Efficient Edgebreaker for surfaces of arbitrary topology

THOMAS LEWINER[1,2], HÉLIO LOPES[1], JAREK ROSSIGNAC[3] AND ANTÔNIO WILSON VIEIRA[1,4]

[1] Department of Mathematics — Pontifícia Universidade Católica — Rio de Janeiro — Brazil
[2] Géométrica Project — INRIA – Sophia Antipolis — France
[3] GVU Center — GATECH — Atlanta — USA
[4] CCET — Universidade de Montes Claros — Brazil
{tomlew, lopes, awilson}@mat.puc--rio.br. jarek@cc.gatech.edu.

Abstract. The typical surfaces models handled by contemporary Computer Graphics applications have millions of triangles and numerous connected component, handles and boundaries. Edgebreaker and Spirale Reversi are examples of efficient schemes to compress and decompress their connectivity. A surprisingly simple linear–time implementation has been proposed for triangulated surfaces homeomorphic to a sphere and was subsequently extended to surfaces with handles. Here, we further extend its scope to surfaces with multiple components, handles, and multiple boundaries. The result is a simple and efficient compression/decompression solution for the broad class of orientable manifold surfaces.

Keywords: ***Edgebreaker. Spirale Reversi. Mesh Compression. Triangular Meshes.***

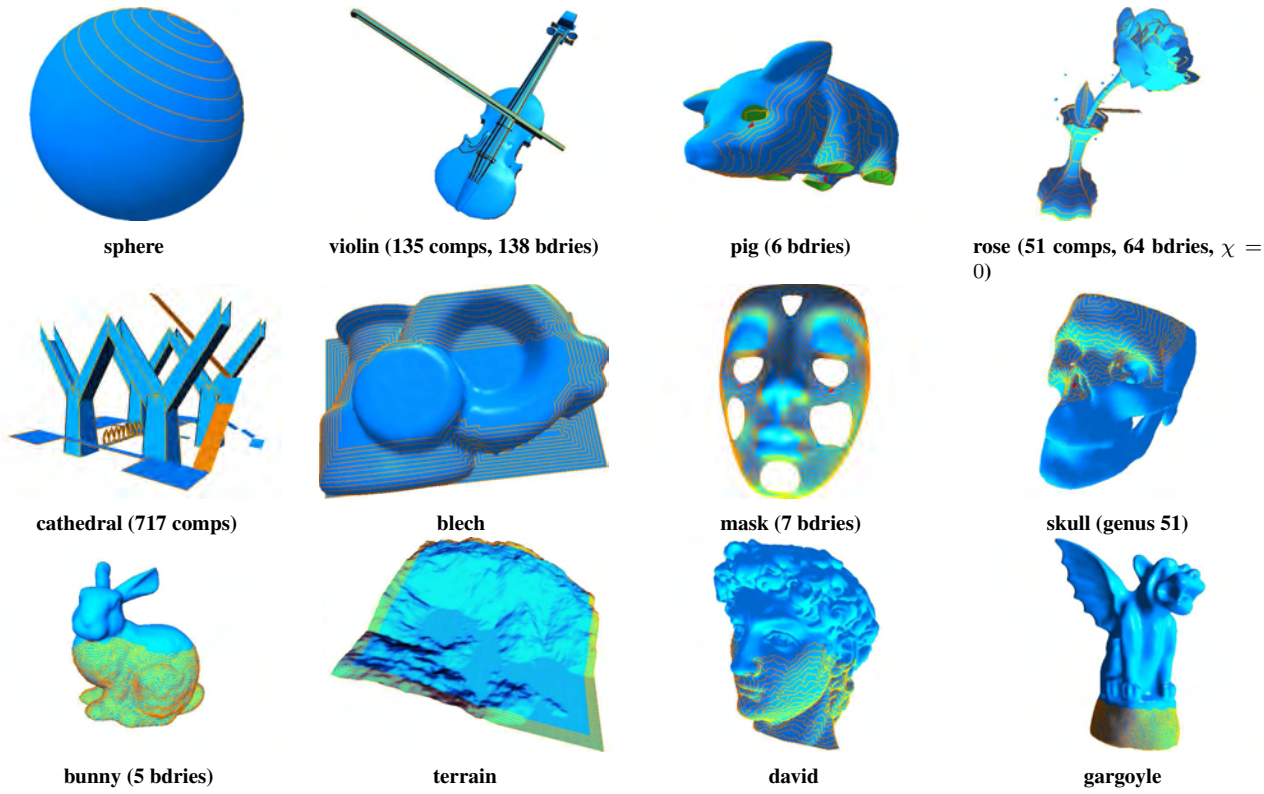


Figure 1: Some of the models used for the experiments, with the beginning of Edgebreaker's dual spanning tree.

Preprint MAT. 07/03, communicated on September 30^{th}, 2003 to the Department of Mathematics, Pontifícia Universidade Católica — Rio de Janeiro, Brazil. The corresponding work was published in the proceedings of the Sibgrapi 2004, pp. 218–225 IEEE Press, 2004.

1 Introduction

The Edgebreaker scheme [12] encodes the connectivity of any manifold triangle mesh homeomorphic to a sphere with a guaranteed worst case code of 1.83 bits per triangle [6]. The Spirale Reversi algorithm [5] enhanced the Edgebreaker decompression worst–case complexity from $O(n^2)$ to $O(n)$. But the true value of Edgebreaker and Spirale Reversi lies in the efficiency and in the simplicity of their implementations [14], which is very concise. They can be simply implemented on a reduced topological data structure (the Corner–Table) which only uses two arrays of integers to represent the connectivity of the mesh. This simple algorithm has been extended to deal with surfaces with handles in [8]. Because of its simplicity, Edgebreaker is viewed as the emerging standard for 3D compression [15] and may provide an alternative for the current MPEG–4 standard which is based on the Topological Surgery approach [16].

Prior Works. There are many different compression schemes for triangular meshes. In order to encode efficiently their geometry, the best known methods traverse the cells of the mesh, and differ in the way they encode this traversal. The Edgebreaker scheme has been enhanced and adapted from the Topological Surgery [16] to yield an efficient but initially restricted algorithm [12], and subsequently extended to more general meshes [5, 7]. Edgebreaker encodes the connectivity of the mesh by producing the *clers* string of symbols taken from the set C,L,E,R,S. A different approach encodes the connectivity of the mesh by the valence of its vertices [17, 1]. Valence–based compression approaches are very efficient, especially for regular meshes and it can also be extended to general polygonal meshes. Another approach [10] computes a uniquely defined traversal for a given mesh, leading to asymptotically optimal results for the worst case. However, it is restricted to meshes without boundary and without handle.

