Computers & Graphics (2021)

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

The Stellar decomposition: A compact representation for simplicial complexes and beyond == Supplementary material ==

Riccardo Fellegara[a,*], Kenneth Weiss[b], Leila De Floriani[c]

[a] German Aerospace Center (DLR), Braunschweig, Germany
[b] Lawrence Livermore National Laboratory, Livermore, CA, USA
[c] University of Maryland at College Park, College Park, MD, USA

ARTICLE INFO

Article history:
Received May 24, 2021

Keywords: mesh data structure, scalable representations, Vietoris-Rips complex, high dimensional simplicial complexes

ABSTRACT

This document contains supplementary material for *"The Stellar decomposition: A compact representations for simplicial complexes and beyond"*. Appendix A provides additional details about the generation process for a Stellar decomposition. Appendix B provides details of the algorithm to reindex and SRE-compress the top cells CP within the Stellar decomposition. Finally, Appendix C discusses how a Stellar tree can be used as an intermediary data structure in generating other topological mesh data structures, for example, on large datasets or in memory constrained environments.

Appendix A. Generating a Stellar decomposition

In this section, we describe how to generate a COMPRESSED Stellar decomposition from an indexed CP complex Σ and a given partition Δ on the vertices of Σ. This process consists of three main phases:

1. reindex the vertices of Σ following a traversal of the regions of Δ and SRE-compress the r_V arrays;
2. insert the top CP cells of Σ into Δ;
3. reindex the top CP cells of Σ based on locality within common regions of Δ and SRE-compress the regions r_T arrays.

As it can be noted, the generation process ignores how the partitioning on the vertices is obtained, since this step is defined by the data structure instantiating a Stellar decomposition. The reindexing of the vertices follows a traversal of the regions of Δ in such a way that all vertices mapped to a region have a contiguous range of indices in the reindexed global vertex array Σ_V (as detailed in Appendix A.1). Figure .1 illustrates a reindexing of the vertices of a triangle mesh in the plane into a decomposition defind by four rectangular regions (dashed lines).

We then insert each top CP k-cell σ, with index i_σ in Σ_{T_k}, into all the regions of Δ that index its vertices. This is done by iterating through the vertices of σ and inserting i_σ into the r_T array of each region r whose vertex map $\Phi_{VERT}(r)$ contains at least one of these vertices. As such, each top CP k-cell σ appears in at least one and at most $|R_{k,0}(\sigma)|$ regions of Δ. Due to the vertex reindexing of step 1, this operation is extremely efficient. Determining if a vertex of a given cell lies in a block requires only a range comparison on its index i_v.

Finally, we reindex the top CP cell arrays Σ_T to better exploit the locality induced by the vertex-based partitioning and compress the local r_T arrays using a sequential range encoding over this new index. The reindexing and the compression of the top CP cells is obtained following a traversal of the regions of Δ in such a way that all top CP cells mapped from the same set of regions have a contiguous range of indices in the reindexed arrays Σ_T. This last step is detailed in Appendix A.2 and in Appendix B. As we demonstrate in the paper, this compression yields significant storage savings in a wide range of mesh datasets.

*Corresponding author
e-mail: riccardo.fellegara@dlr.de (Riccardo Fellegara), kweiss@llnl.gov (Kenneth Weiss), deflo@umiacs.umd.edu (Leila De Floriani)

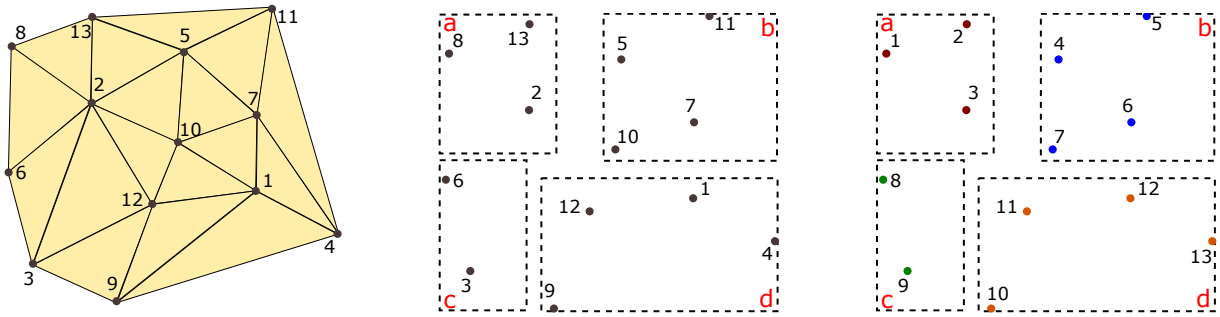

Figure .1. Generating a partition Δ over the vertices of a triangle mesh (a). After inserting the vertices (b), we reindex Σ_V according to the regions of Δ (c).

