Dynamic updates of succinct triangulations

Luca Castelli Aleardi, Olivier Devillers, Gilles Schaeffer

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Dynamic updates of succinct triangulations

Luca Castelli Aleardi — Olivier Devillers — Gilles Schaeffer

N° 5709

29 septembre 2005

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--5709--FR+ENG

Dynamic updates of succinct triangulations

Luca Castelli Aleardi* , Olivier Devillers†, Gilles Schaeffer‡

Thème SYM — Systèmes symboliques
Projet Geometrica

Rapport de recherche n° 5709 — 29 septembre 2005 — 23 pages

Abstract:

In a recent article, we presented a succinct representation of triangulations that supports efficient navigation operations. Here this representation is improved to allow for efficient local updates of the triangulations.

Precisely, we show how a succinct representation of a triangulation with m triangles can be maintained under vertex insertions in $O(1)$ amortized time and under vertex deletions/edge flips in $O(\lg^2 m)$ amortized time.

Our structure achieves the information theory bound for the storage for the class of triangulations with a boundary, requiring asymptotically $2.17m + o(m)$ bits, and supports adjacency queries between triangles in $O(1)$ time (an extra amount of $O(g \lg m)$ bits are needed for representing triangulations of genus g surfaces).

Key-words: graph encoding, succinct dynamic data structures, triangulations, geometric data structures.

This work has been supported by the French "ACI Masses de données" program, via the Geocomp project: `http://www.lix.polytechnique.fr/Labo/Gilles.Schaeffer/GeoComp/`

* INRIA - Geometrica and LIX, Ecole Polytechnique, 91128 Palaiseau
† INRIA - Geometrica
‡ LIX, Ecole Polytechnique, 91128 Palaiseau

Mises à jour dynamiques de triangulations succintes

Résumé :

Dans un travail récent, nous avons presenté une représentation succinte des triangulations permettant une navigation éfficace. Dans ce travail nous avons amélioré cette représentation afin de permettre une mise à jour locale éfficace de la triangulation.

Plus précisement, nous montrons comment une représentation succinte d'une triangulation ayant m triangles peut être mise à jour en temps constant après insertion de sommets, et en temps $O(\lg^2 m)$ amorti après suppression de sommets et flip d'arêtes.

En ce qui concerne l'espace utilisé, notre structure atteint l'entropie de la classe des triangulations avec un bord, nécessitant asymptotiquement de $2.17m + o(m)$ bits, et permet des requêtes d'adjacence entre triangles en temps constant ($O(g\ lgm)$ bits supplémentaires sont nécessaires pour représenter des triangulations de surfaces de genre g).

Mots-clés : Codage de graphes, représentations compactes, triangulations, structures de données géométriques.

1 Introduction

The most common data structures are explicit pointer based representations. For instance, a binary tree is typically implemented using for each node a pointer to its left and right sons: $O(m)$ pointers of size $O(\lg m)$ bits are used to represent a tree with m nodes. Since there are less than 4^m different such trees, $2m$ bits should be enough. This observation leads, given a class $\mathcal{C}_m$ of objects of size m, to the question whether these objects can be given a *succinct dynamic representation*, that is, whether one can design a data structure, such that:

- the representation allows to answer queries associated to the objects in constant time,
- the representation allows update of the objects in polylogarithmic time per operation.
- the storage cost of the representation R of an object in $\mathcal{C}_m$ matches asymptotically at first order the entropy of the class: $\text{size}(R) = \log_2 |\mathcal{C}_m| \cdot (1 + o(1))$, as m goes to infinity,

The first property ensures that the representation deserves to be considered as a data structure for the class. The second property gives the *dynamical* aspect of the representation. The third property yields its succinctness: in some cases, one contents with the weaker *compactness* property which reads $\text{size}(R) = O(|\mathcal{C}_m|)$.

While there is a large literature on compact data structures, dynamic succinct data structures were only developed recently, for some basic fundamental data structures as dynamic arrays [5], [12], dynamic dictionaries [13], dynamic bit-vectors [12].

Our interest is in geometric data structures. We consider, as a test case, triangulations of a surface of genus g with a simple boundary, since this is the most standard type of mesh found in the literature. Following an approach that proved successful for compression, we distinguish between the combinatorial information (the triangulation) and the geometric information (point coordinates), and drive our construction according to the combinatorial part. Our main result is the following:

Theorem 1. *There exists a representation of triangulations of a surface of fixed genus g with a simple boundary using asymptotically at first order 2.17 bits per triangles, thus matching the entropy of this class of triangulations. For a triangulation with m triangles the storage requires*

$$2.17m + O(g \lg m) + o(m) \text{ bits},$$

supporting standard navigation in $O(1)$ time and updates operations in

- *amortized constant time for vertex addition without data access, amortized $O(\lg m)$ time with data access,*
- *$O(\lg^2 m)$ amortized time for vertex deletion or edge flip.*

This *dynamic* result extends a previous *static* result [2] in which update operations were not dealt with. To our knowledge, this is the first dynamical version of succinct data structures for triangulations or complex graphs structures: related previous results

are static or non succinct, or deal with simpler structures like trees. For dynamic binary trees with n nodes, an optimal succinct representation has been proposed [13, 10] allowing basic modifications on the nodes, while navigation operations are performed in $O(1)$ time. If external $O(\lg n)$ bits data are associated to nodes, adding/deleting a leaf or inserting a node along an edge require $O(\lg^2 n)$ amortized time in [10]. The cost of update was later improved to $O((\lg \lg n)^{1+\varepsilon})$ [13]. Compact representations for static planar graphs were designed [8, 9, 6], combining succinct representations for trees with combinatorial decompositions of planar graphs as four pages embedding or canonical orderings. This yields space-efficient solutions that allow to perform local navigation in $O(1)$ time, but the use of non trivial combinatorial decompositions makes it hard to maintain the structure under local modifications of the graph. With a different approach, based on short separators, a compact representation for separable graphs was developed in [3] that uses $O(n)$ bit, supports local navigation in $O(1)$ time and for which updates appear possible in practice: however it is difficult to give good bounds on the cost of updates in this approach, again because the separators are hard to track under local modifications.

Our approach is similar to the hierarchical 3 level approach used in [10, 12] for binary trees: we decompose the structure into *small* regions themselves cut into *tiny* pieces. However, these works can use canonical total orderings like the standard preorder on trees, which is relatively stable under insertion/deletion of leaves. This fails on graphs, because canonical orders on vertices are typically perturbed at large distance by local modifications of the graph: as illustrated by Fig. 5, this is the main new difficulty we have to deal with.

Contribution Our main contribution, as stated in Theorem 1 above, is to prove that we can theoretically update and traverse a triangulation with good complexity using an asymptotically optimal storage. We also discuss the cost of providing access to attached data (like vertex coordinates).

Practical implementations [4] are far from that and use $6m + n$ pointers (each triangle knows its neighbors and its incident vertices; each vertex has a reference to an incident face): for triangle meshes with n vertices this yields in practice $7 \cdot 32 = 224$ bits per triangle, or in theory $7 \log m$ bits per triangle. Even if this work is clearly theoretical, it suggests ideas for more practical trade-offs between storage and access time.

2 Overview of the solution

As in previous work on succinct representations [2] our structure relies on a 3 level description of the initial triangulation $\mathcal{T}$.

