TopoCluster: A Localized Data Structure for Topology-based Visualization

Guoxi Liu, Federico Iuricich, Riccardo Fellegara, and Leila De Floriani

Abstract—Unstructured data are collections of points with irregular topology, often represented through simplicial meshes, such as triangle and tetrahedral meshes. Whenever possible such representations are avoided in visualization since they are computationally demanding if compared with regular grids. In this work, we aim at simplifying the encoding and processing of simplicial meshes. The paper proposes *TopoCluster*, a new localized data structure for tetrahedral meshes. TopoCluster provides efficient computation of the connectivity of the mesh elements with a low memory footprint. The key idea of TopoCluster is to subdivide the simplicial mesh into clusters. Then, the connectivity information is computed locally for each cluster and discarded when it is no longer needed. We define two instances of TopoCluster. The first instance prioritizes time efficiency and provides only a modest savings in memory, while the second instance drastically reduces memory consumption up to an order of magnitude with respect to comparable data structures. Thanks to the simple interface provided by TopoCluster, we have been able to integrate both data structures into the existing Topological Toolkit (TTK) framework. As a result, users can run any plugin of TTK using TopoCluster without changing a single line of code.

Index Terms—Data visualization, data structures, topological data analysis, simplicial meshes, tetrahedral meshes

1 Introduction

PROCESSING irregularly distributed data has always posed challenges in scientific visualization. Most tools (e.g., Paraview [1], VisIt [5], or Inviwo [21]) prioritize the analysis of regularly distributed data (i.e., 2D and 3D images), whose encoding is both simple and efficient. The same tools present relevant overheads when analyzing irregularly distributed data that require more involved data structures to be encoded.

This work focuses on data defined on tetrahedral meshes and aims at simplifying the processing and encoding of such data. Specifically, our goal is to define a data structure that is both compact and easy to integrate into existing visualization tools. We tackle this problem by introducing a new data structure called *TopoCluster*. TopoCluster partitions a simplicial complex into clusters and processes its simplices with a two-level technique. At the global level, only the minimum amount of information is stored. At the local level, the full information is extracted within each cluster and discarded when no longer needed. The result is a data structure capable of self-adjusting its memory consumption at run time.

The main contributions of this work are two instances of TopoCluster designed with opposite intents. While one instance prioritizes time performance, the second instance focuses on reducing the memory footprint with a consequent loss in time efficiency. Both data structures are designed to easily adapt to existing frameworks for mesh processing and visualization. To prove their flexibility, we have integrated our data structures into the Topological Toolkit (TTK) [30]. Such integration is transparent to a user or a developer. That is, TTK plugins can be executed either by using the original data structure provided by TTK, or our proposed structures, without changing a single line of code.

The structure of the paper is organized as follows. In Sections 2 and 3, we present background notions and related work, respectively. In Section 4, we provide an overview of the main features of TopoCluster, while, in Section 5 and Section 6, we describe two instances of TopoCluster. In Section 7, we describe performance optimization strategies used in both data structures. In Section 8, we discuss our experimental setup and compare the performance of TopoCluster against its natural competitors. Finally, in Section 9, we conclude the paper with some remarks and directions for future works.

- *G. Liu and F. Iuricich are with School of Computing, Clemson University, Clemson, SC, 29631. E-mail: {guoxil, fiurici}@clemson.edu.*
- *R. Fellegara is with Institute for Software Technology, German Aerospace Center (DLR), Braunschweig, Germany. E-mail: riccardo.fellegara@dlr.de.*
- *L. Floriani is with University of Maryland, College Park, MD, 20742. E-mail: deflo@umd.edu.*

Manuscript received ; revised .

2 Background

A *simplex* of dimension k, k-simplex for short, is defined as the convex hull of $k+1$ linearly independent points in the Euclidean space. A k-simplex σ is a *(proper) face* of an m-simplex τ, with $k < m$, if σ is a proper subset of τ. In this case, τ is said to be a *coface* of σ. A simplex which is not the proper face of any other simplex in Σ is called *top simplex*. The set of cofaces of a simplex σ forms the *star* of σ.

A simplicial complex Σ is a collection of simplices such that every face of a simplex σ is also in Σ, and the intersection of any two simplices σ and τ is either a face of both, or it is empty. The dimension d of Σ is the largest dimension of its simplices. Even if a simplicial complex can be defined in any dimension, we focus on its 3D instances, called *tetrahedral meshes*.

2.1 Topological relations

In a simplicial complex, simplices are involved in *topological relations*. A boundary relation maps a simplex to its faces, for instance, σ is on the boundary of τ iff σ is a face of τ. Vice versa, τ is said to be on the *coboundary* of σ. Two k-simplices τ_1 and τ_2 are said to be *adjacent* if they share a $(k-1)$-simplex on their boundaries. Informally, we say that two 0-simplices (vertices)

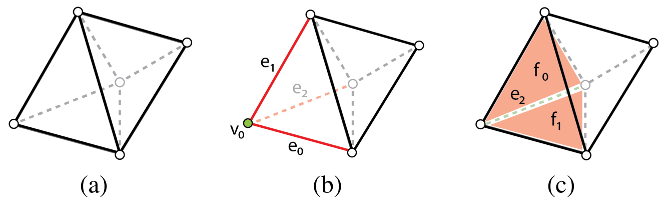


Fig. 1. (a) Tetrahedral mesh composed by two tetrahedra sharing a triangle. (b) VE relational operator for the vertex v_0 (c) EF relational operator for the edge e_2.

are adjacent if they share the same 1-simplex (edge) in their coboundaries. A *relational operator* associates a simplex σ to a set of simplices having a specific topological relation with σ.

In the remainder of this paper, we only consider relational operators for tetrahedral meshes. We use capital letters to indicate whether the operator involves vertices (V), edges (E), triangles (F), or tetrahedra (T), and each operator is specified with a pair of letters. For example, EF indicates the relational operator associating an edge (E) to the triangles (F) on its coboundary. On a tetrahedral mesh we have six boundary operators (EV, FV, TV, FE, TE, TF), six coboundary operators (VE, VF, VT, EF, ET, FT), and four adjacency operators (VV, EE, FF, TT). Figure 1 shows two examples of relational operators. Figure 1(b) shows operator $VE(v_0) = \{e_0, e_1, e_2\}$ representing the edges e_0, e_1 and e_2 which are in the coboundary of vertex v_0. Figure 1(c) shows operator $EF(e_2) = \{f_0, f_1\}$ representing the triangles f_0 and f_1 which are on the coboundary of edge e_2.

3 Related Work

Generally speaking, data structures differ in the type of simplices and relational operators they encode. Data structures described in Section 3.1 are *static* (as opposed to *dynamic* structures) where relational operators are computed for the entire mesh in a preprocessing step. In Section 3.2, we describe the Stellar decomposition, a model for dynamic structures where relational operators are computed and discarded at runtime.

3.1 Static data structures

There has been extensive research on topological data structures for simplicial complexes, especially for triangle and tetrahedral meshes [8].

The *incidence graph* [11] encodes explicitly all simplices plus all boundary and coboundary operators, which makes it the most general data structure for simplicial complexes. Multiple data structures have been defined to reduce the extremely large memory requirements of the incidence graph, either by cutting down the number of relational operators or by limiting the simplices encoded. The *Simplex tree* [2] is a variant of the incidence graph which organizes all simplices in a trie [14] and avoids encoding boundary operators. When data size increases, representing all simplices is no longer feasible. For this reason, alternative representations have been designed to prevent encoding simplices of specific dimensions. The *half-edge* [23] is a well-known data structure for triangle meshes, which drastically reduces memory consumption by encoding only the relational operators involving edges.

More recently, compact representations have been developed to maintain the same expressive power of the incidence graph while halving the space required [3], [7], [9]. The novelty of these data structures relies in the encoding of adjacency operators, instead of the more expensive coboundary operators. Examples include the *Indexed data structure with Adjacencies* [24], [25], the *Corner-Table* data structure [27] and its several extensions proposed specifically for triangle meshes [16], [22] and tetrahedral meshes [17]. The *generalized indexed data structure with adjacencies* (IA^*) [4] is the first data structure extending this approach to non-manifold simplicial complexes of arbitrary dimension. Among static data structures for non-manifold simplicial complexes, the IA^* is the most compact [15].

3.2 Stellar decomposition

The Stellar decomposition [12] represents a family of data structures in which relational operators are computed and discarded, at runtime, based on user requests. For this reason, they are called *dynamic* as opposed to the static data structures discussed in Section 3.1. The *Stellar decomposition* is based on three elements:

- an input simplicial complex Σ (represented by an indexed mesh representation encoding the vertices and the top simplices),
- a subdivision Δ of the vertices of Σ into clusters,
- a map associating elements of Σ to clusters of Δ.

The simplicial complex is processed with a *localized approach*. Instead of extracting relational operators altogether in a preprocessing step, the localized approach extracts operators, inside each cluster, at runtime. Given a k-simplex σ and a relational operator o, the simplices in relation with σ (i.e., $o(\sigma)$) will be extracted as follows:

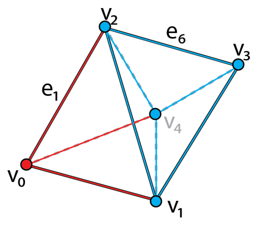
(i) locate the cluster c of Δ containing σ;
(ii) compute the relational operator o for all the k-simplices contained in c;
(iii) return the set of simplices in relation with σ (i.e., $o(\sigma)$);
(iv) discard (delete) o.