Algorithm 1 COMPRESS_AND_REINDEX_VERTICES(Δ, Σ)

Input: Δ is the decomposition defined on the vertices of Σ
Input: Σ is the CP complex
Variable: *v_perm* is an array containing new vertex indices
Variable: *current* references the current vertex id
Variable: v_s and v_e are the range of vertex indices in region r

```
current ← 0
// Step 1: Generate and apply new vertex index ranges
for all regions r in Δ do
    v_s ← current    // v_s is the first vertex index in r
    for all vertices v in Φ_VERT(r) (with index i_v in Σ_V) do
        v_perm[i_v] ← current
        current ← current +1
    v_e ← current    // v_e is the first vertex index outside r
// Step 2: Update R_{k,0} relation for all top CP k-cells in Σ
for all top CP cells σ in Σ_T do
    for j ← 0 to |R_{k,0}(σ)| do
        // the j-th entry of R_{k,0}(σ) has value i_v
        // v_perm[i_v] contains the new index of vertex v
        R_{k,0}(σ)[j] ← v_perm[i_v]
// Step 3: Update the vertex array in Σ_V
// (see Algorithm 6 in Appendix B)
PERMUTE_ARRAY(Σ_V,v_perm)
```

Algorithm 2 COMPRESS_AND_REINDEX_CELLS(Δ, Σ)

Input: Δ is the decomposition defined on the vertices of Σ
Input: Σ is the CP complex
Variable: *M* is an associative array mapping an integer identifier to each unique tuple of regions from Δ
Variable: *I* is an array associated with the unique tuples of regions in *M*
Variable: *t_position* is an array associated with the top CP cells

```
// Step 1: find unique tuples of regions and their counts
// (see Algorithm 3 in Appendix B)
EXTRACT_TUPLES(Δ,Σ,M,I,t_position)
// Step 2: find new position indices for top CP cells
// (see Algorithm 4 in Appendix B)
EXTRACT_CELL_INDICES(I,t_position)
// Step 3: reorder and SRE compress the top CP cells arrays
// (see Algorithm 5 in Appendix B)
COMPRESS_CELLS(Δ,t_position)
// Step 4: update the top CP cells array in Σ
// (see Algorithm 6 in Appendix B)
PERMUTE_ARRAY(Σ_T,t_position)
```

Appendix A.1. Reindexing and compressing the vertices

After generating the partition Δ and the vertex map Φ_{VERT} for the Stellar decomposition, we reindex the vertex array Σ_V to better exploit the coherence induced by Δ. At the end of this process, each region of Δ has a consecutive range of indices within the global vertex array Σ_V, and thus it trivially compresses under SRE to two values per block. We denote the starting and ending vertex indices as v_s and v_e, respectively.

This reindexing procedure is organized into three major steps, as outlined in Algorithm 1. The first step performs a traversal of the regions, which generates new indices for the vertices in Σ. For a region r, it generates a contiguous range of indices for the vertices in r. For example, in Figure .1, after executing step 1 on region b, we have $v_s = 4$ and $v_e = 7$.

The new indexes are then incorporated into mesh Σ by updating the vertex indices in $R_{k,0}$ relations for all top k-cells in Σ_{T_k} (see step 2 of Algorithm 1) and then permuting the vertices (detailed in step 3 of Algorithm 6 in Appendix B). These updates take place in memory without requiring any extra storage.

Appendix A.2. Reindexing and compressing the top CP cells

After inserting the top CP cells of Σ_T into Δ, we reorder the top CP cells array Σ_T based on the partitioning and apply SRE compaction to the region arrays to generate our COMPRESSED encoding. This reindexing exploits the coherence of top CP cells that are indexed by the same set of regions, translating the proximity in Δ into *index* proximity in Σ_T. This procedure is organized into four main phases, as shown in Algorithm 2. A detailed description can be found in Appendix B.