The underlying level consists of a set of sub-triangulations of size $\Theta(\lg m)$ that is a partition of the m triangles of $\mathcal{T}$: we will call *tiny triangulations*, denoted by $\mathcal{T}_{ij}$ (and stored in Table A), those sub-triangulations having between $\frac{1}{12}\lg m$ and $\frac{1}{4}\lg m$ triangles. Their tiny size ensures that we can store the catalog of all possible different triangulations having up to $\frac{1}{4}\lg m$ triangles using a sub-linear amount of space.

Each tiny triangulation is a planar triangulation with a boundary of arbitrary size (no assumptions are made on the way we partition the m triangles of $\mathcal{T}$). We call *multiple vertices* those boundary vertices that are shared by more than 2 tiny triangulations;

these multiple vertices cut the boundary of tiny triangulations in pieces called *sides*.

Tiny triangulations are packed together to form bigger connected triangulations of about $\Theta(\lg^2 m)$ triangles, called *small triangulation* and denoted $\mathcal{ST}_i$. A small triangulation contains between $\frac{1}{3}\lg m$ and $\lg m$ tiny triangulations.

The second level of the structure is designed to describe adjacency relations between tiny triangulations: this is done by introducing a graph G whose nodes and arcs correspond to tiny triangulations and neighborhood relationships between them. G is a planar map, whose faces have degree at least 3 and having loops and multiple edges (auto-intersections of boundaries are allowed). A small triangulation $\mathcal{ST}_i$ maps to a group of size $\Theta(\lg m)$ of nodes of G, denoted by G_i.

The upper level is needed to describe adjacency relations between small triangulations: for this purpose we introduce a graph F, represented with true pointers on $O(\lg m)$ bits, which links adjacent small triangulations.

As described in [2], F is represented explicitly using $\lg m$ size pointers: this requires $o(m)$ for the storage since $|F| = O\left(m/\lg^2 m\right)$, which is asymptotically negligible. As opposed to that, adjacencies in G are represented by pointers of size $\lg\lg m$ that are local to each small region G_i (which has size $|G_i| = O(\lg m)$). The overall cost of these local pointers also sum up to a sub-linear storage. Finally the dominant linear cost arises from the storage for each vertex of G of a pointer to a tiny triangulation in Table A.

The overall succinctness ultimately comes from the fact that a tiny triangulation appearing several times in $\mathcal{T}$ is explicitly stored only once, with several nodes of G pointing at it.

3 Preliminaries

The model of computation we use is the standard RAM machine, with access and arithmetic operations in $O(1)$ time on words of size $\lg m$. In particular dynamic memory allocation can done for blocks of 2^k consecutive words in time $O(2^k)$.

3.1 Dynamic operations on the triangulation

Our present work extends previous representation that allowed efficient navigation in the triangulation: all navigation operations between triangles provided in [2] are also supported here in $O(1)$ time.

In addition to these static queries we present a complete set of basic operations that permit to perform local modifications of the triangulation:

- $Insert(\triangle)$: adds a degree 3 vertex in triangle $\triangle$.
- $Delete(v)$: deletes a degree 3 vertex.
- $Flip(\triangle, v)$: flips the edge of $\triangle$ opposite to vertex v.

These are the standard operations on triangulations without boundary. The insertion/deletion of a vertex on the boundary is dealt in the same manner so that we do not discuss it explicitely.

3.2 Previous results used

Tree partitioning. As mentioned in section 2 our structure is based on a two levels decomposition of the initial triangulation: first in pieces of size $\Theta(\lg m)$ (tiny triangulations) and then regroupings of about $\Theta(\lg m)$ tiny pieces (small triangulations).

Two key procedures that help us in that decomposition are described by the following lemmas. The first one concerns the decomposition of binary trees [10] and it will be used in the construction/update of tiny triangulations.

Lemma 1. *Given a binary tree $\mathcal{B}$ on n nodes (each node has degree at most 3) and a positive integer parameter M, it is possible to produce in $O(n)$ time a partition of $\mathcal{B}$ into sub-trees $\mathcal{B}_i$, such that the size of every subtree satisfies $3M \geq \|\mathcal{B}_i\| \geq M$.*

A different result [7] concerning ordered trees will be useful for constructing and updating small triangulations:

Lemma 2. *Let us consider an ordered tree $\mathcal{B}$ on n nodes and a positive integer parameter $M \geq 2$. We can compute in $O(n)$ time a family of sub-trees covering the original tree $\mathcal{B}$, that intersect only at their roots and satisfy the following constraints on their sizes: every sub-tree $\mathcal{B}_i$ satisfies $\|\mathcal{B}_i\| \leq 3M - 4$. Furthermore, if $\mathcal{B}_i$ does not contain the root we have $\|\mathcal{B}_i\| \geq M$.*

Succinct dynamic data structures. One crucial aspect in our strategy for maintaining the representation under updates concerns the organization of memory storage. A key tool for managing memory used here is a dynamic data structure called *extendible array*.

An extendible array contains n equal-size records (with index between 0 and $n-1$) and supports static and dynamic operations: accessing a record given its index, creating a new record with index n (*grow*), discarding the record with index n (*shrink*). The nominal size of an array with n records each of size r is of nr bits.

When considering a collection of extendible arrays, we can add the following dynamic operations: $create(r_i)$ returns a new extendible array having record size r_i; $destroy(A)$ frees the memory corresponding to the array A in the collection. If the ith array A_i in the collection has n_i records, each of size r_i, we say that the nominal size of the entire collection is $\sum_i n_i r_i$.

Some works introducing space-efficient solutions for the implementation of dynamic arrays have been proposed ([5]). The description of our memory storage is based on some recent improvement presented in [13] and expressed in the following form:

Lemma 3. *Let us consider a collection of a extendible arrays whose nominal size is s and denote by w the machine word size. Then there exists a succinct representation of the*

collection that uses $s+O(aw+\sqrt{saw})$ *bits of space and supports, in our model of computation,* access *operations (reading/writing) in* $O(1)$ *time and* create, grow *and* shrink *operations in* $O(1)$ *amortized time.*

We shall need an extra operation on extendible arrays: the *deletion* of an arbitrary record in an array. When record i is deleted in an array of size $n+1$, record n is copied at its place and a *shrink* is performed. This operation requires also the update of references to record n. The amortized cost of the copy/shrink operations is $O(1)$, and in our case the update cost will be easy to evaluate.

Enumeration of triangulations Since our representation is based on the optimal coding of all the triangulations having up to $\Theta(\lg m)$ triangles, we cite an enumeration result for this class of objects, expressed in the following form (for a sketch of the proof see [1]):

Lemma 4. *The number of triangulations of a polygon having* p *triangles, with interior vertices but not multiple edges, is bounded by* $2^{2.175p}$.

4 Catalogs of all tiny triangulations and boundary bit vectors

Tiny triangulations In our previous work [2], Table A contained $\frac{1}{4}\lg m$ pointers to Tables A_i, each storing the list of all triangulations of size i. Elements in Table A_i, one for every triangulation, are pointers to an explicit representation $A_{i,j}^{explicit}$ of the corresponding combinatorial structure. A number of informations was stored to allow efficient navigation in the tiny triangulation, as the adjacency relations between triangles. Moreover, an $O(i)$ code for each triangulation of size i is computed and associated to the $A_{i,j}^{explicit}$, following a recent compact encoding ($4i-2$ bits suffice, see [11]): this does permit to index efficiently in tables A_i, ensuring a linear time cost for the construction of the entire represnetation (see [1] for more details about the construction phase costs).

