdge=true;
                else
                    U^left_blue[u] = v;T^left_blue[u] = true;
                    U^right_blue[v] = u;T^right_blue[v] = true;
                end
            else
                // (u, v) is black
                if U^left_black[x] ≠ u then
                    v = U^right_black[u];
                    U^left_blue[u] = v;T^left_blue[u] = true;
                    U^right_blue[v] = u;T^right_blue[v] = true;
                else
                    U^left_blue[u] = x;T^left_blue[u] = false;        // (u, x) last blue edge
                    LastBlueEdge=true;
                end
            end
            u = v;
        end
    end
end
```

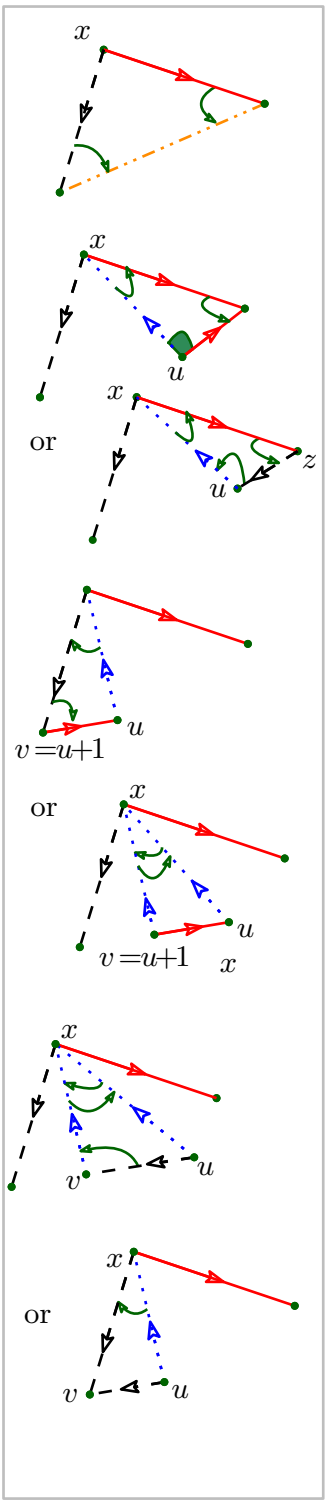


Figure 11: Decoding algorithm: third phase.

```
ConstructBlueTree5n()
for x = 2 to N − 2 do
    if U_red^right[x] == U_black^left[x] and T_red^right[x] ≠ T_black^left[x] then
        // x has no blue child
        S_blue[x] = false;
    else
        S_blue[x] = true;
        u = U_red^right[x] ;                          // (u,x) is the first blue edge towards x
        ColorLastEdge=red ;                           // RightFront must be red (no cw triangles)
        z = x;
        LastBlueEdge = false;
        while LastBlueEdge==false do
            // {v,x} will be the next edge cw around x;
            if S_red[u] = true // check whether (v,u) is red
             then
                v = u + 1 ;                           // v is the first red child of u
                if ColorLastEdge==black then
                    U_red^toleft[v] = z; T_red^toleft[v] = black;          // Modified red ref
                end
                ColorLastEdge=red;
                if U_black^left[x] = v and T_black^left[x] = false then
                    LastBlueEdge=true ;                       // (u,x) last blue edge
                    U_blue[u] = x; T_blue[u] = false ;        // blue ref is to the left
                else
                    U_blue[u] = v; T_blue[u] = true;
                    if S_red[v] = false then
                        U_blue[v] = u; T_blue[v] = true;
                    end
                end
            else
                ColorLastEdge=black ;                         // (u,v) is black
                U_blue[u] = z ;                               // blue ref is to the right
                if z = x then
                    T_blue[u] =false ;                        // (u,x) is first blue child
                else
                    T_blue[u] =true;
                end
                if U_black^left[x] ≠ u then
                    v = U_black^right[u];
                    if S_red[v] = false then
                        U_blue[v] = u; T_blue[v] = true;
                    end
                else
                    LastBlueEdge=true ;                       // (u,x) last blue edge
                end
            end
            z = u;
            u = v;
        end
    end
end
```

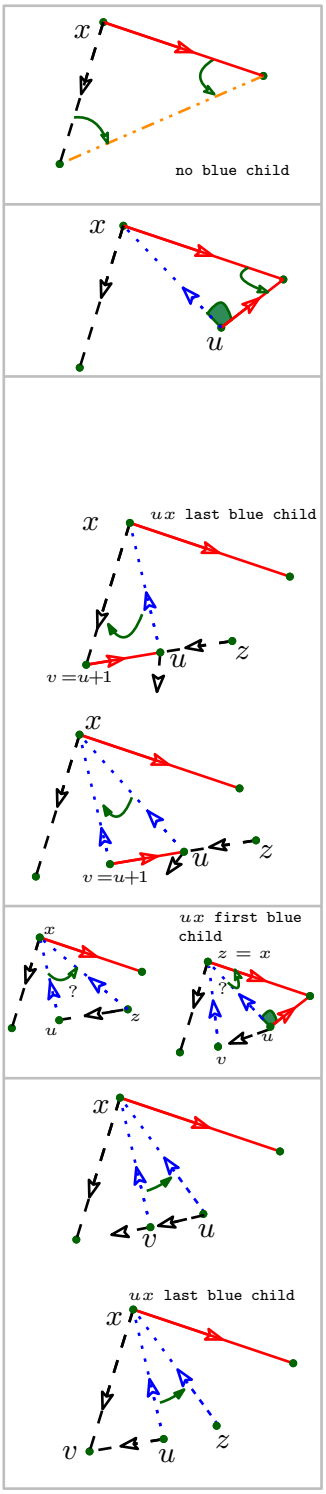


Figure 12: Decoding algorithm: third phase for the 5n version. Changes with respect to function `ConstructBlueTree` are in green.