The first data structure implementing this model was the *PR-star octree* [32], which was explicitly defined for tetrahedral meshes embedded in $\mathbb{R}^3$. Successively, this has been generalized by the *Stellar tree* [12], which can encode simplicial complexes embedded in any dimension and with arbitrary domain. The Stellar tree uses a hierarchical decomposition $\mathbb{H}$ (an n-dimensional bucketed Point Region quadtree [28]) to organize the mesh vertices. Relational operators are extracted locally to the leaf nodes of such hierarchy, following the Stellar decomposition model. As a result, the Stellar tree is even more compact than adjacent-based data structures like the IA* data structure [4]. On the other hand, simplices in a Stellar tree can only be accessed through a visit of the hierarchy $\mathbb{H}$, which introduces an additional layer of complexity for the developer (see Appendix A). As a consequence, algorithms need to be re-designed to adapt to the data structure processing model.

Our work aims to maintain the low memory footprint of the Stellar tree while providing an easy interface for implementing and running topological algorithms.

4 TopoCluster

The goal of all data structures for simplicial complexes is that of providing easy access to the relational operators. The proposed data structure, called *TopoCluster*, inherits the localized approach for extracting relational operators from the Stellar decomposition. Different from the Stellar decomposition, it aims at enumerating all the simplices of the simplicial complex Σ through an *enumeration schema*. An explicit enumeration of the simplices of Σ provides



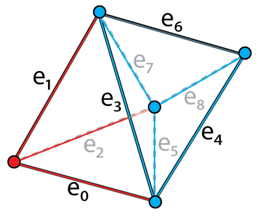
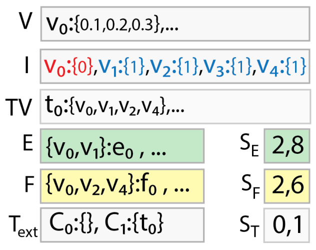


Fig. 2. Tetrahedral mesh Σ formed by two clusters. (a) Subdivision of vertices and edges across two clusters depicted with red and blue colors. (b) Enumeration of the edges of Σ.



V	v0:{0.1,0.2,0.3},...		
I	v0:{0},v1:{1},v2:{1},v3:{1},v4:{1}		
TV	t0:{v0,v1,v2,v4},...		
E	{v0,v1}:e0 , ...	S_E	2,8
F	{v0,v2,v4}:f0 , ...	S_F	2,6
T_{ext}	C0:{}, C1:{t0}	S_T	0,1

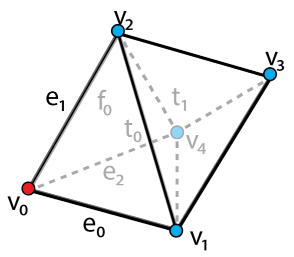


Fig. 3. Table on the left shows the global layer of Explicit TopoCluster for the tetrahedral mesh on the right. V encodes the coordinates of each vertex, I stores the cluster index of each vertex, TV stores the boundary vertices of each tetrahedron, E and F are hash maps encoding indices for edges and triangles respectively, T_{ext} stores indices of external tetrahedra, S_E, S_F, and S_T are arrays storing the enumeration intervals for edges, triangles and tetrahedra respectively.

multiple benefits from a developer perspective. A practical example is shown in Appendix A.

In the following, we describe the enumeration schema used by TopoCluster. In the remainder of this paper, σ_i indicates a simplex σ based on its index i; $\bar{\sigma}$ indicates a simplex σ based on its vertices $\{v_0,...,v_k\}$.

Cluster-based enumeration. Given any subdivision Δ that divides the vertices of the simplicial complex Σ into clusters, we define an enumeration schema by assigning each k-simplex to a single cluster. We assume that each vertex v is associated to a single cluster c. We say that v is *internal* to c, and c *contains* v.

For edges, triangles, and tetrahedra we define a k-simplex, with $k > 0$, *internal* to a cluster c as follows.

Definition 4.1. Without loss of generality, we assume a total order on the clusters of Δ. Given a cluster $c \in \Delta$ and a k-simplex $\sigma \in \Sigma$, with $0 < k \leq d$, σ is *internal* to c iff. c is the first cluster containing a vertex of σ.

In Figure 2(a), vertices and edges are depicted with the same color if they belong to the same cluster. In this example, we assume that the red cluster is the first cluster (i.e., c_0) and the blue cluster is the second cluster (i.e., c_1). Thus, vertex v_0 is internal to cluster c_0, while vertex v_2 is internal to cluster c_1. Edge e_1 is internal to c_0 since one of its vertices (i.e., v_0), is internal to c_0. Following the same assumption, edge e_6 is internal to c_1 since both its vertices are internal to c_1.

The *cluster-based enumeration* is obtained by enforcing the following rules:

- k-simplices internal to a cluster c are enumerated within a closed interval $[l,u]$, where $u-l+1$ is the number of k-simplices internal to c;
- For any pair of clusters, the corresponding intervals do not overlap. As a consequence, for any pair of clusters c_i, c_j, with $i < j$, k-simplices in c_j have indices greater than those in c_i.

The result is an explicit enumeration of the simplices of Σ, where each simplex is associated with a unique integer. Figure 2(b) shows the cluster-based enumeration for the edges of a tetrahedral mesh. Edges internal to the red cluster (i.e., c_0), are enumerated from 0 to 2. Edges internal to the blue cluster (i.e., c_1), are enumerated from 3 to 8.

Once defined the enumeration schema, we describe how such enumeration is encoded in the data structure. To this end, we have designed two strategies. The first strategy, named *Explicit*, prioritizes the time efficiency (see Section 5), while the second strategy, named *Implicit*, prioritizes the memory efficiency (see Section 6).

5 Explicit TopoCluster

The first approach for encoding the enumeration schema is that of explicitly storing the index associated with each simplex of Σ. This is the strategy implemented by the first data structure introduced in this paper called *Explicit TopoCluster*. We recall that the idea behind TopoCluster is that of computing relational operators at runtime. To allow for this interaction, Explicit TopoCluster organizes information into two layers: the *global*, and the *local* layer.

The global layer, described in Section 5.1, is the static part of Explicit TopoCluster. The local layer, described in Section 5.2, is the dynamic part of Explicit TopoCluster. This layer is where topological relations are computed, stored for a short period, and then discarded to keep memory usage under control.

5.1 Global layer

The global layer of Explicit TopoCluster includes the input tetrahedral mesh Σ, the input subdivision Δ, the enumeration schema, and the list of simplices intersecting each cluster defined in Δ.

Tetrahedral mesh. Mesh Σ is represented through an indexed representation, in which the vertices and tetrahedra are encoded in two arrays, V and TV, respectively. V encodes the coordinates of each vertex, while TV stores the boundary vertices of each tetrahedron (i.e., TV operator). For example, as shown in Figure 3, vertex v_0 has coordinates $(0.1, 0.2, 0.3)$, and tetrahedron t_0 has vertices v_0, v_1, v_2 and v_4 on its boundary.

Clustering. The subdivision Δ is encoded with an array I, storing the cluster index of each vertex v in the simplicial mesh. For example, in Figure 3, vertex $I[v_0] = 0$ since it belongs to cluster c_0. The value of I for any other vertex is equal to 1 since they all belong to cluster c_1.

Enumeration. As described in Section 4, the enumeration assigns a unique integer to each simplex of Σ. The enumeration of vertices and tetrahedra is defined by their order in the arrays V and TV. The enumeration of edges and triangles is encoded by two hash tables, E and F, respectively. Table E associates each edge (represented with a pair of vertex indices) with the corresponding index. Similarly, table F encodes the index associated with each triangle. Table E and F are defined using the hash table implementation provided by Boost library [29]. In Figure 3, edge e_0 is formed by vertices v_0 and v_1. Similarly, triangle f_0 is formed by vertices v_0, v_2 and v_4.

Internal and external simplices. Finally, we need to encode how simplices are distributed across the clusters of Δ. Specifically,

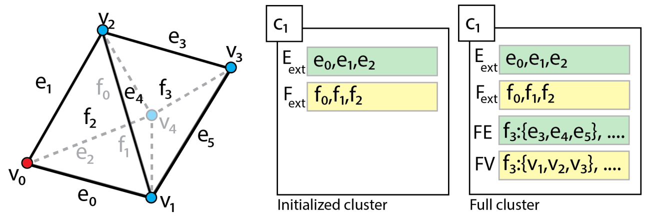


Fig. 4. The local layer of Explicit TopoCluster. An initialized cluster c_1 contains only external edges and triangles denoted by E_{ext} and F_{ext} respectively. The full cluster stores two additional arrays containing FE and FV relational operators.

for each cluster c, we encode the simplices internal to c and the tetrahedra intersecting c that are internal to some other cluster. This information provides the full connectivity of the simplicial complex and will be used to compute relational operators (see Section 5.2).

To retrieve the simplices internal to each cluster c, the number of tetrahedra, triangles, and edges encoded in c are stored in three global arrays named S_T, S_F, and S_E respectively. As mentioned in Section 4, a cluster c_i contains tetrahedra with index in the interval $[u,l]$. This interval is stored in the array S_T (i.e., $[S_T[i-1]+1, S_T[i]]$). In a similar fashion, indices of triangles and edges internal to c_i are retrieved using S_F and S_E, respectively.