In order to deal with local updates of the triangulation, we enrich the previous description to obtain a new explicit representation that allows to perform local modifications in tiny triangulations. In particular, to implement local vertex insertions in $O(1)$ time, we add the following fields to the explicit representation $A_{i,j}^{explicit}$:

— $A_{i,j}^{explicit}.add\_vertex$ is a table containing all possible triangulations of size $i+2$ obtained by adding a vertex to the current tiny triangulation in A_{ij}. The kth record in this table (one for each triangle in the triangulation) contains a $\Theta(\lg m)$ index that permits to access to the modified triangulation in $A_{i+2,j'}$, obtained adding a vertex to the kth face.

— $A_{i,j}^{explicit}.del\_vertex$ is a similar table for all possible deletions of degree 3 vertices.

— $A_{i,j}^{explicit}.flip\_edge$ is a similar table for all possible flips.

Lemma 5. *The storage of Table* A*, and of all the information associated with Tables* A_i *requires asymptotically* $o(m)$ *bits.*

Proof. As long as the information attached to a tiny triangulation of size i remains polynomial in i, the cost of the exhaustive catalog remains sublinear since it consists of $O(2^{2.17\frac{1}{4}\lg m})$ entries of polylogarithmic size. □

Observe that the field *add_vertex* allows to perform the local insertion of a vertex in time $O(1)$, unless the tiny triangulation becomes too big and must be split. For pure vertex insertions this cannot happen too often and $O(1)$ amortized cost is obtained. The fields *del_vertex* and *flip_edge* also allow $O(1)$ time for local operations inside a tiny triangulation but at a theoretical level they do not help much because, as we shall see, non local operations may be required arbitrarily often.

Boundary bit vectors The catalog of all boundary bit vector consists of Tables B_{pq}, containing all bit-vectors of size p and weight q for $p < \frac{1}{4}\lg m$ together with the information needed to answer rank and select queries in constant time. Observe that these tables contain $O(m^{\frac{1}{4}})$ entries of polylogarithmic size: they can be constructed in sublinear time and their storage requires a negligible amount of space. In particular these Tables can be enhanced to support adding/removing/changing one bit in the vector in constant time as was done for Table A (this can be used to implement efficient insertion of boundary vertices).

5 Map of tiny triangulations

In [2] we introduce a map G (resp. F) to describe the adjacency relations between tiny (resp. small) triangulations.

In this section we provide the description of the memory organization relative to the submap G_i, which corresponds to those nodes of G belonging to the same small triangulation $\mathcal{ST}_i$. The map G (as well as F) may have multiple arcs or loops but all its faces have degree ≥ 3. Each arc of G between $\mathcal{TT}_j$ and $\mathcal{TT}_{j'}$ corresponds to a side shared by $\mathcal{TT}_j$ and $\mathcal{TT}_{j'}$.

5.1 Detailed description of the memory organization

Here we give a detailed description of the 5 collections of extendible arrays, associated with a node G_i, that store the information relative to the combinatorial structure, the boundary labelling and the adjacency relations of all the tiny triangulations within the same small triangulation. These notations extend and are coherent with the notations of [2].

- An extendible array S_i is used to store adjacency relations between small triangulations (represented by arcs in map F): this array has $O(\lg^2 m)$ records, each of size $\Theta(\lg m)$ bits, listing the neighbors of the sub-map G_i. Although S_i may have size $O(\lg^3 m)$, it is much smaller on average (see Lemma 6 below).

- An extendible array T_i is used to store informations relative to each tiny triangulation: the record $T_i[j]$, relative to the tiny triangulation $\mathcal{TT}_{i,j}$ consists of the $O(\lg\lg m)$ bit concatenation of the following fields:

- $G^t_{i,j}$ is the number of triangles in $\mathcal{TT}_{i,j}$.
- $G^b_{i,j}$ is the size of the boundary of $\mathcal{TT}_{i,j}$.
- $G^s_{i,j}$ is the degree of the node $G_{i,j}$ in the map G_i.
- $G^{PA}_{i,j}$ is a reference to the record, in an array of the collection PA_i below, where the combinatorial structure of $\mathcal{TT}_{ij}$ is implicitly stored as a reference in Table A.
- $G^{PB}_{i,j}$ is a reference to the record, in an array of the collection PB_i below, where the boundary labelling of $\mathcal{TT}_{ij}$ is implicitly stored as a reference in Table B.
- $G^E_{i,j}$ is a reference, in an array of the collection PE_i below, to the list of halfarcs incident to node G_{ij}.

Each of these fields can be represented on $O(\lg\lg m)$ bits, since map G_i has at most $O(\lg m)$ nodes and $O(\lg^2 m)$ arcs.

- A collection PA_i of extensible arrays is used to store implicitly (as references to Table A) the combinatorial structures of the tiny triangulations of G_i. The collection PA_i consists of $O(\sqrt{\lg m})$ extendible arrays: the kth array in this collection has records of size $k\sqrt{\lg m}$. Each record is meant to contain a pair of the form:
  - The index of a tiny triangulation in Table A. This index has size $2.17r$ bits if the triangulation has size r.
  - A $\lg\lg n$ bits index used as backward pointer in table T_i.

  Data pairs will be added, removed and modified in PA_i but we shall ensure that each pair is always affected to the array with smallest index k among arrays with large enough records for it to fit. The words representing these pairs, encoding implicitely a tiny triangulation, are padded out to the next multiple of $\sqrt{\lg m}$: let us observe that this padding operation does wast at most $\sqrt{\lg m}$ bits per tiny triangulation.
- A collection PB_i of extensible arrays is used to store implicitly (as references to Table B) the boundary bit vectors. The collection PB_i is modeled after PA_i with the only difference that it contains indices of boundary bit vectors in Table B, instead of indices of tiny triangulations. For a tiny triangulation with boundary size p and q sides, these references require $O(q\lg p)$ bits.
- A collection PE_i of extensible arrays is used to store the (ccw sorted) lists of half arcs incident to the nodes $G_{i,j}$.

  For each half-arc, the following $O(\lg\lg m)$ bit fields are stored:
  - $G^T_{ij}.source$ is a reference in Table T_i to the record associated with G_{ij} (source node of the arc);
  - $G^T_{ij}.target$ is a reference in Table T_i to the record associated with $G_{i'j'}$, which is pointed by the halfarc ($G_{i'j'}$ is the target of the halfarc);

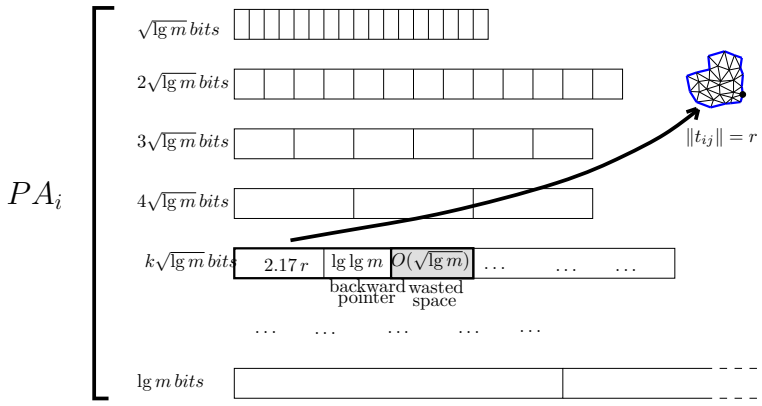

Figure 1: Collection PA_i storing the implicit representations of tiny triangulations lying in a small triangulation $\mathcal{ST}_i$

- $G_{ij}.back$ is the index k' of the side corresponding to the current arc in the numbering of sides at the opposite node $G_{i'j'}$.
- $G_{ij}.small$ is a pointer to the small triangulation $G_{i'}$, in the list of the neighbors of G_i in the main map F (if $G_{i'} = G_i$, then G_{ij}^{small} is set to 0).