The Spirale Reversi algorithm [5] enhanced the Edgebreaker original decompression [12] and the Wrap&Zip decompression [13]. It reconstructs the connectivity encoded in the *clers* string in only one pass, and performing the same tests as the compressor. However, it needs to read it in a reverse way.

The Edgebreaker algorithm has been previously extended to efficiently support surfaces with handles in [8], introducing the *handle* stream to store in an efficient way two edges for each handle on the surface. Those edges together with the *clers* string are sufficient to recover all the surface connectivity. This extension doesn't add a new symbol to the Edgebreaker original *clers* set.

Surfaces with boundary are usually encoded by closing each boundary curve, using a dummy vertex to maintain the triangular structure [4, 12]. This is a very simple but expensive solution: first, it requires encoding each boundary edge with a useless triangle; second, it requires extra code to localize the dummy vertex; and third, it gives bad geometrical predictors on the boundary. The original Edgebreaker encodes boundary curves with an extra symbol **M**, and writing the length of each boundary curve. This allows a better geometrical prediction, but gives a complex implementation with an implicit representation of the topology, and it requires extra codes which harm the coding of the *clers* string. The scheme introduced here does not require more symbol than the original Edgebreaker for closed surfaces, and encode each boundary with only two integers.

Contributions. In this paper, we provide efficient and robust extensions of the Edgebreaker compression and of the associated Spirale Reversi decompression schemes for surfaces with an arbitrary topology. This new approach is based on a new semantics, which enables us to use the Edgebreaker 5–symbols *clers* string to encode the connectivity of an orientable surface, possibly having several connected components, handles or boundary curves. To do so, we exploit a topological analogy between handles and boundaries, and capitalize on the simplicity with which edges may be identified in the *topology* stream. Moreover, the resulting compression format represents separately the topology, the local connectivity and the geometry of the surface, leading to a simple and robust implementation.

Paper outline. Section 2 *Basic concepts* introduces some basic concepts. Section 3 *Corner–Table Data Structure* describes the Corner–Table data structure. Section 4 *Surface Duality and the Edgebreaker* establishes some notation and presents important properties that connect the surface duality to the Edgebreaker algorithm. Section 5 *Algorithm overview* presents the algorithm overview. Section 6 *Compression* and section 7 *Decompression* introduce, respectively, the enhanced Edgebreaker compression and the extended Spirale Reversi decompression algorithms. The theoretical analysis of the algorithm is presented in Section 8 *Theoretical Analysis*. Finally, Section 9 *Results* shows some results and comparison with former Edgebreaker algorithm.

2 Basic concepts

We will consider an orientable triangulated combinatorial surface. This is the general case of manifold triangle meshes embedded in $\mathbb{R}^3$, but we are only concerned with their connectivity, e.g., the triangle/vertex incidence and the triangle/-triangle adjacency information.

Definition 1 (Combinatorial surface) *A triangle mesh $\mathcal{S}$ is a combinatorial surface if:*

- *Every edge in $\mathcal{S}$ is bounding either one or two triangles.*
- *The link of a vertex in $\mathcal{S}$ is homeomorphic either to an interval or to a circle.*

The corresponding work was published in the proceedings of the Sibgrapi 2004, pp. 218–225 IEEE Press, 2004.

The set of edges in $\mathcal{S}$ incident to only one triangle is called the *boundary* of $\mathcal{S}$, denoted by $\partial(\mathcal{S})$. A *boundary curve* of a surface is a maximal connected set of adjacent edges of the boundary. Each boundary curve is closed. From now on, we denote by $\mathcal{T}(\mathcal{S})$, $\mathcal{E}(\mathcal{S})$ and $\mathcal{V}(\mathcal{S})$ the set of triangles, edges and vertices of $\mathcal{S}$.

Theorem 2 (Surface classification) *[2] Any compact oriented connected surface $\mathcal{S}$ is homeomorphic to a sphere ($g(\mathcal{S}) = 0$) or a connected sum of $g(\mathcal{S})$ tori ($g(\mathcal{S}) > 0$), in both cases with possibly some finite number $b(\mathcal{S}) \geq 0$ of open disks removed. The number $g(\mathcal{S})$ is called the* genus *of $\mathcal{S}$, and $b(\mathcal{S})$ its number of boundary curves. The Euler characteristic $\chi(\mathcal{S})$ of $\mathcal{S}$ is equal to $\chi(\mathcal{S}) = |\mathcal{T}(\mathcal{S})| - |\mathcal{E}(\mathcal{S})| + |\mathcal{V}(\mathcal{S})| = 2 - 2 \cdot g(\mathcal{S}) - b(\mathcal{S})$.*

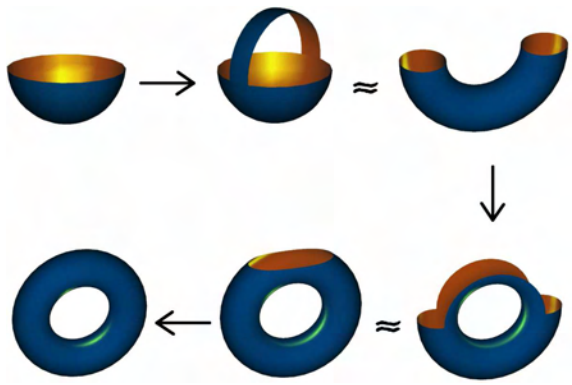


Figure 2: Handlebody decomposition of a torus [8].

Each torus of the connected sum above is composed by two 1–handles, as shown on Figure 2. This decomposition can be generalized using the Handlebody theory [8]. Since this concept of handle contains a complete representation of boundaries, it will play a fundamental role in our algorithm.

3 Corner–Table Data Structure

The Corner–Table is a very concise data structure for triangular meshes. It uses the concept of *corner* to represent the association of a triangle with one of its bounding vertices, or equivalently the association of a triangle with its bounding edge opposite to that corner: it may be viewed as a compact version of the half–edge representation of triangular meshes.