Finally, an indexed array T_{ext} stores the list of external tetrahedra for each cluster. As shown in Figure 3, cluster c_0 has no external tetrahedra as the internal tetrahedron t_0 is the only one in the boundary of vertex v_0. Cluster c_1 has t_0 as external tetrahedron.

5.1.1 *Initializing global structures*

All information in the global layer are either received as input, or computed at initialization time. Vertex coordinates (i.e., array V), TV operators (i.e., array TV), and clustering function I are provided as input. Vertices and tetrahedra are reindexed in order to conform with the enumeration property. In practice, this means assigning contiguous indices to vertices(/tetrahedra) contained in the same cluster which takes $O(|V|+|T|)$ time. The array of external tetrahedra T_{ext} and array S_T are populated in $O(|T|)$ time by iterating over the array TV.

Hash tables E and F are initialized by visiting the clusters in any order. For each cluster c, internal edges and triangles are enumerated by checking the list of tetrahedra intersecting c. Since each internal tetrahedron is visited exactly once, and each external tetrahedron is visited at most four times, hash tables E and F are computed in $O(|T|)$. Arrays S_E and S_F are initialized during the same step with no additional cost.

Encoding the input mesh requires $O(|V|+|T|)$ memory. This cost includes the coordinate values of the each point and the TV operator of each tetrahedron. The size of S_E, S_F, and S_T arrays have size linear in the number of clusters (i.e., $O(|C|)$). The size of the hash maps E and F are determined by the number of edges and triangles in the mesh. Then, the global layer requires $O(|V|+|E|+|F|+|T|+|C|)$ memory.

5.2 Local layer: clusters

The local layer is where relational operators are computed and stored. Once a relational operator for a simplex σ is required, TopoCluster locates the cluster c_i containing σ, computes the relational operators of the simplices internal to c_i, and returns the relational operator of σ.

Algorithm 1 computeVT(c)

```
1: Input: $c$, cluster
2: Output: $VT$, tetrahedra incident in each vertex of cluster $c$
3:
4: $VT = \{\}$    // create empty table
5: for each tetrahedron $t_i$ intersecting $c$    // both internal and external do
6:     for each $v_i$ in $TV[t_i]$ do
7:         if $v_i$ internal to $c$ then
8:             $VT[v_i] \leftarrow t_i$    // save $t_i$ in the list associate to $v_i$
9:         end if
10:    end for
11: end for
12: return $VT$
```

The cluster c_i is considered to be *empty* until a new relational operator is requested. Upon request, the cluster is *initialized* by retrieving the information necessary to compute the relational operator. After a relational operator is computed and stored in c_i, we refer to c_i as *full*.

5.2.1 *Initializing clusters*

Initializing the cluster c_i means computing the list of internal and external simplices for c_i. Internal simplices are deduced from the arrays S_T, S_F, and S_E. The list of external tetrahedra intersecting the cluster is encoded in the global layer, specifically the array $T_{ext}[i]$. Upon initialization, the cluster c_i creates two arrays, E_{ext} and F_{ext}, encoding the list of external edges and triangles of c. Array E_{ext} is computed by cycling on the list of external tetrahedra $T_{ext}[i]$. For each external tetrahedron $\bar{t} = \{v_1, v_2, v_3, v_4\}$, for each pair of vertices $\bar{e} = \{v_j, v_k\}$ such that $\{v_j, v_k\} \in \bar{t}$ and $v_j \neq v_k$, the index e_j is retrieved from the global hash map E (i.e., $e_j = E(\bar{e})$) and added to E_{ext}. Array F_{ext} is built in a similar fashion by considering triples of vertices for each tetrahedron.

When both E_{ext} and F_{ext} are computed, c_i is said *initialized*. Since the number of vertices per tetrahedron is constant, the initialization of cluster c_i requires $O(|T_{ext}[i]|)$ time. Figure 4 shows the information encoded in c_1 after the initialization step. Indices of external edges e_0, e_1 and e_2 are listed in E_{ext}, and, similarly, external triangles f_0, f_1 and f_2 are listed in F_{ext}.

5.2.2 *Computing relational operators*

Relational operators for a cluster c_i are computed only after the cluster is initialized. In the following, we describe as an example the extraction of relational operators VT and FE.

VT operator represents the set of tetrahedra incidents in each vertex. Algorithm 1 describes the steps performed in extracting such an operator. The only information required by the algorithm is the list of tetrahedra intersecting c (line 5). These are computed during the initialization step and are the indices of internal tetrahedra, encoded in S_T, and the external tetrahedra, encoded in T_{ext}. For each tetrahedron t_i, the list of vertices is retrieved using the TV operator (line 6). Then, for each vertex v_i internal to c, t_i is associated to the list of co-faces of v_i (line 8).

Algorithm 2 describes the extraction of the FE operator. This operator encodes, for each triangle, the indices of the three edges in its boundary. Since this operator involves edges and triangles, it also requires the local FV operator that computes for each internal triangle f_i, its list of vertices (line 4). For each pair of such vertices (line 9), we retrieve the index e_i of the corresponding edge using

Algorithm 2 computeFE(c)

```
1:  Input: $c$, cluster
2:  Output: $FE$, edges of each triangle of cluster $c$
3:
4:  $FV \leftarrow$ computeFV($c$)    // retrieve local information
5:  $FE = \{\}$    // create empty table
6:  for each internal triangle $f_i$ in $c$ do
7:      for each pair of vertices $\bar{e} = \{v_i, v_j\}$ in $FV[f_i]$ do
8:          $e_i$ = E($\bar{e}$)    // retrieve edge index
9:          $FE[f_i] \leftarrow e_i$
10:     end for
11: end for
12: return $FE$
```

the global hash table E (line 10). Finally, e_i is added to the set of edges on the boundary of f_i (line 11).

Figure 4 shows an example of the *full* cluster c_1. The FV array encodes, for each internal triangle, the index of its three vertices. The indexed array FE stores, for each triangle, the list of boundary edges. The time complexity of extracting a relational operator in the explicit data structure is linear to the number of higher dimensional simplices involved. For example, the complexity of extracting the FE operator is linear to the number of triangles in the cluster c, i.e., $O(|F_c|)$.

6 Implicit TopoCluster

Explicit TopoCluster fully encodes the enumeration of edges and triangles with the two hash tables E and F. We defined a second data structure, called *Implicit TopoCluster*, implementing a different strategy. Instead of encoding the two hash tables E and F in the global layer, the indexing of edges and triangles is computed on-the-fly when accessing a cluster. This drastically reduces the total cost of the global layer to $O(|V|+|T|+|C|)$.

In the following, we describe the local layer of Implicit TopoCluster since the initialization of the global layer is as in the Explicit TopoCluster (see Section 5).

6.1 Local layer: clusters

The local layer of Implicit TopoCluster resembles that of Explicit TopoCluster. Also in this case, a cluster c_i is said *empty* until a new relational operator is requested. Upon request, the cluster is *initialized* by retrieving the information necessary to compute the relational operator. After a relational operator is computed and stored in c_i, we refer to c_i as *full.*

6.1.1 Initializing clusters

Upon initialization, cluster c_i computes the hash tables of external edges and triangles (i.e., E_{ext} and F_{ext}) as in the Explicit TopoCluster. Additionally, two hash tables are computed associating the enumeration of edges and triangles to their vertices (i.e., hash tables E and F). The latter encodes the same information of the hash tables used by Explicit TopoCluster, but instead of being stored globally, these are stored locally to c_i and encode information limitedly to edges and triangles internal to c_i.

Figure 5 shows an example of the information encoded in cluster c_1. Hash tables E and F encode the enumeration of edges and triangles, respectively, and E_{ext} and F_{ext} arrays encode the list of external edges and triangles, respectively.

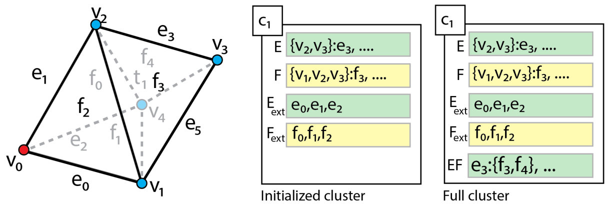


Fig. 5. The local layer of Implicit TopoCluster. An initialized cluster c_1 contains the hash maps for edges and triangles denoted by E and F respectively, and external edges and triangles denoted by E_{ext} and F_{ext} respectively. The full cluster stores and additional array containing the EF relational operator.

All structures are generated by iterating the list of tetrahedra intersecting c_i. First, we create hash tables E and F by computing a local enumeration of the internal edges and internal triangles defined over a closed interval $[0,p]$, where p is either the total number of internal edges or the total number of internal triangles. Global indices for the edges are obtained by shifting the local enumeration according to the global enumeration, i.e., $[S_E[i-1]+1, S_E[i]] = [l,u]$. If an edge has local index j, its global index is $j+l$. Global indices for triangles are retrieved in a similar way. The time complexity for computing the local enumeration is $O(|T_{int}|)$, where $|T_{int}| = S_T[i] - S_T[i-1]$ is the number of internal tetrahedra of c_i.