A node $G_{i,j}$ can have degree between 1 and $\lg m$, so that it requires between $\lg\lg m$ and $\lg m \lg\lg m$ bits.

The collection PE_i is thus taken of $\ell = \sqrt{\lg m \lg\lg m}$ arrays with records of size $k\ell$ for $k = 1, \ldots, \ell$. Each node is stored in a record of the array with smallest index k having large enough records. In this way, $O(\ell)$ bits per edge are wasted, but this does not matter since the total number of edges in G is $O(m/\lg m)$.

5.2 Storage analysis

Lemma 6. *The storage of maps G and F requires asymptotically $2.175m + o(m)$ bits.*

Proof. Let us first recall some basic identities on the number of arcs and nodes in maps G and F. A small triangulation contains at most $\lg m$ tiny triangulations, hence between $\frac{1}{12} \cdot \frac{1}{3}\lg^2 m$ and $\frac{1}{4}\lg^2 m$ triangles. Since the number of tiny triangulations neighboring G_{ij} is $G_{ij}^s \leq \frac{1}{4}\lg m$, the number of arcs in sub-map G_i is at most $O(\lg^2 m)$. There are $n = \Theta(m/\log^2 m)$ nodes in G, but the overall number a of arcs is linear in n: indeed G has only faces of degree ≥ 3, so that $2a = \sum_f d(f) \geq 3f$ with f the number of faces of G, and, recalling that G is a map of genus g like $\mathcal{T}$, Euler's formula reads $3(a+2) = 3n + 3f + 6g$, that is, $a \leq 3n + 6(g-1)$ (an analogous relation holds for arcs and nodes in main map F).

Since the number of nodes of G is at most $\frac{m}{\frac{1}{12}\lg m}$, we obtain the following bound on the sum of their degrees (which is twice the number of arcs):

$$2\sum_{ij} G^s_{ij} \leq 2(3\frac{m}{\frac{1}{12}\lg m} + 6(g-1))$$

For map F a similar relation holds, and recalling that F has at most $\frac{m}{\frac{1}{12}\frac{1}{3}\lg^2 m}$ nodes, its number a_F of arcs (twice the sum of degrees of its nodes) satisfies:

$$a_F = 2\sum_{i} F^s_i \ \leq \ 2(3\frac{m}{\frac{1}{36}\lg^2 m} + 6(g-1))$$

For each of the extendible arrays described above we compute their nominal size together with the auxiliary storage, needed according to lemma 3.

• The amount of space needed to store all pointers between adjacent small triangulations is negligible, since tables S_i (one for each small triangulation) contain in overall $O(\frac{m}{\lg^2 m})$ records, each on $O(\lg m)$ bits.

• Let us consider the extendible array T_i: each of its records has size $O(\lg\lg m)$. Since in a block there at most $O(\lg m)$ tiny triangulations, the nominal size of each T_i is $O(\lg m \lg\lg m)$ then we can conclude that in overall the storage of all arrays T_i take $O(\frac{m}{\lg^2 m}\lg m \lg\lg m)) = o(m)$.

• Let us consider the information stored in the collection PA_i, which gives the dominant term for the storage of our representation: let us compute at first its nominal size (when computing the nominal size of the collections PA_i, as well for the PB_i and PE_i, we consider also the amountof space wasted by the padding phase).

Each record in an array of the collection PA_i contains a pointer to the explicit representation of a tiny triangulation in tables A_i and a backward reference to array T_i.

Concerning the explicit representation: following lemma 1, to represent a tiny triangulation having G^t_{ij} triangles we need $2.17G^t_{ij}$ bits, which implies that the nominal size s_i of the whole collection PA_i for a given small triangulation is given by the sum:

$$s_i = \sum_j (2.17G^t_{ij} + \lg\lg m + \sqrt{\lg m}) = O(\lg^2 m)$$

while the auxiliary storage needed, as expressed by lemma 3, is:

$$a\lg m + \sqrt{s_i a \lg m} = \sqrt{\lg m}\lg m + \sqrt{s_i\sqrt{\lg m}\lg m} =$$

$$= O(\lg^{3/2} m) + O(\sqrt{\lg^2 m \lg^{3/2} m})$$

Then the storage of all the collections of extendible array PA_i, one for each small triangulation $\mathcal{ST}_i$, is in overall

$$\sum_{ij}(2.17G^t_{ij} + \lg\lg m + \sqrt{\lg m}) + O(\frac{m}{\lg^2 m}\sqrt{\lg m}\lg m) +$$

$$+O(\frac{m}{\lg^2 m}\sqrt{\lg^{7/2} m}) = 2.17m + O(\frac{m}{\lg^{1/4} m})$$

- A similar argument holds for the collections of extendible arrays PB_i and PE_i.

□

6 Data structures updating

Before providing a detailed description of the algorithms that maintain the triangulation under local modifications, we prefer to describe some basic procedures designed to perform updates on arrays introduced in previous Section.

For each of these arrays storing the information relative to a small triangulation we give a short description of the updating operations with an analysis of their relative computational cost.

6.1 Updating the connectivity/boundary coloring of tiny triangulations

In this case we only have to modify the implicit representation of the tiny triangulation concerned: recall that the combinatorial structure, together with the boundary coloring, are implicitly represented as references, stored in arrays of the collections PA_i and PB_i.

Lemma 7. *Modifications of the combinatorial structure or of the boundary coloring of a tiny triangulation can be implemented in $O(1)$ amortized time.*

Proof. Once the new representation and reference (with a table look-up in tables A_i) are computed, we first check if their size fit the size of records storing the old implicit representation. If it is not the case we have to perform the following steps to avoid of wasting unused space after the memory relocation:

- erase from array $PA_i[k]$ (array having records of size $k\sqrt{\lg m}$) the record containing the old reference;

- create a new record in an array $PA_i[k']$ (k' is chosen such that the new implicit representation can be stored on $k'\sqrt{\lg m}$ bits);

- finally the empty record just erased, will be filled with the last record in $PA_i[k]$: since this record has moved to a new location we use backward pointers to table T_i for updating the references that changed.

Since access operations (reading/writing) on extendible arrays are performed in $O(1)$ time, the complexity is dominated by the cost of creating a new record in the collection PA_i which takes $O(1)$ amortized time. An analogous argument holds for the update of the collection PB_i. □

Lemma 8. *If a tiny triangulation is obtained by applying a split/fusion procedure as described in section 7 the above updating operations require $O(\lg m)$ time.*

It suffices to observe that in this case the most expensive operation is the computation of the new implicit representation of the tiny triangulations, which is performed with a binary search in tables A_i and B_{pq} and takes $O(\lg m)$ time.