			Construction from binary OFF				Decoding	
Mesh type	vertices	faces	building DS from OFF	comput. SW	building CDS6n	compression	reading compres.	decoding CDS6n
Egea	8268	16K	0.015	0.005	0.007	0.016	0.007	0.013
Bunny	26002	52K	0.046	0.010	0.012	0.038	0.009	0.022
Iphigenia	49922	99K	0.063	0.019	0.019	0.044	0.011	0.30
Camille's hand	195557	391K	0.154	0.036	0.080	0.080	0.014	0.055
Eros	476596	953K	0.30	0.06	0.14	0.12	0.030	0.091
Chinese Dragon	655980	1.3M	0.53	0.11	0.17	0.17	0.046	0.118
Pierre's hand	773465	1.5M	0.55	0.12	0.18	0.22	0.049	0.12
Isidore horse	1.1M	2.2M	0.69	0.18	0.28	0.26	0.050	0.14
Hand (poisson)	1.6M	3.3M	1.01	0.24	0.35	0.39	0.078	0.16
Random 2.5M	2.5M	5M	1.75	0.30	0.46	0.45	0.082	0.28
Random 5M	5M	10M	3.97	0.58	0.93	0.77	0.127	0.50
Random 7.5M	7.5M	15M	5.15	0.86	1.36	1.23	0.161	0.75
Random 10M	10M	20M	8.16	1.16	1.49	1.50	0.185	0.96
Random 12.5M	12.5M	25M	30.39	1.54	2.36	2.18	0.236	1.21
Random 15M	15M	30M	32.58	1.83	2.68	2.32	0.314	1.47

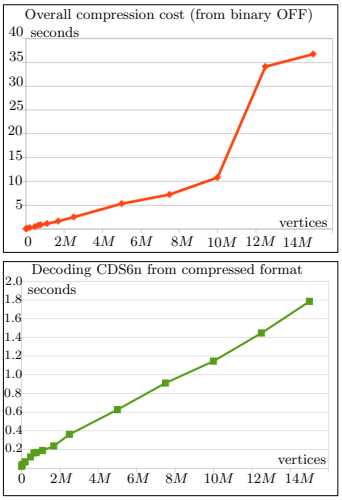


Table 2: (left) This table reports the runtime performances of all steps involved in the construction, compression and decoding phases of our compact data structure using $6n$ references (CDS6n). (right) The red chart corresponds to the overall cost of the compression phase (sum of the first, second and fourth columns), while the green chart corresponds to the whole decoding phase (sum of the last two columns). All results are expressed in seconds and represent the average performances obtained with several runs of our tests. In our tests we allocate 6GB of RAM memory for running the construction and compression steps (we use `-Xms6G -Xmx6G` as argument for the Java Virtual Machine). We run the decoding from compressed format allocating 800MB of RAM for the JVM.

of meshes, whose sizes range from few thousands to millions of vertices, to evaluate both the construction and navigation time performances. Our tests involves various kinds of datasets, including standard 3D models[4] homeomorphic to a sphere as well as planar random triangulations generated with the uniform random sampler by Poulalhon and Schaeffer [38]. All our tests are run on a HP EliteBook, equipped with 8GB of RAM and an Intel Core i7 2.60GHz, running under linux (Ubuntu 16.04) and with Java 1.8 64-bit.

4.1 Preprocessing: construction vs. decoding.

We first evaluate the runtime performances of our data structures concerning the construction phase. Table 2 shows the construction costs (expressed in seconds) of the data structure using $6n$ references (referred to as *CDS6n*, Thm 2). The runtime costs reported in the first three columns correspond to the different steps in the construction phase of our data structure starting from a binary OFF file [5], which is a standard format storing both the geometry (3D vertex locations) and the mesh connectivity (the mesh is stored with a shared vertex representation). The input OFF file is read with a linear scan in order to construct on the fly, from a shared vertex representation, an half-edge data structure (using Java references) storing the triangulation in main memory (first column, *building DS from OFF*).

[4] Most of them are made available in standard format by the AIM@SHAPE Shape Repository

[5] In order to obtain a fair comparison, all input data (the OFF files as well as the files storing the compressed format described in Section 3.1) are stored using a binary encoding: all integer references and vertex coordinates are stored on 32 bits each.

The mesh is then processed in order to endow the triangulation with a Schnyder wood orientation (second column *comput. SW*), and finally arrays T, U and S are allocated to build the compact data structure (column *building CDS6n*). The overall cost of the whole pre-processing phase is given by the sum of the timing values in the first three columns. As illustrated by the performances reported in Table 2, the timing cost is dominated by the construction of the mesh representation in main memory: this is the unique bottleneck of our pre-processing phase. Observe that most compact data structures [29, 26, 27, 28, 13] have the same limitation.

The runtime performances of the compression algorithm are similar to the ones of the construction phase: the main difference is that it is not necessary to build our compact data structure in main memory, since the compressed file format can be obtained directly from the explicit (non compact) representation of the triangulation once it has been endowed with a Schnyder wood orientation. Thus the total cost for encoding a triangulation into a compressed format of size at most $4n$ bits, is given by the sum of the first two columns plus the fourth column (*c*ompression) that measures the times for both encoding the triangle mesh and for outputting the result into a binary compressed file (the runtime costs of the construction/compression phase in Table 2 are obtained allocating 6GB of RAM for the Java Virtual Machine). The reported runtime costs confirm the asymptotic linear time behaviour of all steps of our algorithms (see the red chart in Fig. 2, reporting the overall cost of the compression phase).