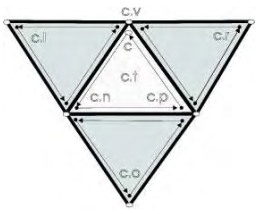
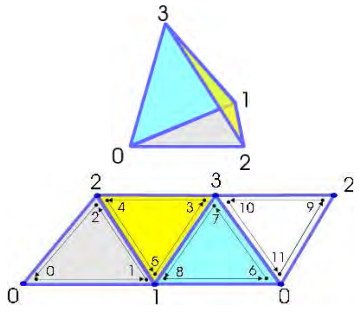


Figure 3: Corner notations.

In this data structure, the corners, the vertices and the triangles are indexed by non–negative integers. Each triangle is represented by 3 consecutive corners that define its orientation. For example, corners 0, 1 and 2 correspond to the first triangle; the corners 3, 4 and 5 correspond to the second triangle and so on... Consequently, a corner with index c is associated with the triangle of index $c.t = c \div 3$. Assuming a counter–clockwise orientation, for each corner c of a triangle $c.t$, the next ($c.n$) and previous ($c.p$) corners of $c.t$ are obtained by the use of the following expressions: $c.n = 3 \cdot c.t + (c+1) \bmod 3$, and $c.p = 3 \cdot c.t + (c+2) \bmod 3$.

The Corner–Table data structure represents the geometry of a surface $\mathcal{S}$ by the association of each corner c with its geometrical vertex index $c.v$. The edge–adjacency between the neighboring triangles of $\mathcal{S}$ is represented by associating with each corner c its opposite corner $c.o$, which has the same opposite geometrical edge (formally $c.n.v = c.o.p.v$ and $c.o.o = c$, see Figure 3). This information is stored in two integer arrays, called the V and O tables. For convenience, we define the left corner of c as $c.l = c.p.o$ and the right corner of c as $c.r = c.n.o$.



Corner	O table	V table
0	3	0
1	10	1
2	7	2
3	0	3
4	6	2
5	11	1
6	4	0
7	2	3
8	9	1
9	8	2
10	1	3
11	5	0

Figure 4: A tetrahedron with its corners and its Corner–Table.

To illustrate the data structure tables consider the tetrahedron of Figure 4. During the decompression, the corners will be enumerated in the order they were visited furing the compression.

4 Surface Duality and the Edgebreaker

In this section we provide some notations and introduce important properties that will be used to describe and analyze the algorithm.

The *primal graph* of a surface $\mathcal{S}$ is the simple graph whose nodes are the vertices $\mathcal{V}(\mathcal{S})$ and whose lines are the edges $\mathcal{E}(\mathcal{S})$ (e.g., a line connect adjacent vertices). The *dual graph* of a surface $\mathcal{S}$ is the graph whose nodes are the triangles $\mathcal{T}(\mathcal{S})$ and whose lines represent the edges $\mathcal{E}(\mathcal{S})$ (e.g., a line connect adjacent triangles). For example, Figure 5 represents the primal and the dual graph of a

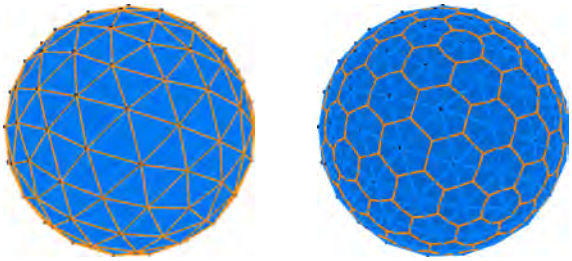


Figure 5: (left): the primal graph and (right): the dual graph of a triangulated sphere.

triangulated sphere.

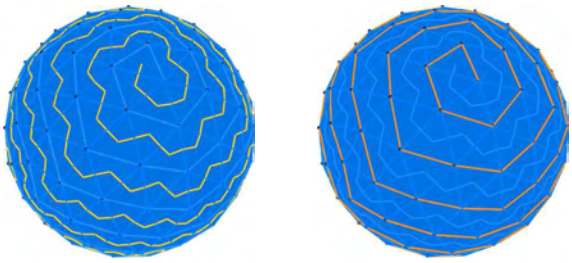


Figure 6: (left): a dual spanning tree $\Theta(\mathcal{S})$ extracted from the dual graph of Figure 5(rioght). (right): the primal remainder $\Gamma(\mathcal{S})$ of $\Theta(\mathcal{S})$, which is a subgraph of the primal graph of Figure 5(left).

Edgebreaker algorithms encode the dual and primal graphs of a triangular mesh $\mathcal{S}$ by an efficient use of surface duality. They extract a spanning tree $\Theta(\mathcal{S})$ of the dual graph of $\mathcal{S}$ (see Figure 6) by traversing and encoding the triangles of $\mathcal{S}$ in a spiral way. This encoding describes simultaneously the primal remainder (see Figure 6): The *primal remainder* is the maximal subgraph of the primal graph of $\mathcal{S}$ which does not intersect $\Theta(\mathcal{S})$, in other words:

Definition 3 (primal remainder) *Given a connected surface $\mathcal{S}$ and a spanning tree $\Theta(\mathcal{S})$ of the dual graph of $\mathcal{S}$, the* primal remainder $\Gamma(\mathcal{S})$ *is the simple graph whose nodes are the vertices $\mathcal{V}(\mathcal{S})$ and whose lines are the edges of $\mathcal{S}$ which are not in $\Theta(\mathcal{S})$.*

The following theorem relates the Euler characteristic of the surface to the connectivity of the primal remainder.

Theorem 4 *For any connected surface $\mathcal{S}$ and any dual spanning tree $\Theta(\mathcal{S})$, the primal remainder $\Gamma(\mathcal{S})$ is a connected graph with $|\mathcal{V}(\mathcal{S})|$ nodes and $|\mathcal{V}(\mathcal{S})| - \chi(\mathcal{S}) + 1$ lines. In particular, $\Gamma(\mathcal{S})$ is a tree if and only if $\mathcal{S}$ is homeomorphic to a sphere.*

