Efficient Implementation of Strong Collapses

Divyansh Pareek

Intern under Prof. Jean-Daniel Boissonnat, Team DATASHAPE
INRIA Sophia Antipolis

May-July 2017

1 Problem Outline

Main Problem : Given a Simplicial Complex K, figure out an efficient way to compute the Simplicial Complex L such that L is the core of K.

We started with the aim of implementing the Nerve definition of Strong collapse, which says that for an (abstract) Simplicial Complex K, $N^2(K)$ gives us an (abstract) Simplicial Complex L such that L is the result of a certain number of strong collapses on K.

This number need not be 1 (ie, K to L need not be an elementary collapse) and this number also need not be the maximum possible (ie, L need not be the core of K). But, there will certainly exist a series of elementary collapses such that K collapses to L.

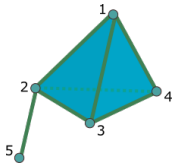
The definition of nerve requires a set X and a cover U of that. Here, the set X is the set of all vertices of K and the cover U is the set of all maximal simplices of K.

2 Data Structure MsMatrix

The Data Structure *MsMatrix* came up naturally when we thought of how to represent the set X (the vertices of K) and its cover U (the Maximal Simplices of K) with the relation of which Maximal Simplex has which vertices.

It is a 2D Matrix of boolean values with the vertices as the rows and the Maximal Simplices as columns of the Matrix.

Call this Matrix M. Then M[v][mxs] stores *true* when the maximal simplex mxs contains the vertex v. Otherwise, M[v][mxs] stores *false*.



(a) A sample Simplicial Complex

1	✓	×
2	✓	✓
3	✓	×
4	✓	×
5	×	✓

(b) Corresponding MsMatrix for the same

Figure 1: An example for the Data Structure MsMatrix

3 Method for creation of MsMatrix from a given Simplex Tree of a Simplicial Complex

The idea is to do a Depth-First search on the Simplex Tree and for all leaf nodes of the tree [candidates for Maximal Simplices], check if there is already a maximal simplex inserted into the matrix that is a co-face of the current simplex in concern.
If not, then insert this simplex as a maximal simplex into the Matrix [candidacy confirmed]. Else, don't, because it is confirmed that this is not a maximal simplex.

This works because of a small point of ingenuity, which is that a candidate for Maximal Simplex (a candidate is a leaf node in the Simplex Tree) is rejected only if there is a co-face to it. And it so happens that :

Statement. A co-face to a candidate simplex (leaf node) must occur "before" it in the Depth-First-Search.

Proof. Assume a candidate simplex σ with vertices $V_\sigma = \{s_0, s_1, ..., s_k\}$ satisfying $s_0 < s_1 < ... < s_k$ and a co-face τ of σ with vertices $V_\tau = \{t_0, t_1, ..., t_l\}$ satisfying $t_0 < t_1 < ... < t_l$. Clearly, $V_\sigma \subset V_\tau$.
Also, $s_0 \neq t_0$ because if $s_0 = t_0$, then σ cannot be at a leaf node of the Simplex Tree. So, it must be the case that $t_0 < s_0$ (and that $s_0 = t_j$ for some $0 < j <= l$). Which means that the "branch" of the vertex t_0 in the Simplex tree contains the simplex τ and the "branch" of the vertex s_0 contains the simplex σ and because of the fact $t_0 < s_0$, we can conclude that τ must appear before σ in the Simplex tree.

4 Algorithm 1

Realisation of another small trick gives us our first algorithm.
Once the Matrix is created, it is very easy to construct the Nerve of the given Simplex Tree. Basically, each Maximal Simplex becomes a vertex in the Nerve, and a simplex appears in the nerve if there is some common intersection between the vertices of the nerves.

And it is not hard to realise that corresponding to each row of the matrix (each vertex of the original Simplex Tree), inserting the Simplex corresponding to all *true* boolean values **with all its faces** gives us the Simplex Tree corresponding to the Simplicial Complex of the Nerve.

Hence, a simple algorithm appears :

$tree_{old} \leftarrow stree$
$tree_{new} \leftarrow nerve(nerve(tree_{old}))$
while $tree_{new} \neq tree_{old}$ **do**
  $tree_{old} \leftarrow tree_{new}$
  $tree_{new} \leftarrow nerve(nerve(tree_{old}))$
end while

The result would be that both the variables $tree_{old}$ and $tree_{new}$ will contain the Simplex Tree corresponding to the Core of the Simplex Tree *stree*.
The function nerve() behaves as :

Simplex_tree nerve(Simplex_tree *st*):
1. Create the Matrix corresponding to *st* using the DFS strategy
2. Create the Simplex_tree st_{nerve} by going over all the rows and inserting the Simplex (with all subfaces) corresponding to all 'ticks' of each particular row
3. Return st_{nerve}

5 Linking the two definitions of Strong Collapse

If we think about the previous section, we get some insight into what we are actually doing.
Basically, after creating the matrix, for getting the Nerve Simplicial Complex, we are inserting simplices with subfaces corresponding to each row of that matrix. What if the 'ticks' in one row (say row number x) are a subset of those in another row (say row number y) ?
What it means is that it does not matter whether we do the inserting simplex with subfaces operation for row x because that will have no effect on the resultant simplicial complex (because doing that operation for row y will automatically include whatever row x had to offer).
Incidentally, this also happens to be the exact criterion for a vertex to be dominated.* Here, vertex corresponding to row x is dominated by the vertex corresponding to row y.

[*As stated in : J.A. Barmak, E.G. Minian. Strong homotopy types, nerves and collapses. Discrete and Computational Geometry 47(2012), pp. 301-328. A vertex v in a complex K is dominated by a vertex v' iff all the maximal simplices of K which contains v also contains v'.]