One can observe that runtime performances get worse for large meshes ($\geq 25M$ faces): this concerns only the construction of the mesh representation in main memory. As observe monitoring the memory usage during our benchmarks, the main reason of such poor performances relies on expensive execution of the garbage collector of the JVM: unfortunately it is not possible to disable the Java garbage collector, whose execution follows a non-deterministic behaviour.

Decoding the compressed format One main advantage of our encoding/decoding algorithm (Section 3), is that the mesh can be directly constructed from a compressed input file, without using additional memory for an intermediate representation and without computing the Schnyder wood, which leads in overall to a smaller construction cost. The last two columns of Table 2 report the runtime costs of the construction from compressed format: the scan of the input file and the decoding phase of Section 3.2. The compressed storage format encoding the connectivity described in Section 3.1 is much more compact compared to the binary OFF format: as each face stores 3 integers (each on 32 bits), the shared vertex representation requires for connectivity $2 \cdot 3 \cdot 32 = 192$ bits per vertex, while our compressed format requires about 4 bits per vertex. As a consequence, the linear scan of the compressed input binary file is largely dominated by the decoding of vertex coordinates (geometric coordinates are not compressed, but encoded on 32 bits each using simple float precision).

We run our tests using the argument `-Xss100m` for the JVM to allow enough memory for the recursive decoding procedure (the number of recursive calls depends on the height of the red tree): nevertheless, as observed in Section 3.2, the decoding does only make use of two stacks and could be implemented in a not recursive way.

Our results confirm the linear time complexity of the whole decompressing phase: decoding a triangulation stored in compressed format is extremely fast and can be performed using little memory requirements even for large meshes (the runtime costs of the decoding step in Table 2 are obtained allocating 800MB of RAM for the JVM).

4.2 Mesh navigation.

In order to evaluate the runtime performances of our representations, we have implemented Java array-based versions of two standard (non-compact) representations: Table 3 and Table 13 report

Mesh type	H-E (19n)	W-E (19n)	CDS6n (slow) Thm 2	CDS6n Thm 2	CDS5n Thm 4	CDS4n Thm 5
Egea	25	30	52	41	81	75
Bunny	29	33	45	34	74	64
Iphigenia	27	31	50	38	77	57
Camille's hand	87	62	47	34	57	53
Eros	24	20	38	28	63	45
Chinese dragon	24	22	41	30	56	44
Pierre's hand	15	14	32	22	57	43
Isidore horse	20	19	37	27	56	57
Hand (poisson)	27	27	51	39	67	61
Random 100K vert.	34	29	55	46	76	71
Random 2.5M vert.	38	33	62	49	93	91
Random 5M vert.	39	33	63	51	93	91

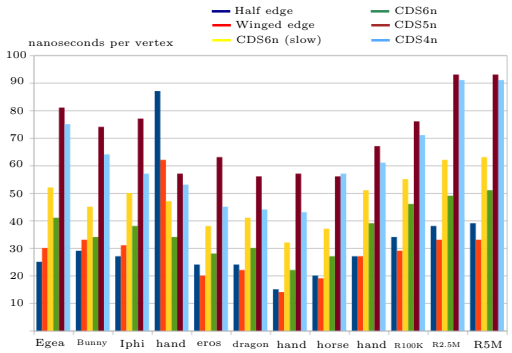


Table 3: Vertex degree computation: this table reports the average runtime performances of our compact data structures compared to array-based implementations of the *Half-edge* and *Winged-edge* data structures (all results are expressed in nanoseconds per vertex).

comparisons with the *Half-edge* and *Winged-edge* data structures.

As in previous works [10, 29, 26, 27, 28, 13] we consider two standard processing procedures: computing *vertex degrees* (involving edge navigation) and *vertex normals* (involving vertex access operators and geometric calculations). In all tests we allocate enough memory so that all representations of the tested 3d models fit in main memory; vertices are accessed sequentially according to their original order in the input mesh (except for the data structure using $4n$ references of Thm 5, where vertices are re-ordered according to the BFS traversal of the red tree). All results reported in Table 3 and Fig 13 are expressed in nanoseconds per vertex, and represent the average runtime performances computed by repeating hundreds of our tests (we run a warming phase in order to avoid initialization costs).

We have two implementations of the compact data structure described in Thm 2. In the first implementation (referred to as *CDS6n slow*) an edge (u, v) of color c is represented with a 32 bits integer encoding the pair (u, c), as detailed in Section 2: the less significant bits are service bits encoding the color, while the remaining part of the reference stores the number of the source vertex u. Both service bits and numbers are retrieved with a combination of bit shifts and masks. We also have another implementation (called *CDS6n*), which is slightly more efficient in practice, where tables U stores directly an edge index (a number between 0 and $3n - 1$): as the three edges outgoing from a vertex u have consecutive numbers $3u$, $3u + 1$ and $3u + 2$, the retrieval of edge colors can be done by performing modulo operations, while the vertex source is computed by integer divisions (as presented in [11]). Analogously, the implementations of the data structures described by Thm 4 and Thm 5, using respectively 5 and 4 references per vertex, are called *CDS5n* and *CDS4n* and store edge numbers (refer to Table 3 and Figure 13). We would like to point out that the main goal of our implementations is to show the practical interest of our compact data structures, whose runtime performances are comparable to the ones of non-compact data structures. But there is room for improvement: a careful optimization of the elementary operations involving the reference encoding could lead to slightly better navigation costs.