External edges and triangles are retrieved by iterating the list of external tetrahedra $T_{ext}[i]$, similarly to the Explicit TopoCluster. The difference is that the index of each external simplex is no longer provided by global hash maps. To get the index corresponding to an external edge or triangle, we have to access the cluster c_j containing it and compute the internal simplices of c_j. This step requires $O(\sum_{j=0}^{n} |T_{int}^{j}|)$ time, where $|T_{int}^{j}|$ indicates the tetrahedra internal to the cluster c_j, sharing a simplex with cluster c_i. Hence, initializing the cluster c_i requires $O(|T_{int}| + \sum_{j=0}^{n} |T_{int}^{j}|)$ time in total.

6.1.2 Computing relational operators

The strategy for computing relational operators for the Implicit TopoCluster is similar to the Explicit TopoCluster. For example, let us consider the extraction of EF operator (as detailed in Algorithm 3). We recall that the EF operator encodes, for each internal edge, the indices of the triangles in its star. In the initialization phase, internal edges and triangles are enumerated to create the local hash tables E and F, while the local hash map of external triangles F_{ext} is obtained by querying external clusters sharing a triangle with c (line 5). At this point, the extraction of the relational operators begins. The function requires visiting both internal and external triangles (line 7). For each pair of vertices $\bar{e}$ of f_i, we check if $\bar{e}$ is internal (line 9). If it is the case, we retrieve the index of $\bar{e}$ from map E and associate f_i to e_i (line 11).

Similar to the Explicit TopoCluster, the time complexity of extracting a relational operator is linear to the number of higher dimensional simplices involved. For example, the complexity of extracting the EF operator is linear to the number of triangles in the cluster, i.e., $O(|F \cup F_{ext}|)$.

7 Performance Optimization strategies

To optimize memory and time performance, we have defined two strategies: a preconditioning approach, and a cache system.

Algorithm 3 computeEF(c)

```
1:  Input: $c$, cluster
2:  Output: $EF$, triangles incident in each edge of cluster $c$
3:
4:  // Initialize the cluster c
5:  $E$, $F$, $F_{ext}$ ← initialize($c$)
6:  $EF = \{\}$    // create empty table
7:  for each triangle $\bar{f}$ in $(F \cup F_{ext})$    // both internal and external do
8:      for each pair of vertices $\bar{e} = \{v_i, v_j\}$ of $\bar{f}$ do
9:          if $\bar{e}$ is internal to $c$ then
10:             if $\bar{f}$ is internal to $c$ then
11:                 $f_i = F(\bar{f})$
12:             else
13:                 $f_i = F_{ext}(\bar{f})$
14:             end if
15:             $EF[E[\bar{e}]] \leftarrow f_i$
16:         end if
17:     end for
18: end for
19: return $EF$
```

The first strategy adopts the same preconditioning approach used by TTK [30]. A developer declares the set of relational operators required by an algorithm. Then, clusters will be initialized only with information for those relational operators. For example, suppose we implement an algorithm using only VV and TV operators. The data structure will never enumerate edges and triangles at generation time and it will never compute the associated structures when initializing a cluster. In practice, all structures depicted in green (for edges) and yellow (for triangles) in Figures 3, 4 and 5, are subject to the preconditioning system. This strategy enhances the time and memory efficiency when an algorithm requires only a limited subset of relational operators.

Both Explicit and Implicit TopoCluster use the same approach for computing relational operators. Specifically, they compute and discard information each time a cluster is accessed. This introduces a clear drawback when a cluster is accessed multiple times. To tackle this problem, we have defined a second technique inspired by the Stellar tree [12]. This strategy defines a cache system for the clusters based on the Least Recently Used (LRU) replacement strategy. Each time a full cluster c_i is computed, it is saved in the cache. The cluster in the cache will be replaced based on the last time it was accessed. Since the cache size (i.e., the maximum number of clusters that the cache can maintain) is controlled by a user-defined parameter, the memory requirements cannot be estimated theoretically, but we provide an experimental analysis in Section 8.3.

8 Evaluation of performance

Explicit and Implicit TopoClusters have been implemented as two modules of the Topology Toolkit (TTK version 0.9.7) [30], and use the same interface as the *abstractTriangulation* class of TTK. As a result, all modules implemented in TTK can run seamlessly with TopoCluster.

We recall that TopoCluster requires a clustering for the vertices of the tetrahedral mesh to be provided in input. In the following evaluation, we use a clustering technique based on the Point Region (PR) octree [28]. An octree uses a hierarchical domain decomposition based on a nested refinement of the unit cube. The containment relationship on such cubes defines a hierarchical relationship among the nodes in the octree. The PR octree is constructed by defining the maximum number of vertices allowed in any leaf node of the octree. In the end, vertices belonging to the same leaf node in the PR octree form a cluster in TopoCluster. We select this clustering approach for its generality since any spatially-embedded mesh can be decomposed into clusters using this subdivision.

TABLE 1
Overview of the experimental datasets. For each dataset, we list the type, the number of vertices $|V|$, edges $|E|$, triangles $|F|$ and tetrahedra $|T|$. *Regular* means the dataset comes from 3D regular grids, while *Irregular* means the dataset comes from a tetrahedral mesh with irregularly distributed points.

Data	Type	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert F\rvert$	$\lvert T\rvert$
Red Sea	Regular	0.95M	6.33M	10.58M	5.20M
Engine	Regular	1.39M	9.14M	15.18M	7.43M
Cat	Irregular	1.97M	13.24M	22.25M	10.99M
Sphere	Irregular	2.62M	17.54M	29.46M	14.53M
Foot	Regular	4.60M	30.79M	51.51M	25.32M
Shapes	Irregular	7.87M	52.37M	87.63M	43.13M
Hole	Irregular	9.26M	63.70M	108.29M	53.85M
Stent	Regular	17.37M	118.79M	201.40M	99.98M

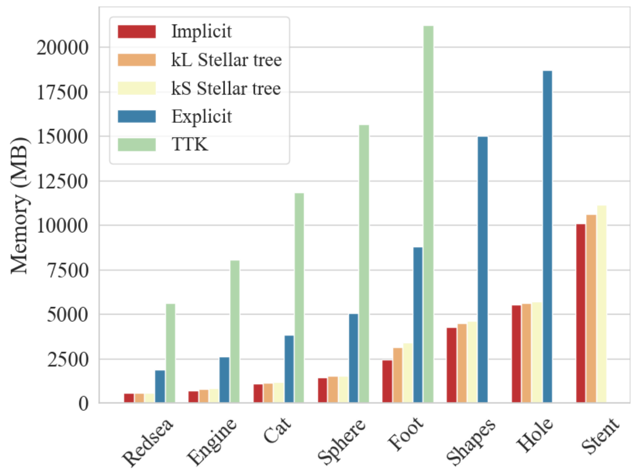
The following performance analysis is conducted on tetrahedral meshes with the number of vertices between 950K and 17M and with number of tetrahedra between 5.2M and 100M (see Table 7). Four datasets (i.e., Red sea [31], Engine, Foot, and Stent) are obtained by thresholding and tetrahedralizing points from 3D regular grids. The remaining four datasets are tetrahedral meshes with irregularly distributed points [20]. All experiments have been performed on a desktop equipped with a 3.2 GHz Intel i7-8700 CPU and 32 Gigabytes of RAM.

8.1 Computing relational operators

In this section, we compare our data structures against the Stellar tree [12] and TTK triangulation [30]. All four data structures use the same encoding for the underlying mesh, that encodes in two arrays the vertex coordinates and the TV operator of each tetrahedron σ, i.e., the list of vertices in the boundary of σ.

TTK Triangulation. TTK triangulation [30] precomputes relational operators at generation time and stores them in multiple lookup tables. Lookup tables, as well as the list of edges and triangles, are extracted in $O(|T|)$ by enumerating all pairs/triplets of vertices. This approach achieves best time performance at runtime since the data structure will provide fast access to all necessary relational operators. At the same time, this strategy is very demanding in terms of memory since relational operators are stored, for the entire execution of the algorithm.

Stellar tree. The Stellar tree is the first data structure defined upon the Stellar decomposition model [12]. It uses a hierarchical decomposition (a Point Region octree [28]) to organize the mesh vertices. The hierarchy $\mathbb{H}$ is encoded through a tree structure used to navigate the mesh. A bucketing threshold is used for limiting the number of vertices per leaf node. In our experiments, we use the bucketing threshold 400 and 800 following the guidelines



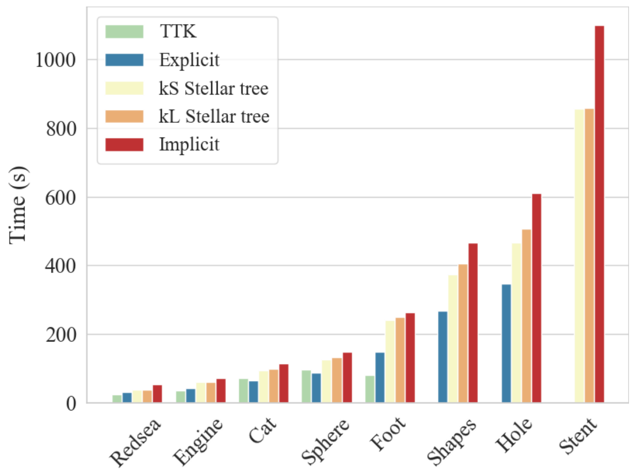


Fig. 6. Memory (in Megabytes) and time (in seconds) required for computing all relation operators with TTK triangulation, Stellar tree, Explicit TopoCluster and Implicit TopoCluster. k_L and k_S indicate a Stellar tree computed with either the larger (800) or smaller (400) bucketing threshold.

from the original paper [12]. Another difference compared with TopoCluster is the internal representation of simplices. The Stellar tree enumerates globally only vertices and tetrahedra while it avoids enumerating edges and triangles and represents such simplices as a tuple of vertices.

