Homology Computations via Acyclic Subspace

Piotr Brendel[1], Paweł Dłotko[1,⋆], Marian Mrozek[1], and Natalia Żelazna[2]

[1] Institute of Computer Science, Jagiellonian University
{piotr.brendel,pawel.dlotko,marian.mrozek}@ii.uj.edu.pl
[2] Motorola Solutions
natalia.zelazna@motorolasolutions.com

Abstract. Homology computations recently gain vivid attention in science. New methods, enabling fast and memory efficient computations are needed in order to process large simplicial complexes. In this paper we present the acyclic subspace reduction algorithm adapted to simplicial complexes. It provides fast and memory efficient preprocessing of the given data. A variant of the method for distributed computations is also presented. As a result, Betti numbers can be effectively computed.

Keywords: Homology algorithms, reduction algorithms, acyclic subspace method.

1 Introduction

The classical way of computing homology consists in finding the Smith Normal Form of the matrix of the boundary map [9]. The complexity of the Smith Normal Form algorithm is supercubical. This is prohibitive in applications where the size of the boundary map matrix is large, in particular in rigorous numerics of dynamical systems, problems in image recognition, data analysis, material science, electromagnetism, robotics, theoretical computer science, ecology, molecular biology and other areas. Therefore, in recent years several methods have been proposed to speed up homology computations, particularly computations of the homology of sets in various representations. Among such methods are geometric reduction algorithms. They aim at finding a smaller representation with the same homology as the original set. A method of this type, recently proposed in [8], is based on the construction of an acyclic subset. We recall that a simplicial complex $\mathcal{A}$ is acyclic iff the homology of $\mathcal{A}$ is isomorphic to the homology of a point. In this paper a *component-wise acyclic* subcomplex $\mathcal{A}$ of a simplicial complex $\mathcal{S}$ is a simplicial complex whose connected components are acyclic subsets of corresponding connected components of $\mathcal{S}$. The acyclic subspace algorithm is based on the simple observation that if $\mathcal{A}$ is an acyclic subcomplex of a simplicial complex $\mathcal{S}$, then:

$$H_n(\mathcal{S}) \cong \begin{cases} H_n(\mathcal{S},\mathcal{A}) & \text{for } n \geq 1 \\ \mathbb{Z} & \text{for } n = 0 \end{cases}$$

⋆ Corresponding author.

M. Ferri et al. (Eds.): CTIC 2012, LNCS 7309, pp. 117–127, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Note that due to the excision theorem [9] the relative homology $H_n(\mathcal{S}, \mathcal{A})$ depends only on the neighborhood of $\mathcal{S} \setminus \mathcal{A}$ in $\mathcal{S}$. Therefore, if a large acyclic subcomplex $\mathcal{A}$ of $\mathcal{S}$ may be constructed quickly, the problem of finding the homology of $\mathcal{S}$ is reduced to a relatively small set and consequently may be found quickly. The aim of [8] was to show that in the case of cubical complexes a relatively large acyclic subcomplex may be found in linear time.

The goal of the presented paper is to extend the ideas of [8] to the case of simplicial complexes. In particular, we propose two fast algorithms constructing a possibly large acyclic subcomplex $\mathcal{A}$ of every connected component of a simplicial complex $\mathcal{S}$. Moreover, we show how to extend this algorithms for the purposes of distributed computations.

Acyclic subset reduction leads to more efficient computation of Betti numbers, useful in image recognition.

2 Preliminaries

For the purposes of this paper a finite family of finite sets $\mathcal{S}$ is called an *abstract simplicial complex* if for every $P \in \mathcal{S}$ and for every $Q \subset P$ we have $Q \in \mathcal{S}$. An element $P \in \mathcal{S}$ is called a *simplex*. If $P \in \mathcal{S}$ and $Q \subset P$ then Q is called a *face* of P. A simplex $P \in \mathcal{S}$ is said to be *maximal* if there is no simplex $Q \in \mathcal{S}$ such that $P \subsetneq Q$. Throughout this paper $S_{max}(\mathcal{S})$ denotes the set of maximal simplices of $\mathcal{S}$. The *algebraic closure* of a simplex P, denoted by $cl(P)$ is a family of simplices consisting of P and all its faces. The closure of a family of simplices $\mathcal{K}$ is $cl(\mathcal{K}) = \bigcup_{P \in \mathcal{K}} cl(P)$. For a simplex $Q \in \mathcal{S}$ its *neighborhood* consists of all maximal simplices in $\mathcal{S}$ whose intersection with Q is nonempty . We denote this set by

$$n(Q) = \{P \in \mathcal{S} \mid Q \cap P \neq \emptyset \text{ and } P \text{ is a maximal in } \mathcal{S}\}.$$

By the dimension of a simplex P we mean $dim(P) := card(P) - 1$. For a simplicial complex $\mathcal{S}$ by $S_0(\mathcal{S})$ we denote the set of all the vertices of $\mathcal{S}$, i.e. its 0-dimensional simplices, and we make a technical assumption that every vertex in S_0 has a unique label. In the sequel, we use a hash table [1], denoted H, whose keys are labels of vertices and for each key the value is the list of all maximal simplices containing the vertex labeled with the given key.