The EXTRACT_TUPLES procedure (see Algorithm 3 in Appendix B), traverses the regions of Δ to find the tuple of regions $t_b = (r_1, \ldots, r_n)$ in Δ that index a top CP cell σ. Inverting this relation provides the list of top cells from Σ mapped to each such tuple of regions. As we iterate through regions, we ensure that each top CP cell in the complex is processed by only one region r, by skipping the top CP cells whose minimum vertex index i_v

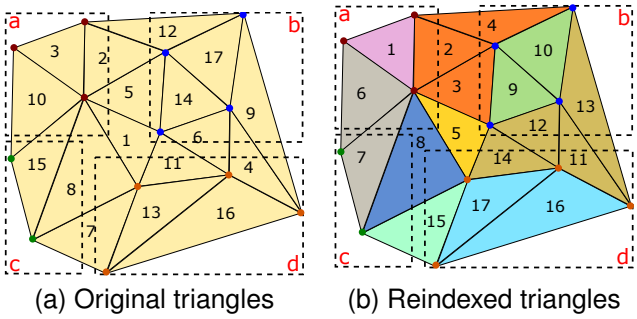

Figure A.2. Top cell indices before (a) and after (b) tuple-based reindexing.

is not in $\Phi_{VERT}(r)$. For example in Figure A.2(a), triangle 5 is indexed by region a and b, and, thus, its tuple is $t_b = (a, b)$. The complete list of triangles in tuple (a, b) is $\{2, 5, 12\}$.

We use this inverted relation in EXTRACT_CELL_INDICES (see Algorithm 4 in Appendix B), to generate a new coherent order for the top CP cells of Σ_T. Specifically, the prefix sum of the tuple cell counts provides the starting index for cells in that group. For example, when considered in lexicographic order, the first three region tuples, (a), (a, b) and (a, b, d) in Figure A.2(b), with 1, 3 and 1 triangles, respectively, get starting indices 1, 2 and 5. We then assign increasing indices to the top CP cells of each group. Thus, e.g., the three triangles belonging to tuple (a, b) get indices $\{2, 3, 4\}$ after this reindexing.

Finally, in COMPRESS_CELLS and PERMUTE_ARRAY (Algorithm 5 and 6 in Appendix B), we reorder and SRE-compact the r_T region arrays and the global top CP cells array Σ_T.

Appendix B. Algorithm details: reindexing and compressing the top CP cells

This appendix provides details for reindexing the top CP cells during the Stellar tree generation algorithm (Algorithm 2) as outlined in Appendix A.2. The reindexing exploits the coherence of top CP cells that are indexed by the same set of regions by translating the proximity in Δ into *index* proximity in Σ_T and depends on three auxiliary data structures:

- an associative array, M, which maps an (integer) identifier to each unique tuple of regions;
- an array of integers, I, having the same number of entries as M. Initially, it is used to track the number of top CP cells associated with each tuple of regions. In a successive phase, it tracks the next index for a top CP cell in a tuple;
- an array of integers, *t_position*, of size $|\Sigma_T|$. Initially, it is used to associate top CP cells with their region tuple identifier. In a successive phase, it is used to store the new coherent indices for the top CP cells.

The remainder of this appendix summarizes the four major steps of Algorithm 2. Figure B.3 illustrates this reorganization process over a triangle mesh.

Algorithm 3 EXTRACT_TUPLES($\Delta, \Sigma, M, I, t\_position$)

Input: Δ is the decomposition defined on the vertices of Σ
Input: Σ is the CP complex
Input: M is a map associating a unique identifier to each tuple
Input: I is an array that tracks top CP cells in each tuple
Input: *t_position* is an array linking top CP cells to tuples
1: **for all** regions r **in** Δ **do**
2: **for all** σ **in** $\Phi_{TOP}(r)$ (with index i_σ in Σ_T) **do**
// extract the minimum vertex index of σ
3: $i_v \leftarrow$ GET_MIN_VERTEX_INDEX(σ)
// we visit σ if r *indexes* i_v
4: **if** i_v **in** $\Phi_{VERT}(r)$ **then**
// extract the tuple t_r of regions indexing σ
5: $t_r \leftarrow$ EXTRACT_TUPLE(Δ,σ)
// get the tuple key in M (if not present insert it)
6: key $\leftarrow$ M[t_r]
// increment the counter of this tuple in I
7: I[key] $\leftarrow$ I[key]+1
// associate the tuple key to the i_σ *entry in t_position*
8: t_position[i_σ] $\leftarrow$ key