We compare the performance of the four data structures for extracting relational operators. We start by computing all relational operators involving vertices. Then, we move to edges, triangles, and tetrahedra. Notice that TopoCluster requires two user-defined parameters. The first parameter is the cache size which defines the maximum number of clusters stored in cache (see Section 7). The second parameter is the cluster size which defines the maximum number of vertices contained in each cluster. The appropriate selection of these parameters is discussed in Section 8.3. For this experiment, we use a fixed cluster size (10000) and a fixed cache size (1% of the number of clusters).

Figure 6 shows the results obtained with the four data structures. As expected, Implicit TopoCluster and the Stellar tree show better scalability. They are the only data structures capable of running on all datasets. TTK triangulation goes out of memory on the larger three datasets, while Explicit TopoCluster goes out of memory on the largest dataset.

Regarding the memory footprint, Explicit TopoCluster provides a good improvement compared to TTK triangulation. Memory usage decreases by three times when using Explicit TopoCluster. Implicit TopoCluster is always the most compact data structure requiring 10% less memory than the Stellar tree.

Considering execution time, Implicit TopoCluster is always the slowest at extracting relational operators. On average, it requires up to 20% time more than the Stellar tree, 70% more time than the Explicit, and it is twice slower than TTK triangulation. TTK triangulation and the Explicit TopoCluster have overall similar performance, even if the latter requires on average 20% more time than TTK triangulation.

Compared to TTK triangulation, the scalability provided by TopoCluster is of practical relevance. Implicit TopoCluster is twice slower than TTK triangulation, but it is also ten times more compact. Explicit TopoCluster is 20% slower than TTK triangulation, but it is also three times more compact.

Compared to the Stellar tree, Implicit TopoCluster is 20% slower, but also 10% more compact. Explicit TopoCluster uses three times the memory than a Stellar tree, but it is also 30% faster. We recall that each simplex is referenced as a unique number in TopoCluster. The generation of such enumeration schema requires more time, which is why TopoCluster is slower than the Stellar tree. Since the enumeration lets us represent each simplex as a single integer, this also explains why Implicit TopoCluster is more compact.

Overall the main advantage of the enumeration strategy implemented in TopoCluster is the easy integration with existing frameworks. In the following sections, we drop the comparison with the Stellar tree, since it does not allow for such integration, and we compare TTK Triangulation and TopoCluster performance by running existing TTK plugins.

8.2 Plugins for topology-based visualization

TTK offers several plugins for topology-based visualization [19]. For the sake of our comparison, we are interested in distinguishing plugins based on how they process the input mesh.

Some plugin visits simplices in a sequential order following the enumeration schema. As a consequence, TopoCluster will access clusters in the same sequential order. We select *TTKScalarFieldCriticalPoints* as an example of a plugin of this kind. Conversely, other plugins visit simplices in a pseudo-random fashion which will force TopoCluster to visit clusters in a random order, possibly initializing the same cluster multiple times. We select *TTKMorseSmaleComplex* as an example of a plugin of this kind.

TTKScalarFieldCriticalPoints. This plugin is used for computing critical points from a given input scalar function. Cluster sizes 5000, and 10000 are chosen for Explicit and Implicit TopoCluster, respectively. Cache size of 1% is selected for both structures.

This plugin requires extracting VV and VT operators. Moreover, VF and FT operators are computed to identify the list of boundary vertices. We recall that a vertex v is on the mesh boundary if at least one of the triangles incident in v has only one tetrahedron on its coboundary.

Figure 7 shows the performance of the three data structures. Implicit TopoCluster uses 60% of the memory required by Explicit TopoCluster, since it stores local hash maps for the triangles instead of a global map. We notice that TTK triangulation requires twice the memory of Implicit structure, and it goes out of memory on the

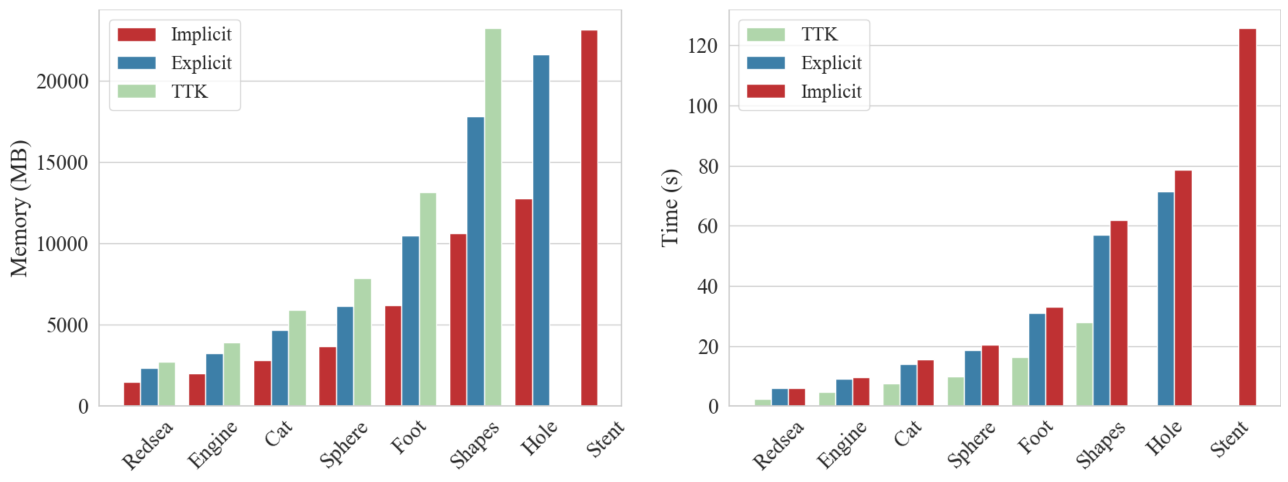


Fig. 7. Memory (in Megabytes) and time (in seconds) required for computing critical points (plugin *ScalarFieldCriticalPoints*) with TTK triangulation (TTK), Explicit TopoCluster (Explicit), and Implicit TopoCluster (Implicit).

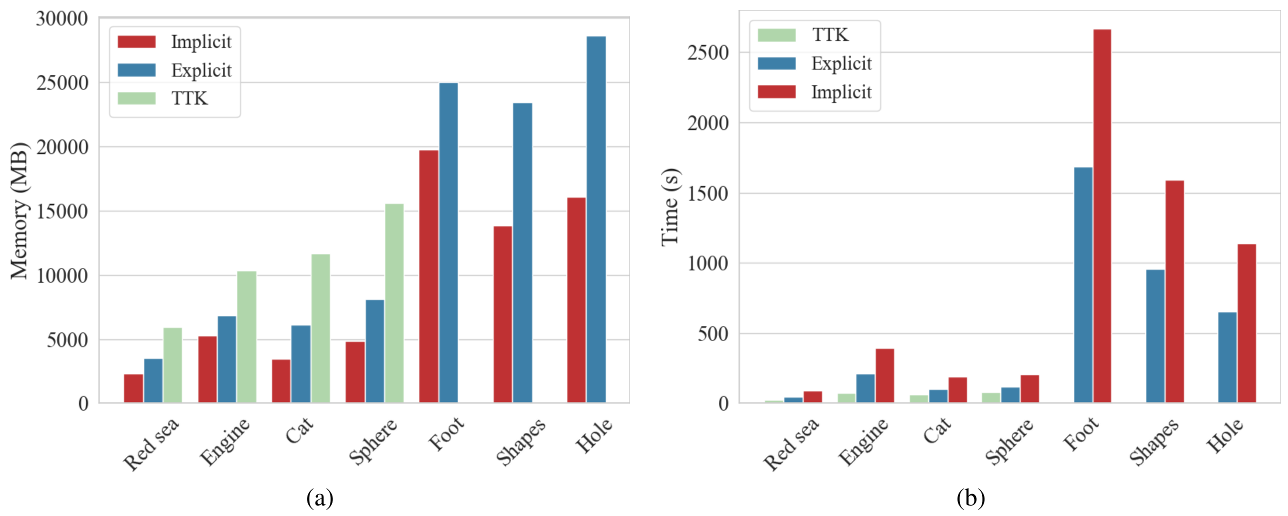


Fig. 8. Memory (in Megabytes) and time (in seconds) required for computing Morse-Smale complex (plugin *MorseSmaleComplex*) with TTK triangulation (TTK), Explicit TopoCluster (Explicit), and Implicit TopoCluster (Implicit).