As one could expect, non-compact mesh representations are faster in most cases (see the runtime performances listed in Table 3). Our data structures are even faster in some cases (*Camille's hand* model). The degraded performances of navigation of standard data structures are explained by the bad vertex ordering in the original model (this leads to bad locality for both vertex and edge storage in main memory). In our compact data structures, while keeping the

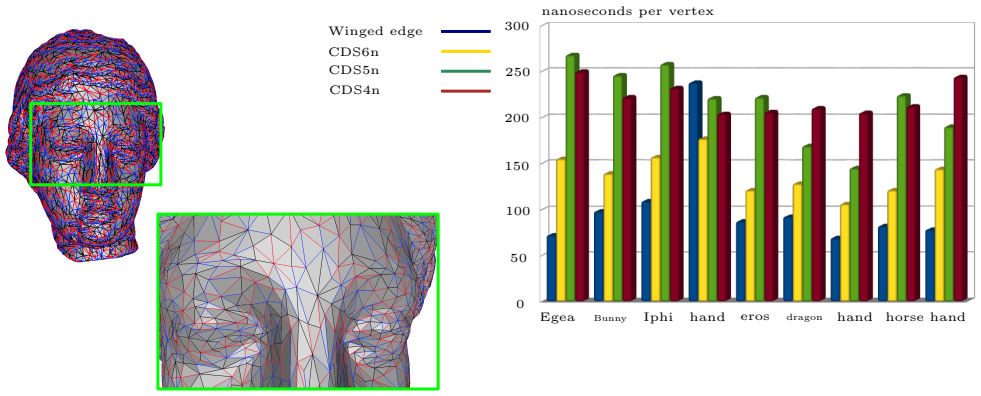


Figure 13: This chart reports the comparison of the average runtime performances, expressed in nanoseconds per vertex, concerning the vertex normal computation (all calculations are done with simple float precision). Experiments are obtained allocating 1GB of RAM (using `-Xms1G -Xmx1G` as argument for the JVM). The left picture shows the Egea mesh, endowed with a Schnyder wood edge coloration.

same original vertex ordering (for *CDS6n* and *CDS5n*), we can take advantage of the locality of edge ordering, since the 3 outgoing edges incident to a vertex are stored consecutively. In overall our data structures achieve good trade-offs between space usage and runtime performances. While being between 3.17 and up to 4.75 times more compact for connectivity than winged-edge or half-edge, our structures are slightly slower: when compared to *Winged-edge* they lose in average on the tested meshes a factor 1.33 for *CDS6n*, 2.65 for *CDS5n* and 2.31 for *CDS4n*, when performing vertex degree computations.

Despite the fact that *CDS4n* stores less information than *CDS5n*, for the computation of vertex degrees it achieves better performances in most cases (compare the performances reported in Table 3): this is mainly due to the better complexity of navigation between siblings in the *red* tree (requiring only one arithmetic operation), and the fact the retrieval of a vertex (`Source` operator) or of an edge (`Edge` operator) involves the computation of modulo 4 or multiplications by 4 (instead of modulo 5 computations for *CDS5n*).

Our representations are still competitive when considering the `target` operator, which requires $O(d)$ time in the worst case for a vertex of degree d for our compact data structures. The results reported in Figure 13 provide a comparison of the runtime performances for the vertex normal computation, and show that our representations are slower than standard data structures by a factor between 1.5 and 2.5 in average (all geometric calculations involve simple float precision). The higher cost of the `target` operator for the *CDS6n*, *CDS5n* and *CDS4n* representations is compensated by the cost of geometric calculations, which dominate the runtime performances, and is the same for all representations (observe that the `target` operator requires $O(1)$ time in the case of *Winged-edge*).

The results of Figure 13 put also in evidence an interesting phenomenon: when geometric calculations are involved, the preservation of vertex locality can play an important role. This explains the behaviour of the *CDS5n* data structure, which is sometimes faster than *CDS4n*. The main reason is that the original vertex ordering is preserved in *CDS5n*, thus leading to a faster access to geometric data, which are stored according to the vertex ordering in the input mesh.

4.3 Limitations

Our data structures suffer of the same limitations as most of existing compact representations.

Static vs. dynamic sata structures In our work we deal with static triangle meshes, where local updates are not supported. Our representations rely on the structural properties of Schnyder woods, which are known to be unstable under local modifications: even a single update, such an edge flip, can result in a global perturbation of the Schnyder wood, that needs to be recomputed from scratch.

Preprocessing time The main bottleneck of many compact data structures [41, 12, 13, 29, 26, 27, 28] concerns the memory ressources required in the preprocessing phase. In our case the linear-time construction (from the input OFF file) is very fast in practice but requires to keep in main memory an explicit representation of the input mesh together with some additional informations needed for the computation of the Schnyder wood (see the Appendix for a more detailed discussion about the runtime and memory costs of the computation of a Schnyder wood). Thus the initial memory cost can go well beyond the size of our compact data structures, which could result to be rather expensive for processing very huge meshes.

It should be noted that this issue does not concern the decoding of our data structures (*CDS6n* and *CDS5n*) from the compressed format: as confirmed by our experiments, decoding the input compressed format described in Section 3 is extremely fast and does not require any additional memory (the memory usage remains thus limited even for very large meshes).

4.4 Storage bounds: theoretical guarantees vs. more concise heuristics

As mentioned in Section 1.1, some interesting heuristics leads to very compact and efficient representations [27, 26]. We recall that their storage performances hold for the tested meshes: according to the statistics reported in [27] (see Table 1), the performances are highly correlated to the regularity of the input mesh. For instance, *LR* achieves the best performance on the *welsh dragon* model (2.04 references per vertex) that has a large majority of degree 6 vertices (86.7%), while the worst performance of LR concerns the *buddha* model (3.16 references per vertex) whose connectivity is less regular (only 32.1% of vertices have degree 6). A similar behaviour holds for the *SQuad* data structure, that requires 4.05 references per vertex for the *welsh dragon* and consumes 4.3 references per vertex for the *buddha* model: for these two meshes *SQuad* achieves the best and worst storage performances (as *LR* does).