Algorithm 4 EXTRACT_CELL_INDICES($I, t\_position$)

Input: I is an array associated with the unique leaf tuples in M
Input: *t_position* contains tuple index for top cell
Ensure: *t_position* contains new top simplex position indexes
Variable: c is the counter variable of the current position index
// convert the cell counts in I into the starting indexes for top CP cells grouped by the same tuple
1: **for all** key **in** I **do**
2: tmp $\leftarrow$ I[key] *// I[key] contains cells count for tuple key*
3: I[key] $\leftarrow$ c
4: c $\leftarrow$ c + tmp
// assign to each top CP cell its new position index in Σ_T
5: **for all** σ **in** Σ_T (with index i_σ in Σ_T) **do**
6: key $\leftarrow$ t_position[i_σ]
7: t_position[i_σ] = I[key] *// set new position index for σ*
8: I[key] $\leftarrow$ I[key]+1

EXTRACT_TUPLES. In Algorithm 3, we generate map M, count the number of top CP cells associated with each tuple of regions in array I and initialize the *t_position* array entries with its tuple identifier:

- for each region r in Δ, we visit the top CP cells σ in $\Phi_{TOP}(r)$ whose minimum vertex index i_v (row 4) is indexed in r. This ensures that each top CP cell is processed only once. Regions of Δ are uniquely indexed by the index of their starting vertex v_s;
- for each such top CP cell σ with index i_σ, we traverse the partitioning on the vertices to find the tuple of regions from Δ that index σ (row 5 function EXTRACT_TUPLE). We then look up its unique identifier *key* in M (or create a new one and insert it into M) (row 6). We then increment the count for this tuple, and associate σ with this tuple (rows 7 and 8).

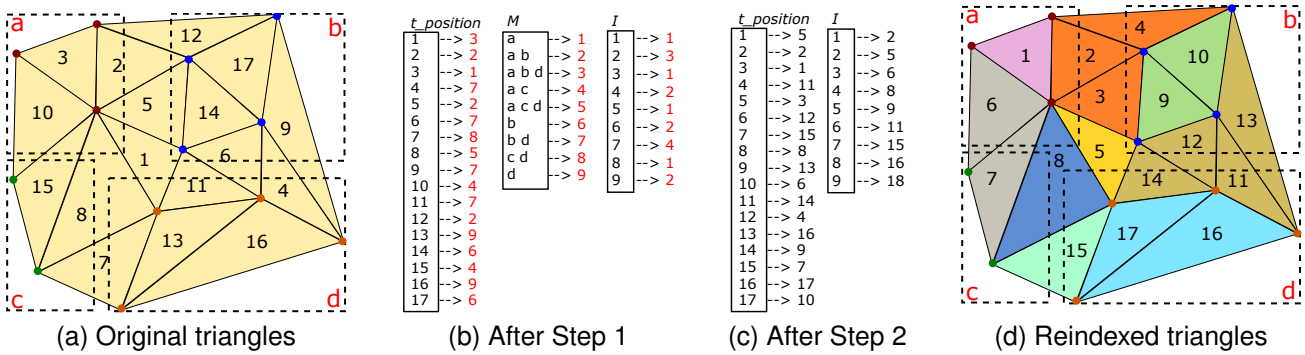

Figure B.3. Top cell reindexing. (a) initial tree with four leaf blocks a, b, c, d (b,c) auxiliary data structures after Steps 1 and 2 (d) reindexed tree.

Algorithm 5 COMPRESS_CELLS(Δ, *t_position*)

Input: Δ is the decomposition defined on the vertices of Σ
Input: *t_position* contains the new top CP cell position indices
1: **for all** regions r **in** Δ **do**
2: r_T_aux $\leftarrow \Phi_{TOP}(r)$ *// copy the top CP cells array of* r
3: $\Phi_{TOP}(r) \leftarrow \emptyset$ *// reset that array*
// update the indices in r_T_aux with those from t_position
4: **for** id $\leftarrow$ 0 **to** |r_T_aux| **do**
5: r_T_aux[id] $\leftarrow$ t_position[r_T_aux[id]]
6: SORT(r_T_aux)
7: start_id $\leftarrow$ r_T_aux.FIRST()
8: counter $\leftarrow$ 0
9: **for** id $\leftarrow$ 1 **to** |r_T_aux| **do**
// if we find consecutive indexes
10: **if** r_T_aux[id]+1 = r_T_aux[id+1] **then**
11: counter $\leftarrow$ counter+1
12: **else**
13: **if** counter > 1 **then** *// found a run of indices*
// create a run in r_T of r
14: CREATE_SRE_RUN(r,start_id,counter)
15: **else** *// simply add the top CP cell index in r_T*
16: r.ADD_TOP(start_id)
// reset the two auxiliary variable
17: start_id $\leftarrow$ r_T_aux.NEXT()
18: counter $\leftarrow$ 0