Proof: Clearly $\Gamma(\mathcal{S})$ is connected, although the formal proof can involve basic topological techniques as thickening [2] or cell collapse (the fundamental group is unchanged by removing a dual spanning tree [3]). By definition, $\Theta(\mathcal{S})$ is a tree with $|\mathcal{T}(\mathcal{S})|$ nodes, hence, it has $|\mathcal{T}(\mathcal{S})| - 1$ lines. Also by definition, $\Gamma(\mathcal{S})$ has $|\mathcal{V}(\mathcal{S})|$ nodes, and the number of lines of $\Theta(\mathcal{S})$ and $\Gamma(\mathcal{S})$ together is $|\mathcal{E}(\mathcal{S})|$. Hence, the number of lines of $\Gamma(\mathcal{S})$ is $|\mathcal{E}(\mathcal{S})| - (|\mathcal{T}(\mathcal{S})| - 1) = |\mathcal{V}(\mathcal{S})| - \chi(\mathcal{S}) + 1$. Then, $\Gamma(\mathcal{S})$ is a tree if and only if its number of nodes equals its number of lines plus one: $|\mathcal{V}(\mathcal{S})| = (|\mathcal{V}(\mathcal{S})| - \chi(\mathcal{S}) + 1) + 1$. That is, iff the Euler characteristic of $\mathcal{S}$ is 2, e.g., iff $\mathcal{S}$ is a sphere (see Theorem 2). ■

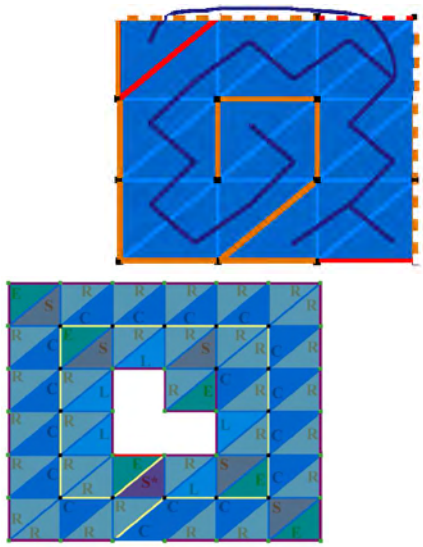


Figure 7: (left): a primal remainder on a torus (genus 1): the topmost and bottommost horizontal edges are identified, and so do the leftmost and rightmost ones. (right) a primal remainder on an annulus (two boundary curves).

For example, in the case of a sphere, the primal remainder is a tree (see Figure 6). For a mesh with genus one or two boundaries, the primal remainder is a graph with two cycles (see Figure 7). We can deduce the following theorem:

Theorem 5 *For every spanning tree $\Psi(\mathcal{S})$ extracted from $\Gamma(\mathcal{S})$, the number of edges that are in $\Gamma(\mathcal{S})$ but not in $\Psi(\mathcal{S})$ will be $2 \cdot g(\mathcal{S}) + b(\mathcal{S})$.*

Proof: We know that $\Gamma(\mathcal{S})$ has $|\mathcal{V}(\mathcal{S})|$ nodes. $\Psi(\mathcal{S})$ is a spanning tree of $\Gamma(\mathcal{S})$, then it also has $|\mathcal{V}(\mathcal{S})|$ nodes and $|\mathcal{V}(\mathcal{S})| - 1$ lines. Therefore, the number of lines of $\Gamma(\mathcal{S})$ that are not in $\Psi(\mathcal{S})$ is $(|\mathcal{V}(\mathcal{S})| - \chi(\mathcal{S}) + 1) - (|\mathcal{V}(\mathcal{S})| - 1) = 2 - \chi(\mathcal{S}) = 2 \cdot g(\mathcal{S}) + b(\mathcal{S})$. ■

5 Algorithm overview

Original Edgebreaker The Edgebreaker algorithm traverses spirally the dual graph of a surface in order to generate a spanning tree. At each step, a decision is made to move from a triangle Y to an adjacent triangle X. To perform this decision, all visited triangles and their incident vertices are marked. Let Left and Right denote the other

The corresponding work was published in the proceedings of the Sibgrapi 2004, pp. 218–225 IEEE Press, 2004.

two triangles that are incident upon X. Let v be the vertex common to X, Left, and Right. The edge opposed to v is called the *gate*. Five situations are distinguished according to Figure 8. Those cases are denoted by the letters **C**, **L**, **E**, **R** and **S**. The arrow indicates the direction to the next triangle. Previously visited triangles are filled in gray.

	v	Left	Right
C	not visited	not visited	not visited
L	visited	visited	not visited
R	visited	not visited	visited
E	visited	visited	visited
S	visited	not visited	not visited

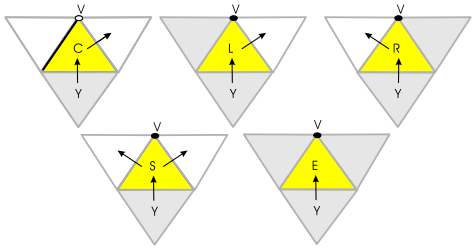


Figure 8: The Edgebreaker encoding.

To have a concise implementation of the Edgebreaker, the compression is done by the use of a recursive procedure that traverses the surface. The recursion starts only at triangles that are of type **S** and compresses the branch adjacent to the right edge of such a triangle. When the corresponding **E** triangle is reached, the branch traversal is complete and the routine returns from the recursion to pursue the left branch.

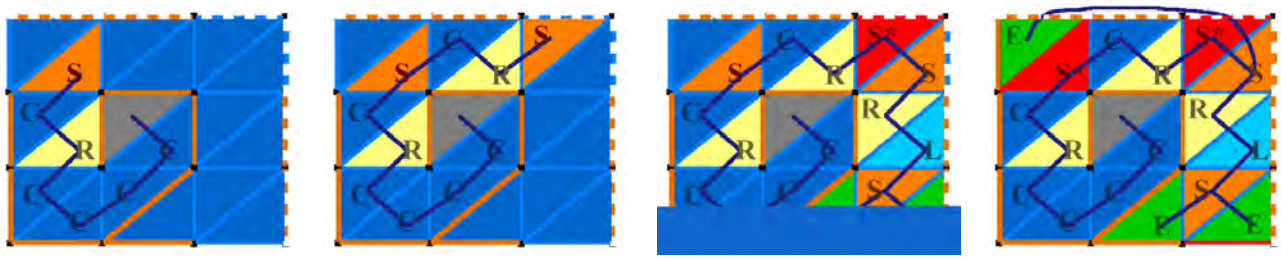
The original Edgebreaker does not handle surfaces with genus, and gives two options to compress the boundary curves. The first one [6] consists in closing each boundary curve by adding a dummy vertex, and joining it to the boundary vertices to form dummy triangles. The mesh does not have anymore boundary, and can be compressed using the algorithm described above. This is a very simple but expensive solution: first, it requires encoding each boundary edge a useless triangle; second, it requires extra code to localize the dummy vertex; and third, it gives bad geometrical predictors on the boundary. The second option [12] encodes the first triangle that reaches the boundary with an extra symbol **M** and sends the length of the boundary curve. At this step of the compression, encoding and removing this **M** triangle joins the boundary curve reached with the external boundary curve, and the resulting mesh has one less boundary curve.