6.2 Topological modifications in graph G_i

These events occur after a call of the decomposition procedure or after performing a flip/vertex deletion that disconnect tiny triangulations.

6.2.1 Modifications in table T_i

The creation/deletion of tiny triangulations requires to add/delete records in table T_i (extendible array): record are always added at the end (after the last element inserted), while deletions require moving the last record of T_i at the erased location (as for collections PA_i, this operation is needed to avoid wasting unused memory).

Moving/adding record in table T_i implies to update all the references to elements that change locations: these include backward pointers from arrays PA_i, PB_i and mainly all the references stored in arrays of the collection PE_i.

Lemma 9. *The insertion/deletion of a new record in table T_i together with the update of all relative references can be performed in $O(\lg m)$ amortized time.*

The only insertion/deletion of a record in T_i takes $O(1)$ amortized time. Updating all backward pointers and references in collection PE_i requires $O(\lg m)$ time: since there are at most $O(\lg m)$ references that change (reading/writing operations are performed in constant time on extendible arrays).

6.2.2 Modification of arcs of G_i: implementation consequences

Adding/Deleting arcs in map G_i produce mainly two kind of modification in the data structures.

- First we have to add/remove records in arrays of the collection PE_i where inter tiny triangulations adjacency relations are stored: recall that halfarcs incident to a node v are stored consecutively reflecting the ccw orientation around v; furthermore halfarcs are regrouped following the degree of their common incident node (an array of the collection PE_i stores list of halfarcs relative to nodes of about the same degree). If the degree of a node v does change drastically, we have to erase the complete list of its incident halfarcs and to move it to another arrays in the collection: as for previous updates of extendible arrays we have to fill erased records).

- Since the degree of the node v did change, we have also to update all the labelings of halfarcs that have v as target: in the general case there are $O(\lg m)$ such labels to update in a small triangulation.

Lemma 10. *Local Modifications in graph G_i require $O(\lg m)$ amortized time.*

From the computational point of view, adding/deleting records in arrays of the collection PE_i may require $O(\lg m)$ operations, each performed in $O(1)$ amortized time.

Updating labelings of halfarcs does cost $O(\lg m)$ reading/writing operations on arrays in the collection PE_i. All references in table T_i are also updated in $O(\lg m)$ time.

7 Two auxiliary key subroutines

In this section we describe two sub-routines used in the updating of the triangulation.

For the case of vertex insertions we have to call them only if the size of some tiny triangulation does change, exceeding the maximum size allowed for a valid tiny triangulation: this happens after $\Theta(\lg m)$ modifications, then their cost can be amortized.

For the case of edge flipping and vertex deletion there is no way to avoid the usage of these two expensive procedures, even if the local sizes of the tiny triangulations concerned do not change significantly.

7.1 Splitting a tiny triangulation

Here we consider a key sub-routine designed to perform the decomposition and the update of a not valid tiny triangulation t: this occurs when its size exceeds the upper bound of $\frac{1}{4}\lg m$ triangles.

Lemma 11. *Splitting a triangulation having between $\frac{1}{4}\lg m$ and $\frac{3}{8}\lg m$ triangles into two valid tiny triangulations requires $O(\lg m)$ amortized time.*

Proof. Let us first decompose $\mathcal{TT}_{ij}$ into two sub-triangulations $\mathcal{TT}_{ij'}, \mathcal{TT}_{ij''}$ whose sizes are now valid: it suffices to compute a spanning tree of the dual graph of , and to apply the partition procedure of lemma 1 to the resulting binary tree.

The initial $\Theta(\lg m)$ triangles are now partitioned into two triangulations of at most $\frac{1}{4}\lg m$ triangles each. All old boundary edges do still belong to the boundary of $\mathcal{TT}_{ij'}$ or $\mathcal{TT}_{ij''}$; a certain number of old internal edges are now lying on the boundary that separates $\mathcal{TT}_{ij'}$ and $\mathcal{TT}_{ij''}$.

Furthermore we claim that only two multiple vertices are created (and thus only 4 sides are created or modified). This guarantees that only a constant number of neighbors of $\mathcal{TT}_{ij}$ change their degree in map G.

In general, when a new decomposition procedure is performed, for each new tiny triangulation t' of size r' we have to

- compute a $O(\lg m)$ bits code for $\mathcal{TT}_{ij'}$ (following the linear encoding proposed by Poulalhon and Schaeffer [11], see section 4), which is used to 'index' in table A_i;

- perform a binary search in table $A_{r'}$: the result of this procedure is a pointer that will be used as an implicit optimal representation for the combinatorial structure of $\mathcal{TT}_{ij'}$;

- perform a binary search in table B_{p_k,q_k}: this gives an implicit representation of its boundary coloring (together with the implementation of the corresponding bit-vector).

All previous operations require $O(\lg m)$ time.

Finally, we have to modify graph G_i in order to take account of new adjacency relations between tiny triangulations.

Concerning the updates of the data structures involved, we have to:

- add a record in table T_i corresponding to the new created tiny triangulation $\mathcal{TT}_{ij'}$ (see Lemma 9).

- modify in map G_i the list of arcs incident to G_{ij}: updating arrays in collection PE_i requires to modify only $O(\lg m)$ old references (see Lemma 10).

- update the labels of the arcs of G_i: when we insert a new arc $e = (G_{i1}, G_{i2})$ in G_i, the lists of labels of all the halfarcs incident to G_{i1} and G_{i2} have to be updated. Since only a constant number of nodes do change their degree in map G_i this phase takes $O(\lg m)$ time.

The last three steps take $O(\lg m)$ amortized time, in accord to Lemmas 9 and 10.

□

If a not valid tiny triangulation $\mathcal{TT}_{ij}$ is obtained by the fusion of 2 or 3 tiny triangulations we can split $\mathcal{TT}_{ij}$ into valid tiny triangulations updating the structure in $O(\lg m)$ time.

It suffices to observe that we need to call the splitting procedure on $\mathcal{TT}_{ij}$ only a constant number of times.

7.2 Splitting a small triangulation

The procedure we are going to describe in this section is designed to decompose a small triangulation whose size exceeds the maximum size allowed: here is not the number of triangles but the number of tiny triangulations in a small one that does matter.

Our main aim it to show that, as in the case of tiny triangulations, only a negligible number of adjacency relations do change: hence we can design a clever way of decomposing and updating a small triangulation that is really better than simply reconstructing the entire data structures involved.

Before describing our procedure that performs the splitting of a small triangulation, let us state a useful result on weighted binary trees, inspired by Lemma 1.

Lemma 12. *Let us consider a binary tree $\mathcal{B}$ having n nodes (each of degree at most 3) to which we associate non negative weights w_i satisfying: $\sum_i^n w_i = K$ and $w_i \leq \frac{2}{3}K$. Then is possible to obtain in linear time a decomposition of $\mathcal{B}$ into two sub-trees whose weights satisfy*

$$\sum_{i' \in I'} w_{i'} \leq \frac{2}{3}K + \max_i w_i \ , \ \sum_{i'' \in I''} w_{i''} \leq \frac{2}{3}K + \max_i w_i$$

where I' (resp. I'') denotes the set of indices of nodes belonging to the sub-tree $\mathcal{B}'$ (resp. $\mathcal{B}''$).