Algorithm 6 PERMUTE_ARRAY(*array*, *permutation*)

Input: *array* is the simplex array to update
Input: *permutation* is the array containing the new position indices
1: **for** id $\leftarrow$ 0 **to** |array| **do**
// current vertex is in correct position or updated already
2: **if** permutation[id] = id **then**
3: permutation[id] $\leftarrow$ -1 *// mark id as updated*
4: **else** *// id is not updated already*
// iteratively update vertices positions
5: **while** permutation[id] $\neq$ id **do**
// swap id and permutation[id] entries in array
6: SWAP(array,id,permutation[id])
// mark id as updated in permutation
// then, get the id of the next vertex to update
7: id $\leftarrow$ MARK_AND_GET_NEXT(id,permutation)
8: *permutation*[*id*] $\leftarrow$ -1 *// mark id as updated*

At the end of the traversal of Δ, each entry of *t_position* contains the identifier of the tuple of regions indexing its corresponding top cell and I contains the number of top CP cells indexed by each tuple. M is no longer needed and we can discard it.

The content of auxiliary data structures, after this step, is illustrated in Figure B.3(b). For example, triangle 5 is indexed by regions a and b, whose key in M is 2. This tuple contains two triangles other than 5, as indicated by the corresponding counter in I.

EXTRACT_CELL_INDICES. In Algorithm 4, we use the I and *t_position* arrays to find the updated index for each top CP cell in Σ_T, which is computed in place in *t_position*.

First, we convert the cell counts in array I into starting indexes for the top CP cells grouped by the same set of regions, by taking the *prefix sum* of array I (rows 1 to 4).

Then, we use array I to update *t_position* array by iterating over the top CP cells, and replacing the tuple identifier in *t_position* with the next available index from I and increment the counter in I (rows 5 to 8). At this point, *t_position* is a permutation array that encodes a more coherent ordering for the top CP cells and I is no longer needed.

The content of auxiliary data structures after this step, is shown in Figure B.3(c). At the end, each entry of I contains the first index of the next tuple, while *t_position* the new position for the i-th triangle.

COMPRESS_TREE_CELLS. In Algorithm 5, we apply this order to the arrays r_T of top CP cells of each region r and compact the r_T arrays using the SRE compression. This procedure iteratively visits all regions of a Stellar decomposition. Within each region r, an auxiliary array, called r_T*_aux*, is used, encoding, initially, a copy of the array of top CP cells position indices encoded by r (row 2). Then, these indices are updated with the coherent ones from *t_permutation* (rows 4 and 5), and, finally, by sorting this array we have sequential indices in consecutive position of r_T*_aux* (row 6).

Next, we identify consecutive index runs by iterating over

r_T_aux array (rows 7 to 18). In this phase, we use two auxiliary variables, a counter, encoding the size of the current run, and a variable, called *start_id*, encoding the starting index of the current run. If we find two consecutive indices, we simply increment *counter*. Otherwise, we check if we have a run (row 13), or if we have to simply add the index in *start_id* to r_T array of r (row 16). If we have to encode a run in r_T (procedure CREATE_SRE_RUN, row 17), we apply the strategy, described in Section 4.2.2, for encoding it.

PERMUTE_ARRAY. Finally, in Algorithm 6, we update the global top CP cells array Σ_T. This is done by iteratively swap the entries in Σ_T (rows 5 to 8), applying the new coherent indices encoded in *permutation* array. This procedure does in place updates and, thus, does not require any additional auxiliary data structure.