Handles When the surface $\mathcal{S}$ has genus $g(\mathcal{S}) > 0$, the primal remainder $\Gamma(\mathcal{S})$ is not a tree anymore (see Theorem 4). For a surface without boundary, $\Gamma(\mathcal{S})$ has $|\mathcal{V}(\mathcal{S})| - \chi(\mathcal{S}) + 1 = |\mathcal{V}(\mathcal{S})| - 1 + 2 \cdot g(\mathcal{S})$ lines: there are $2 \cdot g(\mathcal{S})$ lines in excess [8]. These edges have been simply detected and efficiently encoded in [8], preserving the original Edgebreaker compression scheme. When the Edgebreaker traversing procedure finds an **S** triangle, the recursion starts and two situations are now distinguished. If the Left triangle has not been visited during the right branch traversal (case of normal **S**), we move to the left neighbor and continue our encoding of the left branch. Otherwise (case of a handle **S**) the pair of opposite corners separated by the left edge of the **S** triangle are sent to a stream or a file called *topology* and the routine returns. These corners will be matched during decompression to reconstruct the handle. The encounter of an **E** that does not match an **S** again terminates the compression process of the connected component.

Boundaries The work presented here extends these prior results to surfaces with handles, multiple boundary curves and multiple components. The former techniques for encoding boundaries and holes introduced in [12] are more costly, since they are using extra symbols or encoding more elements. In particular, they do not guarantee anymore the worst–case 1.83 bit per symbol. To improve the conciseness of our codification, we distinguish two distinct cases: a connected component with one boundary curve, and with more than one boundary curve.

Consider first a surface component $\mathcal{S}$ with genus $g(\mathcal{S})$ and only one boundary curve. We will close this component by adding a face incident to each boundary edge of $\mathcal{S}$, called the *infinite face*. The resulted surface $\mathcal{S}^+$ has no boundary, and the same genus, i.e., $g(\mathcal{S}^+) = g(\mathcal{S})$. We could almost use the extended Edgebreaker algorithm of [8] to encode $\mathcal{S}^+$. However, the infinity face is not a triangle. In the same way that the first triangle of the Edgebreaker classical algorithm is not encoded, we will not encode the infinity face, and start the compression from this one (see Figure 10(a)). As in the original Edgebreaker algorithm, we encode first all its vertices, e.g., all the vertices belonging to the boundary of $\mathcal{S}$. Therefore, we only need to know if the surface component has a boundary or not.

Now, consider a surface component with more than one boundary curve. We distinguish arbitrarily one of them as the *external* boundary, and call the others *internal* boundary curves (holes) (see Figure 10(b)). The external boundary curve is encoded as above. All the vertices of the *internal* boundary are marked as visited. Consequently, the first triangle to reach a boundary curve will always be an **S** triangle and we encode the opposite corners of its left edge in the *topology* stream. According to Theorem 5, the compression will store $2 \cdot g(\mathcal{S}) + b(\mathcal{S}) - 1$ edges in this *topology* stream.



Reaching first S triangle | **Reaching second S triangle** | **The lower–right E triangle closes the handle.** | **The upper–left E triangle closes the handle.**

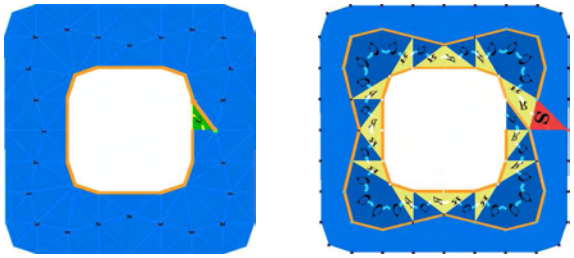
Figure 9: Coding of a torus: the creation of two *handle* S triangles: the first and the second S symbols.

6 Compression

The compression processes successively each surface component. When the component has a boundary, the compression encodes explicitly the first triangle, and the boundary containing an edge of this first triangle will be chosen as the external one. The vertices of this external boundary are encoded first during the component compression.

The compression of a component follows a dual spanning tree $\Theta(\mathcal{S})$. A stack stores the **S** triangles, e.g. the branchings of $\Theta(\mathcal{S})$, above the node being processed. After an **S** triangle has been pushed, the algorithm compresses first the branch adjacent to the right edge of the **S** triangle, until the corresponding **E** triangle is reached. At this point, two situations are distinguished. If the Left triangle has not been visited during the right branch traversal (case *normal* **S**), we move to the left neighbor (popping the stack) and continue our encoding of the left branch until we reach another **E**. Otherwise, the triangle will be called a *handle* **S** or *boundary* **S** depending whether the triangle touches an unvisited boundary or not. The left edge of a *handle* or *boundary* **S** is encoded in the *topology* stream, and the stack is popped. When the stack is empty, the connected component has been entirely compressed.

To illustrate the algorithm, consider firstly the surface given by the model for a triangulated torus as shown on Figure 9. Identifying the edges on the opposite sides of the rectangle, one can build a simplicial complex in $\mathbb{R}^3$ whose polyhedron is homeomorphic to the torus. Figure 9 illustrate the labels of all triangles defined by the Edgebreaker compression algorithm. At the end of the algorithm, the *clers* string obtained for the torus surface is **CCCCRCSCRSSRLSEEE**. As one can observe, in this example, there are four triangles labeled with **S**. In the string sequence, the last two **S** are normal, since their right and left branches are traversed in the compression algorithm. On the other hand, the left branches of the first and of the second **S** triangles are not traversed since their left adjacent triangle have been visited during their right branch traversal. Therefore, the two opposite corners of each left edge drawn in red are encoded separately in the *topology* stream.



The first triangle is chosen adjacent to a boundary. The vertices of the central infinite face are encoded. | **An unvisited boundary is reached: the corresponding S triangle is a *boundary* S triangle.**

Figure 10: Coding of an annulus: initialization and creation of *boundary* S triangles.

On Figure 10, the only *handle* **S** triangle is the first triangle with a vertex on the internal boundary that we encounter during the traversal. As said before, there are $2 \cdot g(\mathcal{S}) + b(\mathcal{S}) - 1$ such *handle* **S** triangles for each surface component with genus $g(\mathcal{S})$ and $b(\mathcal{S})$ boundary curves.