Proof. Let us first perform on $\mathcal{B}$ a post order traversal: this allows to compute sub-tree weights, that we use to label halfedges. We traverse $\mathcal{B}$ by starting on a leaf and conquering nodes in post order: when visiting a node n_i we compute the total weight of the sub-tree

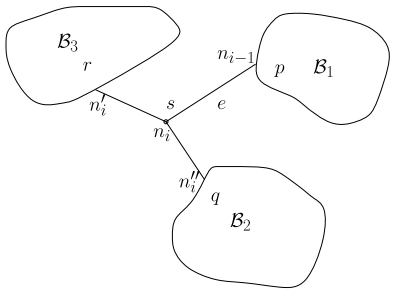

Figure 2: Split of a weighted binary tree

$\mathcal{B}_i$ having n_i as root, say K_i; the remaining nodes in $\mathcal{B} - \mathcal{B}_i$ have total weight $K - K_i$. Moving from n_i to its ancestor, we label the traversed edge e with the pair $(K_i, K - K_i)$, which represents the total weights of sub-trees separated by e: as result of this preprocessing phase we get a labelled binary tree with two labels per edge.

We now design a greedy traversal of $\mathcal{B}$ that will provide the decomposition requested. First choose as starting point for our traversal the leaf with maximal weight: after conquering a node n_i, we move to the adjacent non traversed node with maximum weight between its neighbors (since we enter in a node by traversing an edge, there are at most two such neighbors to visit): our visit does end when the traversed half-edge has label $K_i \geq \frac{1}{3}K$.

Obviously there is a step i at which the visited edge has a label $\geq \frac{1}{3}K$.

At step i, we pass from node n_{i-1} to node n_i (with weight s): let us call $\mathcal{B}_1$ the traversed sub-tree rooted at n_{i-1}, having weight p. From n_i we could move to one of its neighbors n_i' and n_i'': let us call $\mathcal{B}_2$ and $\mathcal{B}_3$ the corresponding sub-trees, having respectively weights r, q (let us suppose that $r \geq q$).

We know that the weights of these sub-trees satisfy $p + r + q + s = K$. Moreover, since our visit did not halt at step $(i-1)$, we deduce that $p < \frac{1}{3}K$.

We claim that our algorithm always halts returning a valid decomposition. If not, as we are supposed halting after visiting n_i, we should have $p+q+s > \frac{2}{3}K + s$, which also implies $r > \frac{1}{3}K$. This leads to a contradiction since it would hold $p + q + r + s > K + s$. □

Lemma 13. *Splitting a small triangulation $\mathcal{ST}_i$ together with updating all data structures involved requires $O(\lg^2 m)$ amortized time.*

Proof. Let us first decompose the small triangulation $\mathcal{ST}_i$ (its size $\|\mathcal{ST}_i\|$ is in term of tiny triangulations contained) into two sub-triangulations whose sizes are less than $\frac{2}{3}\|\mathcal{ST}_i\|$.

One possible strategy is to compute a spanning tree of map G_i (the map describing adjacency relations between tiny triangulations in $\mathcal{ST}_i$): we get a tree $\mathcal{B}$, which is not necessarily a binary tree, with $\|\mathcal{ST}_i\|$ nodes.

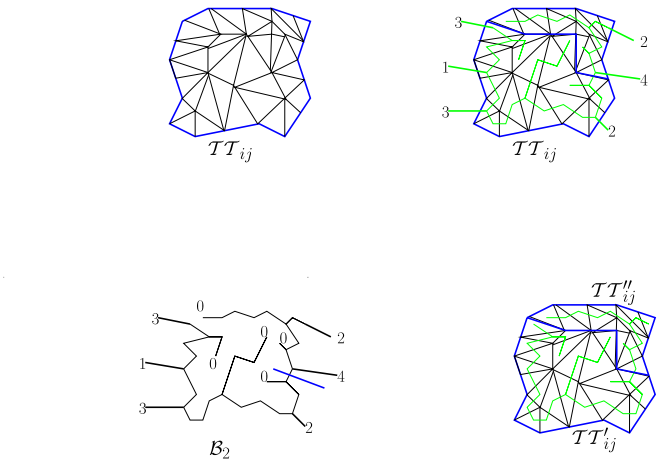

Figure 3: In these Pictures are depicted some steps of the decomposition procedure described in Lemma 13: the split of $\mathcal{TT}_{ij}$ is performed by applying Lemma 12 to the weighted tree $\mathcal{B}_2$.

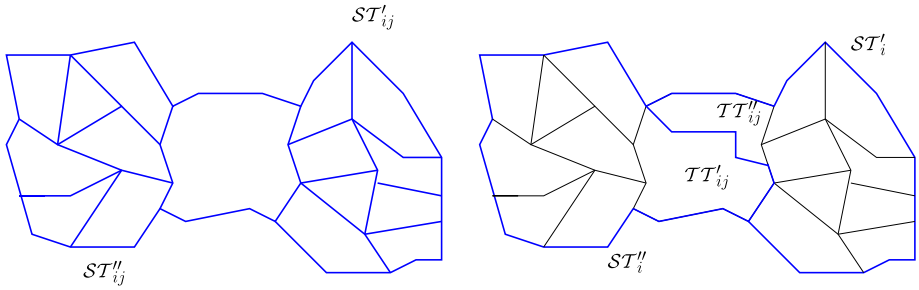

Figure 4: Split of a small triangulation: on the left the initial small triangulation. On the right the decomposition obtained applying Lemma 13. The split of the tiny triangulation $\mathcal{TT}_{ij}$ induces the split of $\mathcal{ST}_i$ into two subtriangulations $\mathcal{ST}'_i$ and $\mathcal{ST}''_i$.

If we apply lemma 2 to $\mathcal{B}$ we can chose the parameter M to obtain a decomposition into two sub-trees $\mathcal{B}', \mathcal{B}''$ sharing at most one node n_{ij} and whose sizes are at most $\leq \frac{2}{3}\|\mathcal{B}\|$.

Let us call n_{ij} the node belonging to $\mathcal{B}'$ and $\mathcal{B}''$ and $\mathcal{TT}_{ij}$ the corresponding tiny triangulation; let s be the number of sides of $\mathcal{TT}_{ij}$.

We now construct a binary tree $\mathcal{B}_2$ having $\|\mathcal{TT}_{ij}\| + s$ nodes to which we associate non negative weights satisfying Lemma 12.

Let us compute a spanning tree $\mathcal{B}_2$ of the dual graph of the triangulation $\mathcal{TT}_{ij}$, which is a binary tree on $\|\mathcal{TT}_{ij}\|$ nodes.

For each side of $\mathcal{TT}_{ij}$ there exists one arc a_k in $\mathcal{B}$ connecting n_{ij} to its adjacent nodes (corresponding to the tiny triangulations adjacent to $\mathcal{TT}_{ij}$ within $\mathcal{ST}_i$).

For each such arc a_k consider the sub-tree $\mathcal{B}_k$ obtained cutting $\mathcal{B}$ along a_k: this gives one sub-tree for each side of $\mathcal{TT}_{ij}$ which contains at most $\|\mathcal{B}_k\| \leq \frac{2}{3}\|\mathcal{ST}_i\|$ nodes (corresponding to tiny triangulations within $\mathcal{ST}_i$).

Let us set to 0 the weight of all its nodes (internal and external).

For each side of $\mathcal{TT}_{ij}$ let us choose one node n_k corresponding to a boundary triangle $\triangle_k$: we then attach a leaf to n_k in $\mathcal{B}_2$. Set the weight of this leaf equal to $\|\mathcal{B}_k\|$.