This would suggest that their storage performances could increase for meshes whose connectivity is very irregular (small proportion of degree 6 vertices).

This could occur in the case of random planar triangulations [6] of size n, where the distribution of vertex degrees follows an exponential decay: as evaluated in [34, 23], the probability of having a vertex of degree i in a large random planar triangulation, as n goes to infinity, is $p_i = \frac{16(i-2)}{i}\binom{2i-2}{i-1}(\frac{3}{16})^i$. For instance, as confirmed in our experiments, the proportion of degree 6 vertices in a large random triangulation is approximately 11.6%.

[6] By random planar triangulation we mean here a simple triangulated planar map randomly chosen according to uniform distribution among all triangulations of size n.

5 Conclusion

We have designed and implemented a new family of compact data structures for planar triangulations that achieve good trade-offs between space requirements and runtime performances. Common navigational operators are supported by our representations in constant time, and the space bounds are all provided with theoretical guarantees in the worst case. Moreover, we presented a decoding procedure allowing retrieving a compact data structure from a standard compressed format in linear time, without using any additional storage.

5.1 Polygonal meshes and higher genus surfaces

Our approach is quite general and could be adapted to extend all the features above to other important classes of meshes (such as polygonal or quadrangular meshes) and to deal with higher genus surfaces or surfaces with boundaries.

Our algorithm can be extended to surfaces with holes and boundaries can be handled by topologically triangulated each boundary around a vertex that is marked as outside vertex. Each boundary edge in the initial surface becomes a triangle having an *outside vertex* in the new surface, thus the storage cost for *CDS5n* becomes $5(n+m)$ where n and m are the number of vertices and holes of the original surface.

For dealing with the higher genus case, the key ingredients are two recent generalizations of Schnyder wood orientations for toroidal [22] and genus g triangulations [17]. More precisely, as described in [17] for a genus g (rooted) triangulation it is possible to define a coloration/orientation of inner edges such that: all the inner edges (not lying on the root face) can be oriented in one direction (having one color red, blue, or black), at the exception of a small set of *special* edges, which have possibly two orientations and two colors. An important feature is that there are at most $2g$ special edges; and all inner vertices (not lying on the root face) have outgoing degree 3 as in the plane, at the exception of at most $4g$ *multiple vertices*, which may have outgoing degree at most $O(g)$ (multiple vertices are the extremities of special edges).

While it is quite easy to adapt the arguments of Thm 2 to obtain a storage of $6n + O(g)$ references, the adaptation of Theorems 4 and 5 is not straightforward, since in the genus g case there is no characterization of *ccw* oriented triangles. As detailed in [11], it is possible to design a compact representation requiring at most $5(n + 4g)$ references, combining the BFS argument used in Thm 5 with genus g Schnyder woods defined in [17].

Our work could be also extended to deal with more general meshes (not triangulated) such as quad and polygonal meshes, since some nice (maximal) edge orientations (with bounded outgoing degree) have also been defined for planar quadrangulations and 3-connected graphs [21, 19].

5.2 Open problems

There are still a few questions which remain open. It would be interesting to design a decoding procedure which allows reconstructing our most compact data structure (CDS4n, using $4n$ references) from compressed format. The main issue relies on the vertex orderings which are different: in the CDS4n representation vertices are numbered according to a BFS traversal of the red tree, while the vertex numbers in the encoding correspond to a DFS traversal.

It would be interesting to see whether our $4n$ bound could be improved, while still achieving the same navigation performances. While in the general triangular case we do not know how to reduce the number of references, a possible improvement could be obtained in the special (but important) case of irreducible triangulations (with no separating triangles), for which Schnyder woods and related orientations do exhibit additional properties.

Finally, the problem of providing rigorous upper bounds for the storage requirements of the heuristics mentioned in Section 4.4 could have practical and theoretical interest. A challenging task would be to provide worst case upper bounds, or even to express the storage complexity in terms of structural properties (e.g. vertex degree distribution).

References

[1] Pierre Alliez and Craig Gotsman. Recent advances in compression of 3d meshes. In *Advances in Multiresolution for Geometric Modelling*, pages 3–26. Springer, 2005. URL: `http://dx.doi.org/10.1007/3-540-26808-1_1`.

[2] Tyler J Alumbaugh and Xiangmin Jiao. Compact array-based mesh data structures. In *Proceedings of the 14th International Meshing Roundtable*, pages 485–503. Springer, 2005. `doi:10.1007/3-540-29090-7_29`.

[3] Bruce G Baumgart. Winged edge polyhedron representation. Technical report, DTIC Document, 1972. URL: `http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=AD0755141`.

[4] Bruce G Baumgart. A polyhedron representation for computer vision. In *Proceedings National Computer Conference and Exposition*, pages 589–596. ACM, 1975. `doi:10.1145/1499949.1500071`.

[5] David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. `doi:10.1007/s00453-004-1146-6`.

[6] Olivier Bernardi and Nicolas Bonichon. Catalan's intervals and realizers of triangulations. *Journal of Combinatorial Theory, Series A*, 116(1):55–75, 2009. 22 pages. URL: `https://hal.archives-ouvertes.fr/hal-00143870`.