7 Decompression

The decompression procedure proposed here is an extension of the Spirale Reversi algorithm [5], which decodes the *clers* string in a one–pass reverse order, allowing the decompression to perform exactly the same tests as the compression. First, we read the *topology* stream, extracting the number of vertices, triangles and components with boundary. Then, we assign opposite vertices over the handles read from the *topology* stream. Then, we parse the *clers* string for reverse reading and for the component detection. We actually process backwards the *clers* string, and hence parse it only once, but we will expose two passes for the sake of clarity. The symbols **S** and **E** acts as parentheses, and each new component opens a parentheses. When all the opened parentheses are closed (at a symbol **E**), a new component begins on the next symbol. When the number of components exceeds the number of components with boundary read in the *topology* stream, we add an extra **P** symbol at the beginning of the component, because the first triangle of a

component without boundary is not explicitly encoded. The *handle* and *boundary* **S** triangles are distinguished from the *normal* **S** ones by their left corner, which has been assigned while reading the *topology* stream. We do not consider handle **S** symbols as open parentheses, as they do not match an **E** symbol. We notice here the power of Edgebreaker that explicitly labels each corner of the mesh by the position of its symbol in the *clers* string.

After this preprocessing step, we read the *clers* string backwards, as Spirale Reversi does. For each symbol, we decode the adjacency of the corresponding gate. When a *boundary* **S** symbol is read when a new component with boundary is processed, the geometry corresponding boundary curve is read (backwards) from the *vertex* stream. When a **C** symbol is read, we close the star of the active corner v and assign the corresponding corners to a new decoded vertex. At the end of this procedure, the connectivity and the geometry of the mesh is entirely restored in linear time.

8 Theoretical Analysis

From the graph theory point of view, the connectivity of a surface $\mathcal{S}$ is completely described by the dual tree $\Theta(\mathcal{S})$ and the primal remainder $\Gamma(\mathcal{S})$, and the way they are interlaced. We will justify here why the algorithm presented here can reconstruct those graphs, and consequently, the surface connectivity.

The strategy of the original Edgebreaker [12] builds simultaneously a spanning tree $\Theta(\mathcal{S})$ on the dual graph and implicitly encodes a spanning tree $\Psi(\mathcal{S})$ on the primal remainder $\Gamma(\mathcal{S})$. Branches on $\Theta(\mathcal{S})$ are created with symbol **S**, and ended by symbol **E** that corresponds to a leaf in $\Theta(\mathcal{S})$. The other symbols (**C, L** and **R**) create the internal nodes of the dual tree $\Theta(\mathcal{S})$.

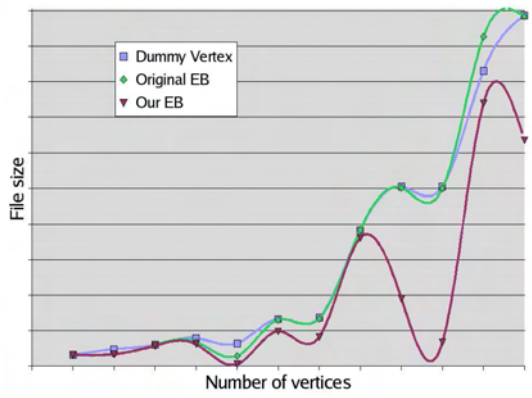
Each symbol **C** also creates the nodes of the tree $\Psi(\mathcal{S})$. The left edges of all **C** triangles are lines of $\Psi(\mathcal{S})$. If the surface component being encoded has no boundary, two edges of the starting triangle also belong to $\Psi(\mathcal{S})$ [8]. The lines of the primal remainder $\Gamma(\mathcal{S})$ that are not on $\Psi(\mathcal{S})$ are stored in the *topology* stream.

Concluding, the *clers* stream encodes explicitly $\Theta(\mathcal{S})$ and implicitly $\Psi(\mathcal{S})$. And by the use of the *topology* stream we explicitly encoded the edges that are missing to reconstruct $\Gamma(\mathcal{S})$. As those graphs are encoded simultaneously, the way $\Theta(\mathcal{S})$ and $\Gamma(\mathcal{S})$ are interlaced is obvious and the connectivity of $\mathcal{S}$ can be reconstructed.

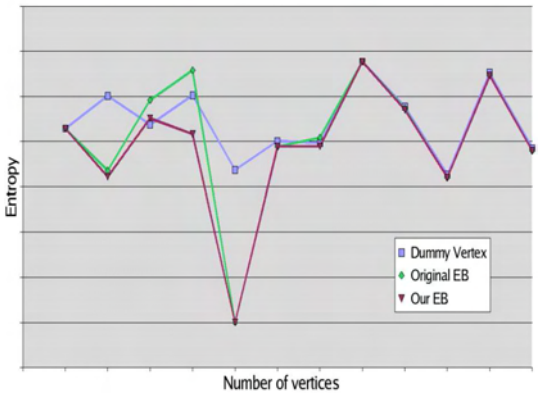
9 Results

We encoded the Edgebreaker symbols using a range encoder [9, 11], which is a one–pass approximation of the entropy coder. This gives very good results for big meshes (on the contrary of the 'sphere' model of Table 1, or meshes with high auto–similarity (like the model 'cathedral' of Table 1).

Our experimental results are recorded on Table 1 and Figure 11. We compared with the original Edgebreaker implementation with the Huffman encoding of [6], and our encoding with a simple arithmetic coder. Our experimental results are always better than the original Edgebreaker, mainly due to the range encoder. However, the entropy of our codes is always better than the other implementations of Edgebreaker (see Figure 11(b)).



Size of the compressed file vs complexity of the model.



Entropy vs complexity of the model.

Figure 11: Comparison of the final size and entropy: for the range encoder, those parameters depends more on the regularity than on the size of the model, but our algorithm really enhances the previous results.