Algorithm 7 EXTRACT_$R_{d,d}$_MANIFOLD(r, Σ)

Input: r is a leaf block in $\mathbb{H}$
Input: Σ is the CP complex indexed by $\mathbb{H}$
Variable: $d\_1\_cell\_top$ encodes $R_{d-1,d}$ for the $(d-1)$-cells in r
1: **for all** top CP d-cells σ **in** $\Phi_{TOP}(r)$ **do**
2: **for all** $(d-1)$-cells τ **in** $R_{d,d-1}(\sigma)$ **do**
3: add σ to $d-1\_cell\_top[\tau]$
4: **for all** $(d-1)$-cells τ **in** $d\_1\_cell\_top$ **do**
5: **if** $|d\_1\_cell\_top[\tau]| = 2$ **then**
6: $\{\sigma_1, \sigma_2\} \leftarrow d\_1\_cell\_top[\tau]$
7: set σ_1 as adjacent d-cell for σ_2 in τ
8: set σ_2 as adjacent d-cell for σ_1 in τ
9: **else** // $|d\_1\_cell\_top[\tau]| = 1$
10: mark cell τ of σ as a boundary cell

Appendix C. Generating topological data structures

As a proxy for more complicated mesh processing workflows, this section describes how Stellar trees can be used to generate an existing topological data structure over CP complexes: the IA* data structure. The IA* data structure is the most compact data structure in the class of connectivity-based representations since it encodes only the vertices and the top cells of a CP complex Σ, as well as a subset of topological relations connecting these cells. As with other topological data structures, the IA* over Σ is typically generated directly from the indexed representation of Σ by extracting its adjacency and co-boundary relations. However, this direct approach could present issues when scaling to large complexes on commodity hardware due to the high storage requirements for auxiliary data structures needed to reconstruct these relations.

This application demonstrates the versatility of the Stellar tree representation and exercises many of the operations necessary for other mesh processing tasks. We define customized topological relations and auxiliary data structures as we stream through the leaf blocks of the tree and take either a *global* approach, to reconstruct the full topological data structure, or a *local* approach, which reconstructs coherent subsets of the full data structure restricted to the portion of the complex indexed within each leaf block. In the former case, Stellar trees enable generating the global topological data structures using a fraction of the memory as would be required to directly generate them from an indexed representation. In the latter case, the local approach can be used to adapt local regions of the Stellar tree's underlying complex to algorithms defined for existing topological data structures.

In the following, we present a local generation algorithm over a single leaf block of the Stellar tree, and compare the local and global generation algorithms against a direct approach that generates the data structure from the original indexed mesh representation. We do this within the Stellar tree framework by setting the bucketing threshold to infinity, since $k_V = \infty$ produces a tree that indexes the entire complex Σ in its root block.

Recall from Section 8.2 of the paper that the IA* data structure is an adjacency-based topological data structure defined over non-manifold d-dimensional CP complexes that gracefully degrades to the IA representation over manifold complexes. The IA data structure is defined over pseudo-manifolds, and, thus, each $(d-1)$-cell can be incident in at most two top CP d-cells. We first describe how to generate the IA data structure from the Stellar tree, and then extend this to the IA* data structure.

The IA data structure encodes the following topological relations: (i) boundary relation $R_{d,0}(\sigma)$, (ii) partial co-boundary relation $R^*_{0,d}(v)$ for each vertex v, consisting of one arbitrarily selected top CP d-cell in the star of v, and (iii) adjacency relation $R_{d,d}(\sigma)$, for each top CP d-cell σ. If σ_1 is adjacent to σ_2 through $(d-1)$-cell τ, and τ is the i-th face of σ_1, then σ_2 will be in position i in the ordered list of $R_{d,d}(\sigma_1)$.

Since the Stellar tree explicitly encodes the $R_{d,0}$ relations for all top CP d-cells, the generation of a *local* IA data structure consists of extracting $R^*_{0,d}(v)$, for each v in $\Phi_{VERT}(r)$, and $R_{d,d}(\sigma)$, for each top CP d-cell σ in $\Phi_{TOP}(r)$. For vertices in $\Phi_{VERT}(r)$, the former is computed by iterating over the top CP d-cells in $\Phi_{TOP}(r)$, and selecting the first top CP cell incident in v that we find.

Algorithm 7 provides a description of a *local* strategy for extracting $R_{d,d}(\sigma)$ relations within block r of the tree. Note that it finds only the adjacencies across (d-1)-faces that have at least one vertex in $\Phi_{VERT}(r)$. While we can locally reconstruct the full adjacency relation for top CP d-cells with d vertices in $\Phi_{VERT}(r)$, a top CP d-cell σ with fewer vertices in $\Phi_{VERT}(r)$ might be missing at least one adjacency. For example, in Figure C.4, we can completely reconstruct the adjacency relations of the triangles having two vertices in r (in yellow), while we can only partially reconstruct the adjacencies of triangles having just one vertex in r (in gray). Adjacencies on the edges opposite to the vertices in red cannot be reconstructed inside r for gray triangles.