As result we obtain a binary tree of size $\|\mathcal{TT}_{ij}\| + s$, with total weight $\|\mathcal{ST}_i\|$, where internal nodes have weights equal to 0, and a selection of leaves have weights satisfying Lemma 12.

We can now apply Lemma 12 to $\mathcal{B}_2$ obtaining two sub-trees $\mathcal{B}'_2, \mathcal{B}''_2$ whose weights are at most $\frac{2}{3}w(\mathcal{B}_2)$.

We get a decomposition of $\mathcal{TT}_{ij}$ into two new sub-triangulations $\mathcal{TT}_{ij'}$ and $\mathcal{TT}_{ij'}$ which does induce a decomposition of $\mathcal{ST}_i$ into two small triangulations $\mathcal{ST}', \mathcal{ST}''$, containing each at most $\frac{2}{3}\|\mathcal{ST}_i\|$ tiny triangulations.

It suffices to add $\mathcal{TT}_{ij'}$ (resp.$\mathcal{TT}_{ij''}$) to the union of adjacent tiny triangulations, which correspond to the nodes of $\mathcal{B}'$ (resp. to the nodes of $\mathcal{B}''$).

As result of the previous steps we get the decomposition of $\mathcal{ST}_i$ into 2 sub-triangulations $\mathcal{ST}', \mathcal{ST}''$: they are valid small triangulations since they define a partition of the triangles of $\mathcal{ST}_i$. Observe that $\mathcal{ST}'$ is a connected sub-region of $\mathcal{ST}_i$ for the way we used to decompose $\mathcal{B}_2$.

To conclude this decomposition phase we observe that if one of the new tiny triangulations created, say for example $\mathcal{TT}_{ij'}$, has not valid size ($< \frac{1}{12}\lg m$ triangles) then it suffices to fusion it with an adjacent tiny triangulation within $\mathcal{ST}'$.

Their union could contain up to $(\frac{1}{4} + \frac{1}{12})\lg m \leq \frac{1}{3}\lg m$ triangles: it suffices to perform a *tiny split* to obtain two valid tiny triangulations having each between $\frac{1}{12}\lg m$ and $\frac{2}{9}\lg m$ triangles.

Concerning the updating of our structure we have to create two new small triangulations $\mathcal{ST}', \mathcal{ST}''$ whose constructions take $O(\lg^2 m)$ amortized time, as the deletion of the old $\mathcal{ST}_i$. (all arrays involved are of size $O(\lg^2 m)$).

Our decomposition phase allows to claim that only a negligible number of new multiple vertices (shared by several tiny triangulations) have been created, as well as the number of new references between small triangulations that did change: updating all these references takes at most $O(\lg m)$ time.

Since only $O(1)$ sides of tiny triangulations do change, updating all the references to S', S'' coming from adjacent tiny triangulations takes $O(\lg^2 m)$ amortized time (at most $O(\lg m)$ nodes in graph G change their degree).

Also updating all the labelings of arcs in graph G requires at most $O(\lg^2 m)$ time for the same reason.

Finally updating pointers in tables S_i takes $O(\lg m)$ amortized time since only a constant number of neighbors of $\mathcal{ST}_i$ do change (when a memory relocation is needed in a table S_i, only $O(\lg m)$ references have to be updated in adjacent small triangulations).

□

8 Update of the triangulation

8.1 Insertion of a new vertex

The insertion of a new vertex always affects the interior of a tiny triangulation: this means that a local modification of the topology and of the adjacency relations is needed only when the size of the new tiny triangulations does not fit between $\frac{1}{12}\lg m$ and $\frac{1}{4}\lg m$.

If the size, after the insertion of a new vertex, does not exceeds the upper bound of $\frac{1}{4}\lg m$, then the new triangulation $\mathcal{TT}_{ij'}$ is a valid tiny triangulation.

If size does increase over the limit of a valid tiny triangulation we must perform a local decomposition making a call of the procedure *splitting a tiny triangulation*, which requires $O(\lg m)$ amortized time.

When the insertion of a vertex does cause a split of a tiny triangulation that increases the size of a small triangulation over the upper bound allowed, we have to perform the procedure *splitting a small triangulation*: the decomposition into two small triangulations as well the update of adjacency relations in maps F and G require $O(\lg^2 m)$ time.

Since these events occurs after $\Theta(\lg m)$ (resp. $\Theta(\lg^2 m)$) vertex insertions, we can state the following:

Lemma 14. *Adding a degree 3 vertex in the triangulation requires $O(1)$ amortized time.*

8.2 Deletion of a degree 3 vertex

If the vertex concerned is not a multiple vertex, the updating phase mirrors the procedure for vertex insertion: since there are not topological modifications of maps G_i, we call the tiny split (resp. small split) procedures only when the size of a triangulation does exceeds the maximum number of triangle allowed (resp. the maximum number of tiny triangulations, within a small one, allowed).

When dealing with multiple vertices, the vertex deletion could change the adjacency relations between the 3 adjacent tiny triangulations: in general there not exists better solution than simply joining the tiny triangulations concerned, performing the vertex deletion and calling the splitting procedures (a constant number of times).

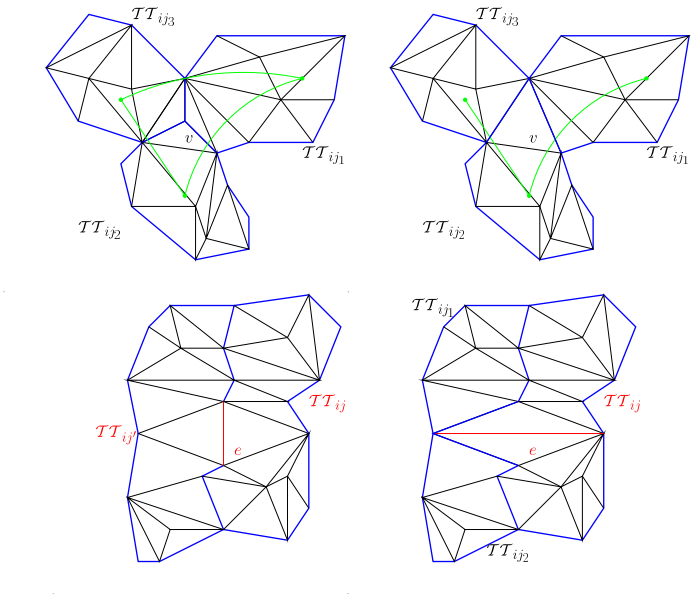

Figure 5: Topological changes in map G_i after a vertex deletion or an edge flip.

Lemma 15. *Deleting a degree 3 vertex from the triangulation needs* $O(\lg^2 m)$ *amortized time.*

8.3 Flipping edges

If the flip concerns an edge (v_1, v_2) that lies in the interior of a tiny triangulation $\mathcal{TT}_{ij}$ we have simply to update pointer G^A_{ij} in collection PA_i. The coloring of the boundary and the adjacencies of $\mathcal{TT}_{ij}$ do not change, and then graph G_i do not need to be updated.