What this means is that :
Doing one $N^2(K)$ operation on a Simplicial Complex K is the same as collapsing *ALL* dominated vertices of the Simplicial Complex K.

MAIN MATERIAL NOW

6 Final Algorithm

Our *algorithm 1* goes from Tree to Matrix to Tree for taking the Nerve of a Simplicial Complex K. Which means for one $N^2(K)$ operation, it goes from Tree to Matrix to Tree to Matrix to Tree. That is a lot of computation.
What if we take hints from the previous section, and once the Matrix is constructed for the original Simplicial Complex K, perform all collapse operations on the matrix only. Our final algorithm does precisely that.

Once the matrix is constructed, we identify ALL dominated vertices using the [*] criterion (mentioned in the previous section), and collapse them (which means remove those rows from the matrix). This step corresponding to taking the first Nerve. This may result in some columns being redundant, ie, some maximal simplices being no longer maximal (This is the same as those vertices in the Nerve being dominated). We then identify such columns using the same criterion [*], and 'collapse' them (remove these columns). We use the same term here also : 'collapse', because MaxSimplices are nothing but vertices of the Nerve. This step corresponds to taking the second Nerve (finding ALL "dominated" columns in the Matrix and removing them).

In a nutshell : We create the matrix using the DFS strategy. Initially, rows may be collapsible but all columns will be non-collapsible (because to begin with, all columns contain maximal simplices, and the fact that they are all maximal means none of them is collapsible).
Then, we find ALL collapsible rows, remove them. This creates the possibility of some columns being collapsible. Find ALL collapsible columns, remove them. This constitutes one step of the collapse.
This may further create the possibility of some rows being collapsible. Continue these row-column operations until no row is collapsible. This would indicate that our matrix now represents the core of the original Simplicial Complex. [Collapsible here means the same as Dominated]

7 Implementational Details

1. **Two Lists Heuristic** : The naive way of doing the row operations would be to check for each *ORDERED* pair of rows, whether one dominates the other. The same goes for columns. This heuristic tries to improve on that.

Firstly, we adopt a strategy to prevent checking through all pairs. [I am writing this for vertices (rows), but the same goes on for MaxSimplices (columns) also]. The idea is to go through the vertices in a sorted manner (sorted on the number of MaxSimplices of which that vertex is a part of). This is because a

row y can dominate another row x only if the number of Maximal Simplices of which y is a part of, is more than the corresponding number for x.

So, what we do is maintain a list of these vertices, sorted in a descending order based on the mentioned criterion. We go from left to right, and for each[1] element of this list, we check if it is dominated by any previous member of the list. If yes, remove this element from the list.
The reason we are using lists is precisely because of this remove operation. Lists offer constant time removal of entries with the cost of losing random access of elements. And we do not want random access. We always access elements sequentially. And so, lists are best for this purpose.

[1] This 'each' will change after the second strategy.

Secondly (and maybe more importantly), removal of a row (vertex) is a ***local operation***. It affects only those columns (MaxSimplices) of which that vertex was a part of. And only these are possible candidates for removal in the subsequent round of column collapse. Hence, removal of one row should not mean we have to check all the pairs of MaxSimplices for redundant (some MaxSimplex is no longer maximal) columns that have to be removed.
And so with each element of the list, we keep a boolean variable that indicates whether this element has to be checked in this particular step of collapse. And this boolean variable is kept *true* only if the previous *half-step*[2] caused some change in this element.

[2] A half-step for the MaxSimplices list is the vertices list operation of the same collapse step. A half-step for the vertices list is the MaxSimplices list operation of the previous collapse step.

Combining these two strategies, we developed the following method. We create two lists, one for the vertices and one for the MaxSimplices. The elements of the lists are tuples of the type *(integer, integer, boolean)*.
For vertices, the tuple represents :
(no. of MaxSimplices of which this vertex is a part of, row number in the matrix, does this have to be checked in this step)
For simplices, the tuple represents :
(no. of vertices that are a part of this MaxSimplex, column number in the matrix, does this have to be checked in this step)

The boolean values for the vertices list are all initialised to *true*. The boolean values for the MaxSimplices list are all initialised to *false*. That's because all vertices are candidates for collapse and have to be checked in the first step. And that all MaxSimplices by the definition of being maximal, are not needed to be checked currently for possible collapses.
Then, we perform operations as explained in the first strategy. The crucial change is that instead of that 'each', we now only do that step for elements that have *true* value in the boolean variable.

2. Instead of actually removing rows/columns from the matrix, we use active/inactive states for them, because removal takes more time than "turning off" a row/column. So basically, in constant time, we can toggle the state of a row/column from active to inactive. This is good because we are actually performing the operations on the lists. The matrix is just to refer to the boolean values it stores.

3. The check on rows and columns is complementary. We do not need to write separate pieces of code, as they are exactly similar operations.

8 Further Possibilities

1. *Exploitation of the implicit structure in the Matrix*
The MaxSimplices in the matrix have an intrinsic order (because of the way they are discovered : the DFS). It cannot be the case that a MaxSimplex with vertex 0 comes after one that does not contain the vertex 0. Basically, the topmost 'ticks' of the columns must form a strictly non-increasing sequence in terms of position in the matrix. It would be interesting if there is some intelligent optimisation possible because of this.

2. ***[IMP]*** *Sparsification of Matrix*
The current code uses a completely dense Matrix for Representation of the Ms-Matrix in the Computer.
It's best that this code be changed to use a sparse matrix, most probably using a C++ package like Eigen.

3. *Optimisations possible in the present code*
There may very well be some optimisations possible in the existing code that I might have not figured out.
One of them is the use of BOOST data structures and iterators instead of STL structures and iterators.