The algorithm first iterates on the top CP d-cells in $\Phi_{TOP}(r)$ (rows 1–3). Given a top CP d-cell σ, we cycle over the d-tuples of the vertices of σ, where each d-tuple defines a $(d-1)$-cell on the boundary of σ. The auxiliary data structure $d\_1\_cell\_top$ encodes, for each d-tuple τ, the top d-cells sharing τ, corresponding to the $R_{d-1,d}$ relation of τ. Then, the algorithm iterates over $d\_1\_cell\_top$ to initialize adjacency relations $R_{d,d}$. Given a

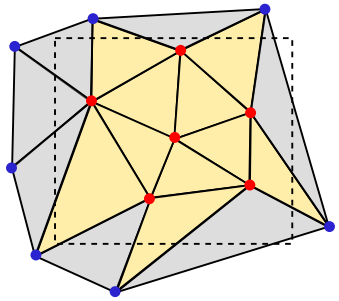

Figure C.4. ***Local*** **adjacency reconstruction finds adjacencies across faces with a vertex in the leaf block** r **(dashed). For yellow triangles, all edges have a vertex in** r**, while some edges of gray triangles do not.**

$(d-1)$-cell τ, if τ has two d-cells in its co-boundary (row 5), namely σ_1 and σ_2, we set σ_1 and σ_2 as adjacent along τ (rows 7–8). Due to its local nature, the Stellar tree adjacency reconstruction provides considerable storage savings compared to its global counterpart: the storage requirements are proportional to the number of top CP d-cells in r, rather than those in Σ_T.

Extending this algorithm to generate a *global* IA data structure requires only a few modifications. Aside from encoding the auxiliary data structures at a global level, the other major difference with respect to the local approach is that, within each leaf block r, $R_{d-1,d}$ relations are extracted only for those $(d-1)$-cells τ for which the two top CP d-cells sharing τ have not been already initialized.

The IA* data structure extends the IA data structure to arbitrary non-manifold CP k-complexes, with $0 < k \leq d$. Recall that, in addition to the relations stored in the IA data structure, it encodes: (i) adjacency relation $R_{k,k}(\sigma)$, for each top CP k-cell σ; (ii) co-boundary relation $R_{0,1}(v)$ restricted to the top 1-cells, for each vertex v; (iii) *augmented* partial co-boundary relation $(R^*_{0,k}(v))$, $1 < k \leq d$, for each vertex v, consisting of one arbitrarily selected top CP k-cell from each k-*cluster* in the star of v, where a k-cluster is a $(k-1)$-connected component of the star of v restricted to its top CP k-cells; and (iv) co-boundary relation $R_{k-1,k}(\tau)$, for each non-manifold $(k-1)$-cell τ bounding a top CP k-cell.

Extracting $R_{k,k}$ relations, when $k < d$, and $R_{k-1,k}$ relations for non-manifold $(k-1)$-cells is performed by a suitable extension of Algorithm 7. Augmented partial co-boundary relation $R^*_{0,k}(v)$, for $k > 1$, is computed by extracting the restricted star of v (Algorithm 3 in the paper) and by using $R_{k,k}$ relation for the top CP cells in the star of v to identify the $(k-1)$-connected components incident in v. $R_{0,1}(v)$ is initialized by iterating over the top 1-cells in the restricted star of v.

Experimental results. In Table C.1, we compare the time, and storage requirements to generate an IA or IA* data structure, depending on whether the complex has a manifold or non-manifold domain, using the Stellar tree or directly extracting it from the indexed representation. For each dataset, we compare the Stellar trees generated by using thresholds k_S and k_L and by using a local and a global algorithm against the direct approach on the original indexed representation of the complex. For the manifold (*triangular*, *quadrilateral*, *tetrahedral* and *hexahedral*) and pure (*probabilistic*) datasets, where all top cells have dimension d, we used Algorithm 7 to compute the adjacencies.