If the edge belongs to the boundary of two tiny triangulations $\mathcal{TT}_{ij}, \mathcal{TT}_{ij'}$ it may occur that the flip does change the topology of a tiny triangulation: for example, if an incident triangle in $\mathcal{TT}_{ij}$ has two edges that are chords, flipping the edge shared by $\mathcal{TT}_{ij}, \mathcal{TT}_{ij'}$ does disconnect $\mathcal{TT}_{ij}$ into two non adjacent tiny triangulations (this event is depicted in Figure 5). As for the vertex deletion, when the topology or the size do change, it only remains to make a fusion of $\mathcal{TT}_{ij}, \mathcal{TT}_{ij'}$, and to call the procedure *splitting a tiny triangulation* (after performing the edge flip). This strategy may require $O(\lg m)$ amortized time if only tiny triangulations in the same small triangulations are concerned. Analogously, if the edge belongs to different small triangulations $\mathcal{ST}_i, \mathcal{ST}_{i'}$ we perform the fusion of $\mathcal{ST}_i$ and $\mathcal{ST}_{i'}$ and the edge flip on the resulting triangulation, then we call the procedure *split a small triangulation*, that updates $\mathcal{T}$ in $O(\lg^2 m)$ amortized time.

Lemma 16. *Performing an edge flip in the triangulation needs $O(\lg^2 m)$ amortized time.*

9 Concluding remarks

9.1 Navigation in the triangulation

The navigation phase mirrors the procedure explained in [2], that we are going to recall briefly.

A triangle t is defined by 3 binary words that represent respectively its number in a tiny triangulation, the index of the tiny triangulation $\mathcal{TT}_{ij}$ to which it belongs (a reference in table T_i), and a pointer to the small triangulation $\mathcal{ST}_i$ containing it.

During the navigation, if we move in the interior of a tiny triangulation $\mathcal{TT}_{ij}$ it simply suffice to access to explicit representation of $\mathcal{TT}_{ij}$ stored in tables A_i, which is done by reading records in arrays T_i and PA_i: the information stored in tables A_i permit local navigation in $O(1)$ time.

If we are lying on the boundary of a tiny triangulation $\mathcal{TT}_{ij}$, we find in tables PB_i its boundary coloring (stored as pointer to tables B_{pq}). We use it to retrieve adjacency information in map G_i. Once we know the color of a side of $\mathcal{TT}_{ij}$ it suffices to find the arc incident to node G_{ij} with the corresponding label. This provides an information concerning the index and the boundary coloring of the adjacent tiny triangulation $\mathcal{TT}_{ij'}$ to which we are moving, together with the rank of the boundary triangle $t' \in \mathcal{TT}_{ij'}$ traversed.

Finally, if we are moving to a different small (and hence tiny) triangulation, we simply repeat the last steps recalling that the index j' does refer now to a tiny triangulation $\mathcal{TT}_{i'j'}$ lying in an adjacent small triangulation $\mathcal{ST}_{i'}$, whose reference is stored in table S_i.

Let us observe that all references and pointers used are on $O(\lg m)$ bits which allows to index in $O(1)$ time. Moreover all tables used to describe our representation permit accessing operations in $O(1)$ time.

9.2 Attaching geometric information

As done in the static case, we can enrich our structure allowing the user to associate geometric data to $\mathcal{T}$: here we assume that one may want to attach $O(\lg m)$ data to vertices or triangles.

The idea is to store this information by associating it to nodes and arcs of map G_i. For this purpose, in the description of the memory organization of a small triangulation $\mathcal{ST}_i$, we introduce a collection of extendible arrays PD_i defined as follows:

• all records in the collection PD_i are of fixed size on $O(\lg m)$ bits and contain the geometric data associated to a given vertex. Data associated with vertices of the same tiny triangulation $\mathcal{TT}_{ij}$ are stored consecutively and sorted reflecting the order on the vertices of $\mathcal{TT}_{ij}$. As done for tables PE_i, records are regrouped: the kth array in the collection has groups of $k\sqrt{\lg m}$ records, corresponding to tiny triangulations having between $(k-1)\sqrt{\lg m}$ and $k\sqrt{\lg m}$ vertices.

In the collection PD_i we store all internal points (in the interior of a tiny triangulation) and a selection of boundary points, in such a way that vertices shared by two tiny triangulations have their associated data stored only once.

One crucial point is about multiple vertices shared by several tiny triangulations: we may stored, for each small triangulation $\mathcal{ST}_i$, the list of multiple vertices with their geometric information using an auxiliary resizable array. Since there are at most $O(\lg^2 m)$ such vertices in $\mathcal{ST}_i$, they can be indexed with references on $O(\lg \lg m)$ bits, which are directly stored in halfarcs of the map G_i: in overall this adds a negligible amount of extra storage. In a similar way we store data associated with multiple vertices shared by several small triangulations (each small triangulation contains data associated to a selection of these vertices).

Increasing of the cost for an update The local modification of the triangulation implies also the update of the above arrays storing the geometric information.

Maintaining the arrays in collection PD_i is an expensive operation, more expensive than updating collections PA_i, and which makes increase the computational cost of an update, even when a vertex insertion is performed.

Lemma 17. *If external data on $O(\lg m)$ bits are associated to vertices of $\mathcal{T}$, the structure can be maintained under vertex insertion in $O(\lg m)$ amortized time. Edge flips and vertex deletions can be performed in $O(\lg^2 m)$ amortized time.*

10 Conclusion

Optimality versus class of triangulations Our storage is optimal for the class of triangulations with boundary, which yields to the 2.175 bits per triangle. Other works often use the class of triangulations whose boundary is a triangle (triangulations of a 3d-sphere) which allows to link strongly the number of triangles and the number of vertices. In such triangulation, the optimum is 3.24 bits per vertex or equivalently 1.62 bits per triangle. Counting in number of bits per vertex in our structure is difficult since the vertices can be shared by several tiny triangulations.

References

[1] L. Castelli Aleardi, O. Devillers, and G. Schaeffer. Compact representation of triangulations. Technical report, INRIA, 2004. RR 5433.

[2] L. Castelli Aleardi, O. Devillers, and G. Schaeffer. Succinct representation of triangulations with a boundary. In *Proc. of WADS*, pages 134–145, 2005. Springer, LNCS 3608.

[3] D. Blanford, G. Blelloch, and I. Kash. Compact representations of separable graphs. In *Proc. of the Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 342–351, 2003.

[4] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. *Comput. Geom. Theory Appl.*, 22:5–19, 2002.

[5] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *WADS*, pages 37–48, 1999.

[6] R.C.-N Chuang, A. Garg, X. He, M.-Y. Kao, and H.-I. Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. *Automata, Laguages and Programming*, pages 118–129, 1998.

[7] R. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *SODA*, pages 1–10, 2004.

[8] G. Jacobson. Space efficients static trees and graphs. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[9] J. I. Munro and V. Raman. Succint representation of balanced parantheses and static trees. *SIAM J. on Computing*, 31:762–776, 2001.

[10] J. I. Munro, V. Raman, and A. J. Storm. Representing dynamic binary trees succintly. In *SODA*, pages 529–536, 2001.

[11] D. Poulalhon and G. Schaeffer. Optimal coding and sampling of triangulations. In *Proc. Intern. Colloqium ICALP'03*, pages 1080–1094, 2003.

[12] R. Raman, V. Raman, and S.S. Rao. Succinct dynamic data structures. In *WADS*, pages 426–437, 2001.

[13] V. Raman and S.S. Rao. Succint dynamic dictionaries and trees. In *ICALP*, pages 357–366, 2003.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399