10 Conclusion

We introduced here a simple, efficient and robust algorithm to code and decode the connectivity of an orientable manifold surface. The compression scheme is based on Edgebreaker, although we use only the 5 original symbols to encode topological features, maintaining explicit labeling of vertices and the ability to use a geometric predictive encoding. The decompression scheme is an extension of Spirale Reversi, which ensures a linear complexity and a one–pass decompression complexity. This guarantees less

Model	$\lvert\mathcal{V}(\mathcal{S})\rvert$	$\lvert\mathcal{T}(\mathcal{S})\rvert$	Dum	Ori	Ours	Ori/Ours	Dum/Ours
sphere	1 848	926	3.39	3.39	3.45	0.98	0.98
violin	1 508	1 498	3.16	2.21	2.25	0.98	1.41
pig	3 560	1 843	3.26	3.24	3.13	1.03	1.04
rose	3 576	2 346	3.37	2.95	2.64	1.12	1.28
cathedral	1 434	2 868	2.25	1.00	0.19	5.27	11.86
blech	7 938	4 100	3.25	3.18	2.40	1.33	1.35
mask	8 288	4 291	3.19	3.12	1.93	1.62	1.65
skull	22 104	10 952	3.51	3.51	3.30	1.06	1.06
bunny	29 783	15 000	3.36	3.34	1.27	2.62	2.64
terrain	32 768	16 641	3.03	3.00	0.40	7.43	7.51
david	47 753	24 085	3.45	3.85	3.07	1.25	1.12
gargoyle	59 940	30 059	3.28	3.27	2.11	1.55	1.55

Table 1: Comparative results on different models (drawn on Figure 1). 'Dum' stands for the dummy vertex method to encode meshes with boundaries [6], and 'Ori' stands for the original Edgebreaker [12], and 'Ours' for the algorithm introduced here. The size of the compressed symbols (columns 'Dum', 'Ori' and 'Ours') is expressed in bit per vertex. Our algorithm has a compression ratio in weighted average 2.5 better than the other two. The 'sphere' model has the same encoding in all the above algorithms, but the range coder used has a lower performance since there are few symbols to encode. The 'cathedral' model is the output of an architecture modeling program, which is almost unstructured: all the connected components are pairs of triangles.

than 2 bits per triangle connectivity compression, with a $2 \cdot \log_2(|\mathcal{T}(\mathcal{S})|)$ bits over–cost for each half handle and for each boundary curve, and bests previous Edgebreaker's encoding for surfaces with boundaries. Moreover, the use of a range encoder significantly improves the final compression results.

References

[1] P. Alliez and M. Desbrun. Valence–driven connectivity encoding of 3D meshes. In *Computer Graphics Forum*, pp. 480–489, 2001.

[2] M. A. Armstrong. *Basic topology*. McGraw–Hill, London, 1979.

[3] L. C. Glaser. *Geometrical combinatorial topology*. Van Nostrand Reinhold, New York, 1970.

[4] A. Guéziec, F. Bossen, G. Taubin and C. Silva. Efficient compression of non–manifold polygonal meshes. *Computational Geometry*, 14(1–3):137–166, 1999.

[5] M. Isenburg and J. Snoeyink. Spirale reversi: reverse decoding of the Edgebreaker encoding. In *Canadian Conf. on Computational Geometry*, pp. 247–256, 2000.

[6] D. King and J. Rossignac. Guaranteed 3.67v bit encoding of planar triangle graphs. In *Canadian Conf. on Computational Geometry*, pp. 146–149, 1999.

[7] B. Kronrod and C. Gotsman. Efficient coding of nontriangular mesh connectivity. *Graphical Models*, 63:263–275, 2001.

[8] H. Lopes, J. Rossignac, A. Safonova, A. Szymczak and G. Tavares. Edgebreaker: a simple implementation for surfaces with handles. *Computers & Graphics*, 27(4):553–567, 2003.

[9] G. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Video & Data Recoding*, 1979.

[10] D. Poulalhon and G. Schaeffer. Optimal coding and sampling of triangulations. In *ICALP*, pp. 1080–1094, 2003.

[11] M. Schindler. Range encoder. www.compressconsult.com/rangecoder.

[12] J. Rossignac. Edgebreaker: connectivity compression for triangle meshes. *Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[13] J. Rossignac and A. Szymczak. Wrap & zip decompression of the connectivity of triangle meshes compressed with edgebreaker. *Computational Geometry*, 14(1–3):119–135, 1999.

[14] J. Rossignac, A. Safonova and A. Szymczak. 3D compression made simple: Edgebreaker on a corner–table. In *Shape Modeling International*, pp. 278–283. IEEE, 2001.

[15] D. Salomon. *Data compression: the complete reference*. Springer, Berlin, 2000.

[16] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *Transactions on Graphics*, 17(2):84–115, 1998.

[17] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface*, pp. 26–34, 1998.

The corresponding work was published in the proceedings of the Sibgrapi 2004, pp. 218–225 IEEE Press, 2004.

Appendix A: Main Algorithms

procedure **Compress**()

```
{ MV[]:= { 0 ... }, MT[]:= { 0 ... };          # tables for marking vertices and triangles
  T:=0, nB:=0;                                 # last triangle, nb of comps with boundary

  for (c:=0; c < 3·nT)                         # tests all corners for a boundary
  { if (c.o < 0) then                          # looks for edges on the boundary
    { c.o := -1; MV[c.p.v] := 1; MV[c.n.v] := 1; }   # marks vertex and unmarked boundary (-1)

  Write(topology, nT, nV);                     # writes number of triangles and vertices

  for (c:=0; c < 3·nT)                         # compress component with boundary
  { if (c.o < 1 && !MT[c.t]) then              # starts from an unvisited border triangle
    { WriteBoundary(c.n); T--;                 # encodes the geometry of this boundary
      CompressComponent(c); nB++; T++;         # starts the compression of this component

  for (c:=0; c < 3·nT)                         # compress component without boundary
  { if (!MT[c.t]) then                         # starts with an unvisited triangle
    { MT[c.t] := 1;                            # marks triangle as visited
      MV[c.v] := MV[c.n.v] := MV[c.p.v] := 1;  # marks vertices as visited
      WriteVertex(c.v, c.n.v, c.p.v);          # encodes the geometry of this triangle
      CompressComponent(c.r); T++;             # starts the compression of this component

  Write(topology, nB)                          # writes the nb of comps with boundary
}
```

procedure **Decompress**()

```
{ T:=0, N:=0, nC:=0, symb:="";                 # triangle, vertex; nb of comps, clers string
  Read(topology, nT, nV);                      # reads the number of triangles and vertices

  while (Read(topology,c0,c1))                 # reads each pair of handle
  { Match(c0, c1); }                           # fills the opposite corners
  Read(topology, nB)                           # reads the number of comps with boundary
  FC[] := { 0, ... }                           # first corner of each comp with boundary

  while (T<nT)                                 # parse the clers string
  { Read(clers,s); c:=3·T; symb[T++]:=s;       # reads and stores next symbol
    if (s='S' && c.l=-1)                       # on an S symbol not related to a handle
    then { i++; }                              # increases the number of parentheses
    if (s='E') then                            # on an E symbol
    { if (i>0) then { i--; } else              # if the S are not all matched
      { nC++;                                  # else increases the number of components
        if (nC ≤ nB) then { FC[nC]:=T; }       # stores the end of the component
        if (nC ≥ nB) then { symb[T++]:=P; }    # if the next comp has no boundary, add P

  SpiraleReversi(nC,nB,symb,FC)                # decodes the connectivity of the mesh
}
```

Figure 12: Main procedures.