When comparing execution times, we find that the global Stellar tree approach is about 25% faster than the direct approach in most cases. However, due in part to the redundant lookups in the adjacency calculation, the local approach is slightly slower than the global approach, but still 10% faster than the direct approach in most cases. For example, it is almost twice as fast on *F16*, on par on *Lucy* and slower on the *5D probabilistic* dataset. Considering the effects of the bucket threshold k_V, we observe little discernible difference on the global Stellar tree approach. However, a larger bucketing threshold (k_L) yielded up to a 25% speedup in the local approach on our larger datasets, compared to its smaller (k_S) counterpart.

Lastly, we consider the storage requirements for generating the IA / IA* data structure. For both the local and global Stellar tree tree approaches, the auxiliary storage requirements are limited to the complexity of each leaf block, requiring only a few KBs of auxiliary storage for the manifold and non-manifold datasets, and a few MBs for the pure (*probabilistic*) datasets. In contrast, the direct approach requires hundreds of MBs for the medium sized datasets. We were not able to generate the IA* data structures using the direct approach on our largest datasets, which ran out of memory (*OOM*) on our workstation, despite its 64 GB of available RAM.

Table C.1. Generation times (seconds) and storage (number of references) for the IA* data structure from Stellar trees (k_S and k_L) and the direct approach (*dir.*) on the original indexed representation of the complex. With the exception of V-RIPS complexes, the IA* is equivalent to the IA representation on these datasets.

Data	k_V	Time		Storage IA/IA*		aux.
		local	global	local	global	d.s.
TRIANGULAR						
NEPTUNE	k_S	6.88	5.69	0.36K		0.70K
	k_L	6.51	5.84	1.65K	6.01M	3.24K
	dir.		9.69			12.0M
STATUETTE	k_S	17.5	14.5	0.38K		0.72K
	k_L	17.0	14.9	1.62K	15.0M	3.22K
	dir.		20.7			30.0M
LUCY	k_S	47.4	39.6	0.42K		0.82K
	k_L	49.0	40.7	1.64K	42.0M	3.28K
	dir.		50.6			84.1M
QUADRILATERAL						
NEPTUNE	k_S	36.6	31.6	0.27K		0.52K
	k_L	35.0	31.3	1.70K	24.0M	3.37K
	dir.		44.1			48.1M
STATUETTE	k_S	92.2	78.9	0.26K		0.50K
	k_L	90.4	79.4	1.74K	60.0M	3.38K
	dir.		102			120M
LUCY	k_S	250	218	0.27K		0.53K
	k_L	250	221	1.74K	168M	3.40K
	dir.		252			336M
TETRAHEDRAL						
BONSAI	k_S	56.5	38.5	3.20K		10.2K
	k_L	49.4	38.8	6.29K	28.6M	20.6K
	dir.		60.0			97.7M
VISMALE	k_S	61.2	42.0	3.22K		10.4K
	k_L	53.3	43.0	5.52K	31.1M	17.4K
	dir.		66.3			106M
FOOT	k_S	72.6	47.2	3.21K		10.3K
	k_L	58.7	47.1	6.42K	34.5M	21.0K
	dir.		72.7			118M
HEXAHEDRAL						
F16	k_S	152	102	0.32K		0.83K
	k_L	129	103	2.38K	53.3M	6.42K
	dir.		237			152M
SAN FERN	k_S	380	217	0.38K		1.05K
	k_L	273	219	2.64K	117M	7.31K
	dir.		285			336M
VISMALE	k_S	844	459	0.23K	261M	0.59K
	k_L	591	477	2.13K		5.82K
	dir.		OOM		–	–
PROBABILISTIC						
5D	k_S	209	77.6	15.3K		84.2K
	k_L	148	75.0	95.3K	26.9M	535K
	dir.		108			159M
7D	k_S	4.84K	1.63K	1.05M	258M	7.66M
	k_L	3.89K	1.53K	4.30M		32.5M
	dir.		OOM		–	–
40D	k_S	30.1K	24.3K	1.36M	16.7M	55.3M
	k_L	28.3K	22.9K	5.04M		205M
	dir.		OOM		–	–
V-RIPS						
VISMALE 7D	k_S	45.8	42.1	2.24K		4.59K
	k_L	47.3	42.4	3.36K	11.0M	6.29K
	dir.		43.3			35.2M
FOOT 10D	k_S	694	595	13.6K		88.8K
	k_L	528	558	17.4K	68.9M	115K
	dir.		899			552M
LUCY 34D	k_S	804	763	5.44K	55.1M	23.2K
	k_L	688	615	12.6K		58.3K
	dir.		OOM		–	–