[7] Daniel Blanford, Guy Blelloch, and Ian Kash. Compact representations of separable graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 679–688, 2003. URL: `https://www.cs.cmu.edu/~guyb/papers/BF10.pdf`.

[8] Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. *Computational Geometry: Theory & Applications*, 22:5–19, 2002. URL: `https://hal.inria.fr/inria-00167199`, `doi:10.1016/S0925-7721(01)00054-2`.

[9] Enno Brehm. 3-orientations and Schnyder 3-tree-decompositions. *Master's Thesis, FB Mathematik und Informatik, Freie Universität Berlin*, 2000.

[10] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed edges—a scalable representation for triangle meshes. *Journal of Graphics Tools*, 3:1–11, 1998. `doi:10.1080/10867651.1998.10487494`.

[11] Luca Castelli Aleardi and Olivier Devillers. Explicit array-based compact data structures for triangulations. Research Report 7736, INRIA, 2011. URL: `http://hal.inria.fr/inria-00623762/`.

[12] Luca Castelli Aleardi, Olivier Devillers, and Abdelkrim Mebarki. Catalog based representation of 2d triangulations. *International Journal of Computational Geometry & Applications*, 21:393–402, 2011. URL: http://hal.inria.fr/inria-00560400, doi:10.1142/S021819591100372X.

[13] Luca Castelli Aleardi, Olivier Devillers, and Jarek Rossignac. ESQ: editable squad representation for triangle meshes. In *25th Conference on Graphics, Patterns and Images, SIBGRAPI 2012*, pages 110–117, 2012. URL: http://dx.doi.org/10.1109/SIBGRAPI.2012.24.

[14] Luca Castelli Aleardi, Olivier Devillers, and Jarek Rossignac. Triangulation data structures. In *Encyclopedia of Algorithms*, pages 2262–2267. Springer, 2016. URL: http://dx.doi.org/10.1007/978-1-4939-2864-4_589.

[15] Luca Castelli Aleardi, Olivier Devillers, and Gilles Schaeffer. Succinct representation of triangulations with a boundary. In *Proc. 9th Workshop on Algorithms and Data Structures*, volume 3608 of *Lecture Notes in Computer Science*, pages 134–135. Springer-Verlag, 2005. URL: http://hal.inria.fr/inria-00090707.

[16] Luca Castelli Aleardi, Olivier Devillers, and Gilles Schaeffer. Succinct representations of planar maps. *Theoretical Computer Science*, 408:174–187, 2008. URL: http://hal.inria.fr/inria-00337821/, doi:10.1016/j.tcs.2008.08.016.

[17] Luca Castelli Aleardi, Éric Fusy, and Thomas Lewiner. Schnyder woods for higher genus triangulated surfaces, with applications to encoding. *Discrete & Computational Geometry*, 42(3):489–516, 2009. URL: https://hal.inria.fr/hal-00712046v1, doi:10.1007/s00454-009-9169-z.

[18] Richie Chih-Nan Chuang, Ashim Garg, Xin He, Ming-Yang Kao, and Hsueh-I Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. In *Automata, Languages and Programming*, pages 118–129. Springer, 1998. doi:10.1007/BFb0055046.

[19] Hubert De Fraysseix and Patrice Ossona de Mendez. On topological aspects of orientations. *Discrete Mathematics*, 229(1):57–72, 2001. doi:10.1016/S0012-365X(00)00201-6.

[20] Vincent Despré, Daniel Gonçalves, and Benjamin Lévêque. Encoding toroidal triangulations. *Discrete & Computational Geometry*, 57(3):507–544, 2017. URL: https://doi.org/10.1007/s00454-016-9832-0.

[21] Stefan Felsner. Convex drawings of planar graphs and the order dimension of 3-polytopes. *Order*, 18(1):19–37, 2001. doi:10.1023/A:1010604726900.

[22] Daniel Gonçalves and Benjamin Lévêque. Toroidal maps: Schnyder woods, orthogonal surfaces and straight-line representations. *Discrete & Computational Geometry*, 51(1):67–131, 2014. URL: http://dx.doi.org/10.1007/s00454-013-9552-7.

[23] Craig Gotsman. On the optimality of valence-based connectivity coding. *Comput. Graph. Forum*, 22(1):99–102, 2003. URL: https://doi.org/10.1111/1467-8659.t01-1-00649.

[24] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM transactions on graphics (TOG)*, 4(2):74–123, 1985. doi:10.1145/282918.282923.

[25] Stefan Gumhold and Wolfgang Straßer. Real time compression of triangle mesh connectivity. In *Proc. of SIGGRAPH 1998*, pages 133–140, 1998. URL: http://doi.acm.org/10.1145/280814.280836.

[26] Topraj Gurung, Daniel Laney, Peter Lindstrom, and Jarek Rossignac. SQuad: Compact representation for triangle meshes. *Computer Graphics Forum*, 30(2):355–364, 2011. doi:10.1111/j.1467-8659.2011.01866.x.

[27] Topraj Gurung, Mark Luffel, Peter Lindstrom, and Jarek Rossignac. LR: compact connectivity representation for triangle meshes. *ACM transactions on graphics (TOG)*, 30(4), 2011. doi:10.1145/2010324.1964962.

[28] Topraj Gurung, Mark Luffel, Peter Lindstrom, and Jarek Rossignac. Zipper: A compact connectivity data structure for triangle meshes. *Computer-Aided Design*, 45(2):262–269, 2013. URL: http://dx.doi.org/10.1016/j.cad.2012.10.009.

[29] Topraj Gurung and Jarek Rossignac. SOT: compact representation for tetrahedral meshes. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pages 79–88. ACM, 2009. doi:10.1145/1629255.1629266.