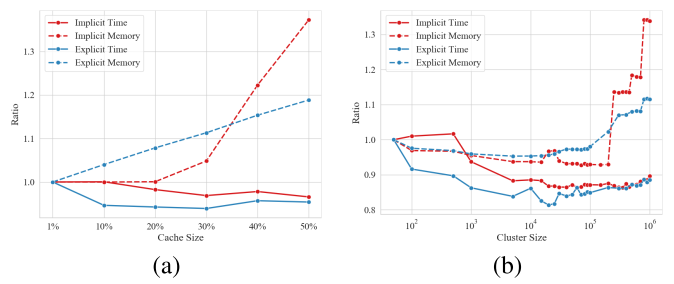


Fig. 9. The changes of memory and time usage on *TTKScalarFieldCriticalPoints* plugin for *Foot* dataset with *Implicit* and *Explicit* TopoCluster when (a) cache rate changes from 1% to 50% and (b) cluster size changes from 10 to 1,000,000.

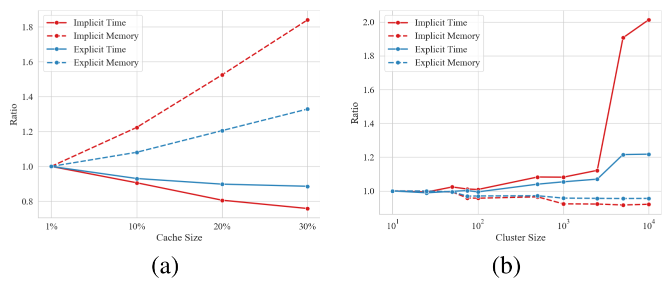


Fig. 10. The changes of memory and time usage on *TTKMorseSmaleComplex* plugin for *Foot* dataset with *Implicit* and *Explicit* TopoCluster when (a) cache rate changes from 1% to 30% and (b) cluster size changes from 10 to 10,000.

three larger datasets. Thanks to the precomputing step executed by TTK, it is 1.8 times faster than Explicit TopoCluster and two times faster than Implicit TopoCluster.

TTKMorseSmaleComplex. This plugin is used for computing a Morse-Smale (MS) complex from an input scalar function f defined on a simplicial complex Σ. An *integral line* is a path on Σ which is everywhere tangent to the gradient of f. Integral lines connect pairs of critical points of f. Intuitively, the *MS complex* is a segmentation of the input scalar field in regions where integral lines are connected to the same pair of critical points. Many algorithms have been proposed in the last twenty years [10] to compute MS complexes both in 2D and 3D. Among these, approaches based on discrete Morse theory [13] have proved to be efficient, simpler to implement, and more scalable [18], [26], [33]. The algorithm implemented in TTK, also based on discrete Morse theory, requires almost all relational operators (i.e., VE, VF, VT, EF, ET, FE, FT and TF operators) [30].

First, a discrete gradient is computed by visiting the simplices of the mesh, dimension by dimension, with an embarrassingly parallel process. After all vector pairs have been computed, simplices that are left unpaired are called *critical*. The cells of the MS complex are computed by visiting the discrete gradient starting from the critical simplices.

For the sake of our evaluation, it is important to underline that the extraction of the MS complex requires visiting clusters in a random fashion. That is, starting at a critical simplex σ, the visit is not limited to the cluster containing σ, but it may expand to the surrounding clusters. Then, there is no limit to the number of times each cluster is visited. This represents a worst-case scenario for TopoCluster, which is forced to recompute topological operators multiple times during the plugin execution.

For this experiment, the cluster size is set to 50, while the cache size is set to 10% for both Explicit and Implicit structure (see Section 8.3). Figure 8(a) shows the comparison of memory consumption. We notice that TTK triangulation uses about twice the memory required by Explicit TopoCluster and can only run on the four smaller datasets.

We can also observe how the memory footprint of TopoCluster does not increase monotonically when using datasets with increasing size. This is a consequence of reducing the memory footprint of relational operators. With only a limited amount of memory dedicated to relational operators, the memory requirement of TopoCluster becomes output-sensitive (i.e., it depends on the size of the MS complex). For this reason, the dataset with a more complicated MS complex, e.g., Foot or Engine datasets, uses more memory than larger datasets.

Figure 8(b) shows the run time comparison. On the four datasets for which TTK triangulation can complete the extraction, Implicit TopoCluster is four times slower, while Explicit TopoCluster is two times slower than TTK triangulation.

8.3 Cache and cluster size

In this section, we discuss the effects that different cache sizes and cluster sizes have on performance. Since a trend has been observed for all datasets, we only show the results about the *Foot* dataset for brevity.

TTKScalarFieldCriticalPoints. For a sequential algorithm, the cache size is of limited importance. Increasing cache size only results in increased memory usage with limited effects on run time. Figure 9(a) shows the trend for the *TTKScalarFieldCriticalPoints* with different cache sizes on two TopoCluster instances. Varying cache size from 1% to 50% of the total number of clusters, we see a sharp increase in memory usage with no significant speedup. However, since each cluster is visited only once by the plugin, a larger cache size does not necessarily improve time performance.

Different cluster sizes, instead, affect both execution time and memory usage. Figure 9(b) shows trends for the *Foot* dataset when varying cluster size from 100 to 1,000,000. As expected, using larger clusters reduces the number of tetrahedra shared by multiple clusters, and this leads to reduced memory consumption and faster algorithm execution. However, if the cluster accommodates too many vertices, the time and memory consumption will spike.

TTKMorseSmaleComplex. Unlike the sequential access pattern, the cache size parameter plays an important role in the algorithm that accesses clusters in a pseudo-random way. Figure 10(a) shows the speedup and compression factor obtained by varying cache size with *TTKMorseSmaleComplex* plugin. The largest cache size here is 30% since both Explicit and Implicit TopoCluster run out of memory when the cache size is larger than 30%. For this plugin, increasing the cache size leads to faster execution at the cost of memory requirements.

Timings increase when increasing the cluster size (see Figure 10(b)). This is explained by the fact that clusters are visited more than once, and re-computing larger clusters take more time. Similar to the sequential algorithm, memory consumption slightly decreases when larger clusters are used.

The lesson learned is that small cluster size is beneficial when an algorithm accesses clusters in a pseudo-random fashion. The available system memory should guide the choice of the optimal cache size since a larger cache size is always beneficial as long as the program does not run out of memory.

8.4 Parallel processing

TTK allows multithread execution by using OpenMP [6]. The main problem in allowing the use of OpenMP with TopoCluster is the cache system. A global LRU cache becomes the main bottleneck since each thread needs exclusive access to it. To address this problem, we have implemented a thread-based caching system. In practice, each thread has a dedicated cache.

If the thread-based cache solves the bottleneck issue, the cache size requires some adjustment since the maximum number of clusters stored in each cache will be multiplied by the number of threads. To this end, TopoCluster provides the functionality of a *dynamic* caching system, which allows the user to specify the size of the cache for a specific subset of the algorithm. In practice, the user can increase the cache size for serial sections, and divide the cache size across multiple threads for parallel sections.

The performance of the new thread-based caching system has been evaluated with *TTKScalarFieldCriticalPoints* and *TTKMorseSmaleComplex* plugins, using the same cluster size of the serial execution and using 12 threads. Since the main goal of TopoCluster is to provide control over memory usage, we balance the cache size requested for multi-thread and single-thread executions. We select a 12% cache size for the single-thread execution, while in parallel sections the cache size is reduced to 1%.

The algorithm implemented in *TTKScalarFieldCriticalPoints* is embarrassingly parallel. Thus, the cache size is maintained at 1% for the entire algorithm. Figure 11 (a) and (b) show memory usage and run time for the plugin, respectively.

The memory consumption is roughly the same as the serial execution for all data structures. Among the three data structures, Implicit TopoCluster always uses less memory and is the only data structure that can execute the plugin on all the datasets.

Although the run time improves for all data structures, general trends remain similar to the single-thread run. Implicit TopoCluster uses 50% less memory than the TTK triangulation but is 1.5x slower. Implicit TopoCluster has similar time performance as Explicit TopoCluster, while using only 60% of the memory.

Figure 11 (c) shows the speedup of the multi-thread execution for all three data structures compared with the single-thread version using 12% cache size. Both Explicit and Implicit TopoCluster get an average 3x speedup over the single-thread execution, while TTK triangulation is on average 2.5x faster. This is due to the

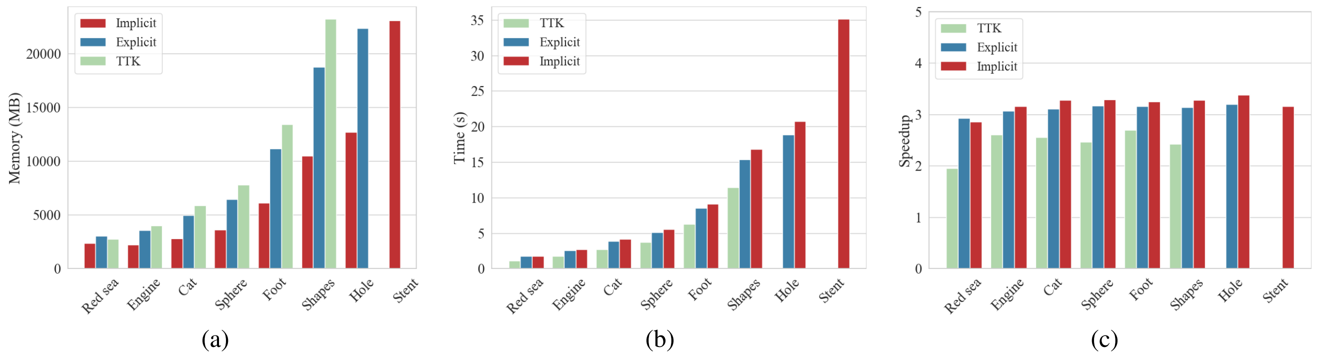


Fig. 11. The results obtained computing critical points (i.e., plugin TTKCriticalPoints) with TTK triangulation (TTK), Explicit TopoCluster (Explicit), and Implicit TopoCluster (Implicit) and enabling OpenMP support. (a) Memory consumption when 1% cache rate is used for TopoCluster. (b) Time usage when 1% cache rate is used for TopoCluster. (c) Speedup for all three data structures compared to serial execution.