Appendix B: Compression Algorithm

```
procedure CompressComponent(c)

{ repeat                                          # visits the triangle-spanning tree
  { MT[c.t] := 1; T++;                            # marks current triangle as visited
    CheckHandle(c);                               # checks for handles

    if (!MV[c.v] ≠ 1) then                        # tests whether the tip vertex was visited
    { Write(clers, C);                            # case C: unvisited tip vertex
      WriteVertex(c.v); MV[c.v] := 1;             # encodes the vertex and marks it
      c := c.r;                                   # continues with the right neighbor
    }
    else if (c.r < 0 || MT[c.r.t])                # tests whether right triangle was visited

    { then if (c.l < 0 || MT[c.l.t])              # tests whether left triangle was visited
      { then { Write(clers, E);                   # case E: both visited
               do { if (stack.empty())            # if the stack is empty
                    then { return ; }             # returns
                    c := stack.pop();             # otherwise, pops stack pushed by S
               while (MT[c.t]);                   # pops until finding a non–handle corner
      } else { Write(clers, R); c := c.l; }       # case R: right visited, left not; moves left

      else if (c.l < 0 || MT[c.l.t])              # tests whether left triangle was visited
      { then { Write(clers, L); c := c.r; }       # case L: left visited, right not; moves right
        else { Write(clers, S);                   # case S: both unvisited
               if (BoundaryOf(c) = -1)            # if the corner is on an unvisited boundary
               then { WriteBoundary(c);           # stores corner number (potential boundary)
                      MT[c.t] := - 3·T-2;         # else stores corner number (potential handle)
               else { MT[c.t] := 3·T+2; }         # encodes the internal boundary
               stack.push(c.l);                   # pushes the left corner on the stack
               c := c.r;                          # moves right
      }
    }
  }
}
```

```
procedure WriteBoundary(b)

{ b := b.n;                                       # decreases the boundary id and goes the next triangle
  while (b.r ≥ 0) { b := b.r; }                   # goes to right towards the boundary

  do                                              # starts the boundary encoding
  { b := b.n; b.o := -2;                          # starts from the next triangle and marks the boundary
    while (b.r ≥ 0) { b := b.r; }                 # goes to the rightmost triangle
    WriteVertex(c.p.v);                           # encodes the vertex
  while (b.r ≠ Bid);                              # stops when all the boundary is marked
}
```

```
procedure CheckHandle(h)

{ if (h.r ≥ 0 && MT[h.r.t]) then                  # checks for handle from the right
  { Write(topology, MT[h.r.t], 3·T+1);            # encodes pair of corners to be glued as a handle

  if (h.l ≥ 0 && MT[h.l.t]) then                  # checks for handle from the left
  { Write(topology, MT[h.l.t], 3·T+2);            # encodes pair of corners to be glued as a handle
}
```

Figure 13: Compression subroutines.

The corresponding work was published in the proceedings of the Sibgrapi 2004, pp. 218–225 IEEE Press, 2004.

Appendix C: Decompression Algorithm

```
procedure SpiraleReversi(T,N,nC,nB,symb,FC)
{ while (nC ≥ 0)
  { c:=-1; nC--;                                   # reads the clers string backwards
    while (nC ≥ nB || T ≥ FC[nC])                  # initiates first corner
    { switch (symb[T])                             # filters components with boundary
      { case C:                                    # test the next symbol
        { b:=3·T+1; c.o:=b; b.o:=c;                # case C:
          CloseStar(3·T+2,N--);                    # matches the right gate
        }                                          # closes star of the corner
        case L: { Match(c, 3·T+1); }               # case L: matches the right gate
        case R: { Match(c, 3·T+2); }               # case R: matches the left gate
        case S:                                    # case S:
        { Match(c, 3·T+1);                         # matches the right gate
          c:=3·T+2; if (c.o = -1)                  # if this is not a handle
          then { Match(c, stack.pop()); }          # matches the left gate
          else if (c.o < 0)                        # if it is an interior boundary
          { Match(c, -c.o);                        # goes backward on the boundary
            while (b.r ≥ 0) { b := b.r;}           # decodes the interior boundary
            ReadBoundary(b.p,N);
          }
        }
        case E: { if (c>0) then { stack.push(c); }}   # case E: push on the stack
        case P:                                    # case P:
        { Match(c, 3·T);                           # matches the gate
          CloseStar(3·T+1, N-2);                   # closes star of the next corner
          CloseStar(3·T+2, N-1);                   # closes star of the previous corner
          CloseStar(3·T, N);                       # closes star of the corner
          N-=3; nC--; c:=-1;                       # updates for a new component
        }
      }
      c:=3·T; T--;                                 # updates for a new component
    }
    if (nB≠0) ReadBoundary(3·FC[nC]+1,N)           # decodes exterior boundary
  }
}

procedure Match(b,c)        { c.o := b; b.o := c;        # associates opposite corners }

procedure ReadBoundary(b,N)
{ while (b.l ≥ 0) { b := b.l }                     # goes to the left towards the boundary
  do { ReadVertex(N); b.v := N;                    # decodes vertex geometry
       b := b.p; b.o := -2;                        # marks the corner
       while (b.l ≥ 0) { b := b.l; b.n.v := N; } } # updates all incident corners to the vertex
  while (b.v < 0);                                 # until closing the boundary
}

procedure CloseStar(c,N)
{ ReadVertex(N); b := c;                           # read new vertex
  while (b.l ≥ 0 && b.l ≠ c) { b.n.v := N; b := b.l; }   # assigns incident corner to the new vertex
  b.n.v := N; Match(b.p, c);                       # closes the star
}
```

Figure 14: Deompression subroutines.