[30] Xin He, Ming-Yang Kao, and Hsueh-I Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *SIAM J. Discrete Math.*, 12(3):317–325, 1999. URL: http://dx.doi.org/10.1137/S0895480197325031.

[31] Marcelo Kallmann and Daniel Thalmann. Star-vertices: a compact representation for planar meshes with adjacency information. *Journal of Graphics Tools*, 6(1):7–18, 2001. doi:10.1080/10867651.2001.10487533.

[32] Lutz Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom.*, 13(1):65–90, 1999. URL: https://doi.org/10.1016/S0925-7721(99)00007-3.

[33] Stephen G. Kobourov. Canonical orders and schnyder realizers. In *Encyclopedia of Algorithms*, pages 277–283. Springer, 2016. URL: http://dblp.uni-trier.de/rec/bib/reference/algo/Kobourov16.

[34] Valery A. Liskovets. A pattern of asymptotic vertex valency distributions in planar maps. *J. Comb. Theory, Ser. B*, 75(1):116–133, 1999. URL: https://doi.org/10.1006/jctb.1998.1870.

[35] Mark Luffel, Topraj Gurung, Peter Lindstrom, and Jarek Rossignac. Grouper: A compact, streamable triangle mesh data structure. *IEEE Trans. Vis. Comput. Graph.*, 20(1):84–98, 2014. URL: http://dx.doi.org/10.1109/TVCG.2013.81.

[36] Adrien Maglo, Guillaume Lavoué, Florent Dupont, and Céline Hudelot. 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Comput. Surv.*, 47(3):7:1–7:41, 2015. URL: http://dx.doi.org/10.1145/2693443.

[37] J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

[38] Dominique Poulalhon and Gilles Schaeffer. Optimal coding and sampling of triangulations. *Algorithmica*, 46(3-4):505–527, 2006. doi:10.1007/s00453-006-0114-8.

[39] Jarek Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *Visualization and Computer Graphics, IEEE Transactions on*, 5(1):47–61, 1999. doi:10.1109/2945.764870.

[40] Walter Schnyder. Embedding planar graphs on the grid. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, volume 90, pages 138–148, 1990. URL: http://departamento.us.es/dma1euita/PAIX/Referencias/schnyder.pdf.

[41] Jack Snoeyink and Bettina Speckmann. Tripod: a minimalist data structure for embedded triangulations. In *Workshop on Comput. Graph Theory and Combinatorics*, 1999. URL: https://www.win.tue.nl/~speckman/papers/Tripod.pdf.

[42] Costa Touma and Craig Gotsman. Triangle mesh compression. In *Proc. of the Graphics Interface 1998 Conference*, pages 26–34, 1998. doi:10.20380/GI1998.04.

[43] Katsuhisa Yamanaka and Shin-ichi Nakano. A compact encoding of plane triangulations with efficient query supports. *Information Processing Letters*, 110(18):803–809, 2010. doi:10.1016/j.ipl.2010.06.014.

A Appendix: Linear-time construction of a (maximal) Schnyder wood

For the sake of completeness we provide a concise description of the procedure that we use to obtain a Schnyder wood of a planar triangulation $\mathcal{G}$: we follow the approach based on vertex conquests described in [9, 33] (this has been generalized to higher genus surfaces in [17]).

The procedure is based on a *growing region* approach, where a region C (white faces in Fig. 14), delimited by a simple cycle B (orange edges in Fig. 14), is incrementally maintained under vertex conquests. Since B is simple, the faces of C and the ones of $\mathcal{G} \setminus C$ (gray region in Fig. 14) both define two regions which are homeomorphic to a topological disk (they are simply connected). We say that a vertex $v \setminus \{V_0, V_1\}$ on the cycle B is *free* if it has no incident chordal edges, that are edges whose two extremities belongs to B (dashed light gray segments).

At the beginning C coincides with the root (infinite) face, $\mathcal{G} \setminus C$ contains all the inner faces of $\mathcal{G}$, and the cycle B consists of the outer vertices (V_0, V_1, V_2): the only free vertex is V_2. The *vertex conquest* of a free vertex $v \in B$ consists in removing v from $\mathcal{G} \setminus C$, together with all its incident edges and faces in $\mathcal{G} \setminus C$: since B is assumed to be simple and v has no incident chordal edges, B is still simple after the conquest of v and the set of removed (gray) faces defines a triangle fan around v. During the vertex conquest we perform the following simple rule which assigns colors and orientations to edges (illustrated by the top-left picture in Fig. 14). Let us denote by v_l and v_r the left and right neighbors of v on B: we assign color blue to the edge (v_r, v) and color red to (v_l, v), both being oriented outgoing from vertex v. All remaining possibly existing edges incident to v in $\mathcal{G} \setminus C$ are colored in black and oriented toward v. The algorithm described above terminates after $n-2$ steps, leading to a triangulation where all edges have been endowed with a color and an orientation, except the edge $\{V_0, V_1\}$. At the end we discard the colors and orientations of edges $\{V_0, V_2\}$ and $\{V_1, V_2\}$ in order to fulfil the requirements of Definition 1 (see Fig. 14(j)).