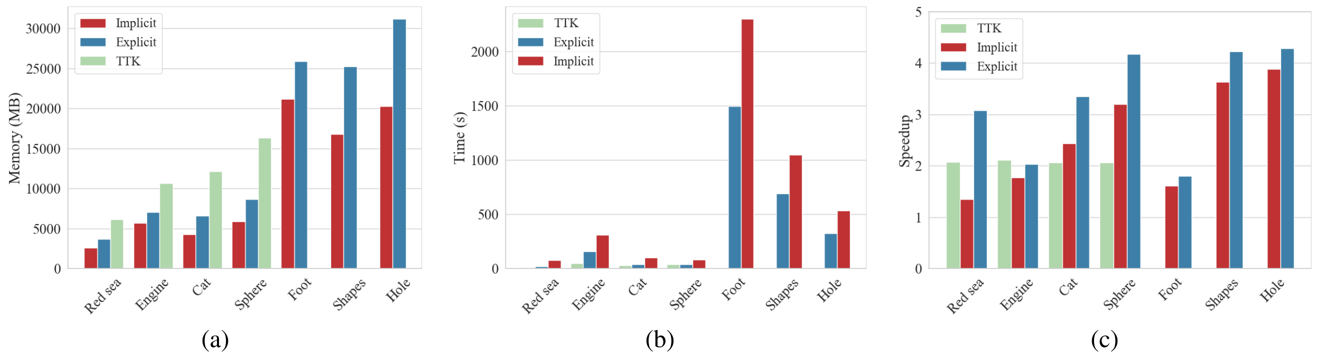


Fig. 12. The results obtained computing Morse-Smale complex (i.e., plugin TTKMorseSmaleComplex) with TTK triangulation (TTK), Explicit TopoCluster (Explicit), and Implicit TopoCluster (Implicit) and enabling OpenMP support. (a) Memory consumption when 1% cache rate is used for TopoCluster. (b) Time usage when 1% cache rate is used for TopoCluster. (c) Speedup for all three data structures compared to serial execution.

preprocessing step in TTK triangulation. Since relational operators are computed using a mix of parallel and sequential operations in the preprocessing phase, this limit the speedup obtained by TTK triangulation when using multiple threads.

Finding general trends in the *TTKMorseSmaleComplex* plugin is more challenging since not all steps can be executed in parallel. In this case, we use the dynamic caching system allocating 12% cache size for sequential steps, and 1% cache size for the parallel ones.

Figure 12 (a) and 12 (b) show the overall memory usage and timings. Implicit TopoCluster always uses less memory, on average 40% of the memory required by TTK triangulation and 70% of the memory required by Explicit TopoCluster. This compactness is ”paid” at run-time, where Implicit TopoCluster is on average 4 times slower than TTK triangulation and 2 times slower than Explicit TopoCluster.

Figure 12 (c) shows the speedup obtained by each structure compared with the results obtained using a single thread. We limit the comparison to the multithreaded execution, excluding the extraction of MS cells which is performed by a single thread. We notice that TTK triangulation provides a 2x speedup independently of the dataset size or the output size.

Since the complexity of the MS complex impacts on the performance of TopoCluster (see Section 8.2), datasets with a more complicated MS complex gets a lower speedup. For example, the speedup on Foot dataset is 1.8x for Explicit, and 1.6x Implicit TopoCluster. In general, Explicit TopoCluster gets an average 3.2x speedup, and Implicit TopoCluster gets an average 2.5x speedup.

In general, TTK triangulation provides best time performance, but it can only be used with meshes of limited size. If the user needs to limit memory consumption while maintaining competitive time performance, Explicit TopoCluster is a satisfactory pick. Implicit TopoCluster is the best choice with very large datasets or when the system has limited memory.

9 Conclusion

In this work, we have designed two new data structures, Explicit and Implicit TopoCluster, based on the Stellar decomposition model [12]. The scope of both data structures is to improve scalability by reducing memory consumption. Both data structures divide the simplicial mesh into clusters in order to process the mesh locally. Explicit TopoCluster encodes more information in the global layer and guarantees run-time efficiency while requiring more memory. On the contrary, Implicit TopoCluster encodes less information in the global layer and guarantees lower memory consumption with limited overhead. We have integrated both data structures in the Topology Toolkit [30], which provides an easy-to-use interface to developers and practitioners in topological data analysis. TopoCluster supports shared memory parallelization based on OpenMP [6], and it can be used with any plugin implemented in TTK.

In our experimental evaluation, we have compared Explicit and Implicit TopoCluster with TTK triangulation [30] and the Stellar tree [12]. Compared to TTK triangulation, Explicit TopoCluster requires half of the memory while still having comparable time performance. When minimal memory usage is crucial, Implicit TopoCluster requires an order of magnitude less memory but is twice slower than TTK triangulation. Compared to the Stellar tree, Explicit TopoCluster uses twice the memory while being 30% faster. Implicit TopoCluster uses 20% less memory while being up to 25% slower than the Stellar tree. However, TopoCluster provides a much easier interface for developers, and it is easier to integrate into existing frameworks for mesh processing.

Even though TopoCluster is currently designed for tetrahedral meshes, it is straightforward to adapt the data structure for triangle meshes. Generalizing TopoCluster to higher dimensions by enumerating all simplices is possible, but this could lead to severe performance decay since the number of simplices grows exponentially with the increase of the complex dimension. This problem affects all data structures that enumerate simplices in full [12].

By enabling OpenMP support in TopoCluster, we have observed that the local processing of the relational operators provides a higher speedup than TTK triangulation. A promising direction of our research is designing a new version of TopoCluster for distributed environments where groups of clusters are distributed across multiple machines.

Acknowledgments

The source code for the TopoCluster can be found at https://github.com/guoxiliu/TopoCluster. This work has been partially supported by the US National Science Foundation under grant number IIS-1910766. It has also been performed under the auspices of the German Aerospace Center (DLR) under Grant DLR-SC-2467209. The authors would like to thank Philips Research, Hamburg, Germany, for the Foot dataset, General Electric for the Engine dataset, and Michael Meißner, Viatronix Inc. for the Stent dataset. The Red Sea dataset is courtesy of the Red Sea Modeling and Prediction Group (PI Prof. Ibrahim Hoteit). The remaining tetrahedral meshes (Cat, Sphere, Shapes, and Hole) are courtesy of Yixin Hu from New York University.

References

[1] U. Ayachit, *The ParaView Guide: Updated for ParaView Version 4.3*, L. Avila, Ed. Los Alamos: Kitware, 2015.

[2] J.-D. Boissonnat and C. Maria, “The Simplex tree: An efficient data structure for general simplicial complexes,” *Algorithmica*, vol. 70, no. 3, pp. 406–427, 2014.

[3] D. Canino and L. De Floriani, “Representing simplicial complexes with Mangroves,” in *Proceedings of the 22nd International Meshing Roundtable*. Springer, 2014, pp. 465–483.

[4] D. Canino, L. De Floriani, and K. Weiss, “IA*: An adjacency-based representation for non-manifold simplicial shapes in arbitrary dimensions,” *Computers & Graphics*, vol. 35, no. 3, pp. 747–753, 2011.

[5] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, “VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data,” in *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, Oct 2012, pp. 357–372.

[6] L. Dagum and R. Menon, “OpenMP: an industry standard api for shared-memory programming,” *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[7] L. De Floriani, D. Greenfieldboyce, and A. Hui, “A data structure for non-manifold simplicial d-complexes,” in *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. ACM, 2004, pp. 83–92.

[8] L. De Floriani and A. Hui, “Data structures for simplicial complexes: an analysis and a comparison,” in *Proceedings of the third Eurographics symposium on Geometry processing*. Eurographics Association, 2005, pp. 119–es.

[9] L. De Floriani, A. Hui, D. Panozzo, and D. Canino, “A dimension-independent data structure for simplicial complexes,” *Proceedings of the 19th International Meshing Roundtable*, pp. 403–420, 2010.

[10] L. De Floriani, U. Fugacci, F. Iuricich, and P. Magillo, “Morse Complexes for Shape Segmentation and Homological Analysis: Discrete Models and Algorithms,” *Computer Graphics Forum*, vol. 34, no. 2, pp. 761–785, 2015.

[11] H. Edelsbrunner, *Algorithms in combinatorial geometry*. Springer Verlag, 1987, vol. 10.

[12] R. Fellegara, K. Weiss, and L. De Floriani, “The Stellar decomposition: A compact representation for simplicial complexes and beyond,” *Computers & Graphics*, vol. 98, pp. 322–343, Aug. 2021.

[13] R. Forman, “Morse theory for cell complexes,” *Advances in Mathematics*, vol. 134, no. 900145, pp. 90–145, 1998.