This procedure terminates without getting stuck since we can always find, at each step of the algorithm, a free vertex on B. Its existence relies on an inductive argument illustrated in Fig. 14 (see top-right picture): because of planarity and the fact the B is a simple cycle, the set of chordal edges in $\mathcal{G} \setminus C$ defines an outerplanar graph (having B as boundary face). As a

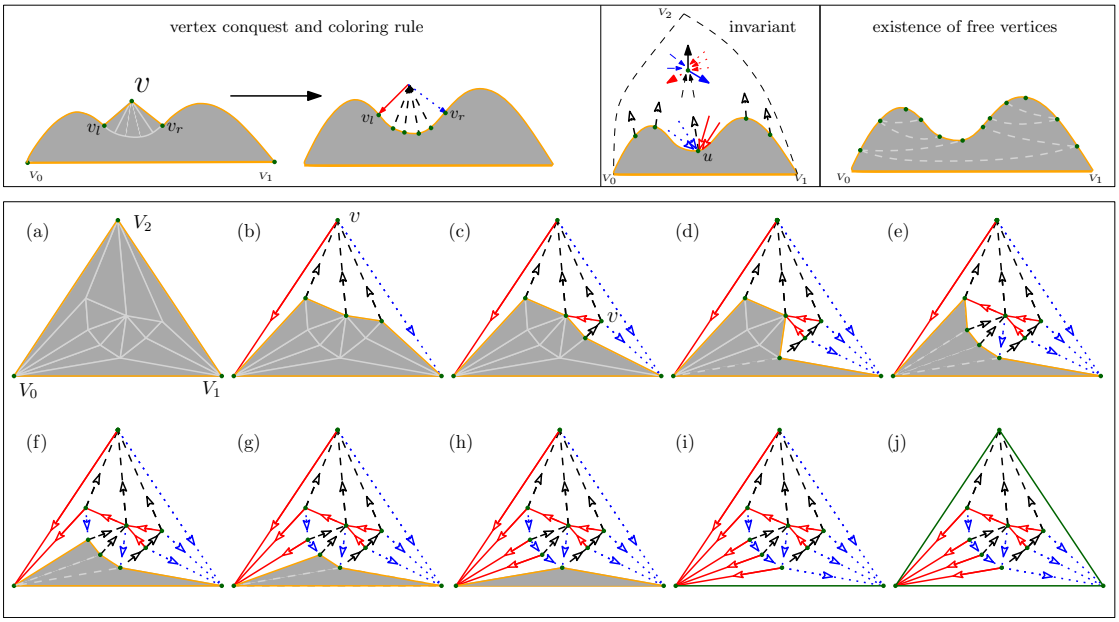


Figure 14: This figure illustrates the linear-time computation of a (maximal) Schnyder wood of a planar triangulation. The bottom pictures (a-j) illustrate the incremental vertex conquest of the triangulation. The algorithm consists in performing $n-2$ vertex conquests, while maintaining a simple boundary cycle B (orange edges) enclosing the region that remains to be visited (gray faces). At the beginning (a) the boundary cycle coincides with the outer cycle (V_0, V_1, V_2). In order to ensure that B remains always simple (and the gray region simply connected), we avoid to remove vertices on B that are incident to chordal edges (dashed light gray segments). The Schnyder wood orientation is obtained by applying a very simple coloring rule (top-left picture). To get the maximal Schnyder wood it suffices to perform, at each step, the conquest of the free vertex closest to V_1.

consequence, there is at least one vertex $v \in B$ without incident chordal edges (this vertex could be incident to a single triangle (v_l, v, v_r)).

The correctness of the algorithm, which computes an orientation of all inner edges satisfying Definition 1, relies on a few invariants involving the coloration and orientation of edges in the conquered region C (top-center picture in Fig. 14). For instance, it is straightforward to check that at the end each inner vertex gets exactly three outgoing edges (one for each color): just observe that each vertex $v \in B$, just before its conquest, has already a black edge in C (v gets such an outgoing black edge the first time it appears on B). Just after its conquest, each vertex v gets two outgoing edges whose colors are red and blue respectively. In the all remaining steps, not involving the conquest of vertex v, no other edges are oriented outgoing from v.

Maximal Schnyder wood. In general a rooted planar triangulation admits many Schnyder woods: just observe that at each steps there are many choices among the possible free vertices on B. The computation of the maximal Schnyder wood (without cycles of edges oriented in cw direction) can be performed as before, adopting the following simple rule. During the incremental vertex conquest described above just perform the conquest of the free vertex on B that is the closest to V_1. The correctness of this procedure is not totally trivial, and for a more detailed

presentation we refer to [9].

Computational complexity and memory cost

We briefly provide an evaluation of the timing and memory cost of the procedure above. As far as we know, this work provides the first experimental evaluation of the runtime performances of the computation of Schnyder woods and canonical orderings for large triangulations.

At each step the algorithm performs the removal (and coloring) of a triangle fan consisting of d_v triangles in $\mathcal{G} \setminus C$ around the conquered free vertex v: this takes at most $O(d_v)$ time, spent to update the boundary cycle B and assigning colors and orientations. This leads, amortizing over the $n-2$ steps, to linear time complexity. This bound is also confirmed by our experiments: as shown in Table 2 (third column), the computation of a Schnyder wood is very fast, as we can process in average more than 10M of faces per second. Concerning the memory consumption: to perform the vertex conquest procedure we needs a data structure for representing the planar triangulation, together with a doubly-linked list supporting constant time addition/removal of vertices on the the simple cycle B (a few further informations are used to mark free vertices and chordal edges). In the worst case the cycle B could be of size n, thus adding $3n$ more references to the memory cost of processing Schnyder woods (two references for implementing the doubly-linked list, and one reference to the stored vertex living on B). Our experiments show that B remains of size $O(\sqrt{n})$ during the vertex conquest, that makes its storage cost negligible in practice: the storage of an explicit representation of the triangulation, required for performing the computation of the Schnyder wood, is by far the dominant term of the memory requirements as well as of the runtime cost of our construction algorithm (see Table 2, second column).

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399