[14] E. Fredkin, “Trie memory,” *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[15] U. Fugacci, F. Iuricich, and L. De Floriani, “Computing discrete Morse complexes from simplicial complexes,” *Graphical Models*, vol. 103, p. 101023, May 2019.

[16] T. Gurung, D. Laney, P. Lindstrom, and J. Rossignac, “SQuad: Compact representation for triangle meshes,” in *Computer Graphics Forum*, vol. 30, no. 2. Wiley Online Library, 2011, pp. 355–364.

[17] T. Gurung and J. Rossignac, “SOT: a compact representation for tetrahedral meshes,” in *Proceedings SIAM/ACM Geometric and Physical Modeling*, ser. SPM ’09, San Francisco, USA, 2009, pp. 79–88.

[18] A. Gyulassy, P.-T. Bremer, and V. Pascucci, “Shared-memory parallel computation of Morse-Smale complexes with improved accuracy,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 1183–1192, Jan. 2019.

[19] C. Heine, H. Leitte, M. Hlawitschka, F. Iuricich, L. De Floriani, G. Scheuermann, H. Hagen, and C. Garth, “A survey of topology-based methods in visualization,” *Computer Graphics Forum*, vol. 35, no. 3, pp. 643–667, Jun. 2016.

[20] Y. Hu, Q. Zhou, X. Gao, A. Jacobson, D. Zorin, and D. Panozzo, “Tetrahedral meshing in the wild,” *ACM Trans. Graph.*, vol. 37, no. 4, pp. 60:1–60:14, Jul. 2018.

[21] D. Jönsson, P. Steneteg, E. Sundén, R. Englund, S. Kottravel, M. Falk, A. Ynnerman, I. Hotz, and T. Ropinski, “Inviwo - a visualization system with usage abstraction levels,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 11, pp. 3241–3254, 2019.

[22] M. Luffel, T. Gurung, P. Lindstrom, and J. Rossignac, “Grouper: A compact, streamable triangle mesh data structure,” *IEEE Annals of the History of Computing*, no. 01, pp. 84–98, 2014.

[23] M. Mantyla, *An Introduction to Solid Modeling*. Computer Science Press, 1988.

[24] G. M. Nielson, “Tools for triangulations and tetrahedralizations and constructing functions defined over them,” in *Scientific Visualization: overviews, Methodologies and Techniques*, G. M. Nielson, H. Hagen, and H. Müller, Eds. Silver Spring, MD: IEEE Computer Society, 1997, ch. 20, pp. 429–525.

[25] A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci, “Dimension-independent modeling with simplicial complexes,” *ACM Transactions on Graphics (TOG)*, vol. 12, no. 1, pp. 56–102, 1993.

[26] V. Robins, P. J. Wood, and A. P. Sheppard, “Theory and algorithms for constructing discrete Morse complexes from grayscale digital images,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 8, pp. 1646–1658, 2011.

[27] J. Rossignac, A. Safonova, and A. Szymczak, “3D compression made simple: Edge-Breaker on a Corner Table,” in *Proceedings Shape Modeling International*. Genova, Italy: IEEE Computer Society, May 2001.

[28] H. Samet, *Foundations of Multidimensional and Metric Data Structures*, ser. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2006, oCLC: ocm61731525.

[29] B. Schäling, *The Boost C++ Libraries*, 2nd ed. Laguna Hills, Calif: XML Press, 2014.

[30] J. Tierny, G. Favelier, J. A. Levine, C. Gueunet, and M. Michaux, “The Topology ToolKit,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 832–842, Jan. 2018.

[31] H. Toye, P. Zhan, G. Gopalakrishnan, A. R. Kartadikaria, H. Huang, O. Knio, and I. Hoteit, “Ensemble data assimilation in the Red Sea: sensitivity to ensemble selection and atmospheric forcing,” *Ocean Dynamics*, vol. 67, no. 7, pp. 915–933, Jul. 2017.

[32] K. Weiss, R. Fellegara, L. De Floriani, and M. Velloso, “The PR-star octree: a spatio-topological data structure for tetrahedral meshes,” in *Proceedings ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2011, pp. 92–101.

[33] K. Weiss, F. Iuricich, R. Fellegara, and L. De Floriani, "A primal/dual representation for discrete Morse complexes on tetrahedral meshes," in *Computer Graphics Forum*, vol. 32, no. 3pt3. Wiley Online Library, 2013, pp. 361–370.

Guoxi Liu Guoxi Liu is currently a Ph.D. student in the School of Computing at Clemson University and a member of the Visual Computing Lab. He got his Bachelor's degree in Computer Science from Chang'an University (China) in 2014. His research interests include Scientific Visualization, Topological Data Analysis, and Spatial Data Structures.

Federico Iuricich Federico Iuricich is an assistant professor in the School of Computing at Clemson University and a member of the Visual Computing Lab. He received his Ph.D. in Computer Science at the University of Genova (Italy) in 2014. Before joining Clemson, he has been a postdoctoral fellow in the Department of Computer Science at the University of Maryland, at College Park. His research expertise lies in topological methods for data analysis and visualization with a focus on unstructured data.

Riccardo Fellegara Riccardo Fellegara is a Senior Researcher of the Department of Software for Space Systems and Interactive Visualization, German Aerospace Center (DLR), Institute for Software Technology, Germany. He earned his Master and Ph.D. degrees in Computer Science at the University of Genova (Italy), in 2010 and 2015, respectively. He has been a Post-Doctoral Fellow Affiliate at Computer Science and Geographical Sciences departments of the University of Maryland, College Park, USA, from 2015 to 2019. His research interests include Spatial Data structures and Algorithms, Scientific Visualization, Topology-based Data Analysis, High Performance Computing (HPC), Geometric Modeling, and Geographic Information Systems.

Leila De Floriani Leila De Floriani is a professor at the University of Maryland at College Park (USA). De Floriani has been the 2020 President of the IEEE Computer Society. She is an IEEE Fellow, an IAPR Fellow, an Eurographics Association Fellow, a Solid Modeling Association Pioneer, and an inducted member of the IEEE Visualization Academy and of the IEEE Honor Society HKN. De Floriani has been the editor-in-chief of the IEEE Transactions on Visualization and Computer Graphics (TVCG) in 2015-2018, and she is currently an associated editor of several international journals, including Computers and Graphics, ACM Transactions on Spatial Algorithms and Systems, GeoInformatica, and Graphical Models. She has authored over 300 peer-reviewed scientific publications in data visualization, geospatial data representation and processing, computer graphics, geometric modeling, shape analysis and understanding.

APPENDIX A

Associating unique identifiers to the simplices of a simplicial complex represents a clear advantage for developers.

Figure 13 shows the snippet of a generic C++ procedure *computeEdgeValues()*. The procedure computes and saves a value for each edge of a simplicial complex. The value is computed based on the edge vertices.

The code shown in Figure 13 uses two functions provided by the underlying data structure; *getEdgeNumber()* returns the total number of edges in the simplicial complex; *getEdgeVertex(k,i)* returns the k^{th} vertex on the boundary of edge i.

```
1  void computeEdgeValues(){
2      int edgeNum = getEdgeNumber();
3      vector<int> values(edgeNum);
4      for(int i=0; i<edgeNum; i++){
5          int v1 = getEdgeVertex(0,i)
6          int v2 = getEdgeVertex(1,i)
7          values[i] = computeValue(v1,v2)
8      }
9  }
```

Fig. 13. Code snippet for the procedure *computeEdgeValues()* implemented in TopoCluster.

```
 1  void computeEdgeValues(){
 2      map<pair<int,int>, int> values();
 3      queue<Node> bfs_visit;
 4      bfs_visit.push(getRoot());
 5      while(!bfs_visit.empty()){
 6          Node node = bfs_visit.head();
 7          bfs_visit.pop();
 8          if(node.isLeaf()){
 9              for(pair<int,int> edge : node.getEdges()){
10                  if(values.find(edge) != values.end()){
11                      values[edge] =
12                      computeValue(edge.first,edge.second);
13                  }
14              }
15          }
16          else{
17              for(Node child : node.getChildren()){
18                  bfs_visit.push(child);
19              }
20          }
21      }
22  }
```

Fig. 14. Code snippet for the procedure *computeEdgeValues()* implemented in the Stellar tree.

The code is simplified thanks to the enumeration provided by TopoCluster. First, results are saved in a simple indexed vector (row 5). Second, each edge is visited by means of a simple for loop (row 6). As a consequence, making a parallel version of the same function would be trivial using OpenMP [6].

Implementing the same procedure without the enumeration property would require more involved code. Figure 14 shows a snippet of the C++ code implementing *computeEdgeValues()* on the Stellar tree [12]. Similar to TopoCluster, the Stellar tree is defined upon the Stellar decomposition model [12]. The difference is that it does not provide an enumeration for all the simplices of the mesh encoded. Simplices are organized in a hierarchical decomposition (a Point Region octree [28]) and represented through tuples of vertices (see Section 3).

The *std::vector* is now replaced by a *std::map* since now each edge is internally represented by a pair of vertices (row 2). The visit of all the edges is replaced by a breadth-first search of the hierarchical decomposition. The visit starts at the root of the hierarchy (row 4) and traverses the entire hierarchy until reaching the leaf nodes, which are the nodes storing the edges (row 8). Moreover, since each edge may appear in multiple nodes, duplicate entries need to be handled accordingly (row 10).