The main homological tools used in the paper are the exact sequence of a pair and the Mayer-Vietoris sequence [9]. The exact sequence of a pair is used to conclude that a subcomplex with trivial reduced homology can be removed from the initial complex without changing its reduced homology. From the Mayer-Vietoris sequence it follows, that a simplex can be added to the constructed acyclic subcomplex if and only if its intersection with the acyclic subcomplex has trivial reduced homology.

3 Incidence Graph

We say that a graph $G = (V, E)$ is an *incidence graph* of a simplicial complex $\mathcal{S}$ if V is the set of maximal simplices of $\mathcal{S}$ and $(S_1, S_2) \in E$ iff $S_1 \cap S_2 \neq \emptyset$.

An *augmented incidence graph* is a triple (V, E, C) where (V, E) is the incidence graph and C is the list of connected components of incidence graph, in which each connected component is represented by a single maximal simplex from this component. We will use augmented incidence graphs to retrieve all the information about neighborhoods in a simplicial complex, necessary in the process of constructing an acyclic subset.

In this section we show an algorithm constructing such a graph for a given simplicial complex. The input data for this algorithm is the list of maximal simplices $S_{max}(\mathcal{S})$ and VertexHash H, described in Section 2. For each vertex v we consider the list $H[v]$ storing the maximal simplices that contain v. $\mathcal{Q}$ denotes the queue used to store simplices which have not yet been added to the incidence graph and whose neighbors are already there. Functions `Enqueue` and `Dequeue` are standard operations on queues and their description can be found in [1].

Algorithm 3.1. IncidenceGraph(MaximalSimplexList $S_{max}(\mathcal{S})$, VertexHash H)

```
1: V := ∅; E := ∅; C := ∅; Q := EmptyQueue;
2: for all Simplex P ∈ S_max(S) do
3:     if P ∉ V then
4:         C := C ∪ {P};
5:         Enqueue(Q, P);
6:         while Q ≠ ∅ do
7:             Simplex current := Dequeue(Q);
8:             V := V ∪ {current};
9:             for all Vertex v ∈ current do
10:                for all Simplex neighbour ∈ H[v], neighbour ≠ current do
11:                    if neighbour ∉ V then
12:                        e := (current, neighbour); E := E ∪ {e};
13:                        if neighbour ∉ Q then
14:                            Enqueue(Q, neighbour);
15: return Graph(V, E, C);
```

Theorem 3.1. Algorithm 3.1 stops and constructs the augmented incidence graph $G = (V, E, C)$ for simplicial complex $\mathcal{S}$ in $O(card(V) \cdot dim(\mathcal{S}) \cdot deg(H))$ where $dim(S) = max_{P \in \mathcal{S}}\{dim(P)\}$ and $deg(H) = max_{v \in S_0(\mathcal{S})}\{length(H[v])\}$. Moreover, for each connected component $G' \subset G$ its set of nodes $V(G')$ equals to the set of maximal simplices in the corresponding connected component $\mathcal{S}' \subset \mathcal{S}$.

Proof. Obviously V contains all maximal simplices in $S_{max}(\mathcal{S})$. Pair $(S_1, S_2) \in E$ iff $S_1 \cap S_2 \neq \emptyset$, therefore augmented incidence graph is obtained. Simplex P is added to C in line 4 only if $P \notin V$ which means $P \cap S = \emptyset$ for all $S \in V$ and P represents a new connected component, since in **while** loop at line 6 BFS procedure, which finds connected components, is implemented. Simplex P is added to $\mathcal{Q}$ only once, hence the **while** loop in line 6 always completes after adding to V all elements from connected component of P.

The internal **for all** loop in line 9 is performed for every d-dimensional simplex at most $dim(\mathcal{S}) \cdot deg(H)$ times. Since the **while** loop in line 6 is performed at most $card(V)$ times, the complexity of the algorithm is $O(card(V) \cdot dim(\mathcal{S}) \cdot deg(H))$. □

4 Constructing Acyclic Subset

Since the homology of a disconnected complex is a direct sum of homologies of its connected components, later in this paper we will construct component-wise acyclic subcomplexes. In this section we present two approaches to construction of such complexes using the incidence graph as well as a function named `AcyclicityTest`. The purpose of this function is to decide whether a simplex may be added to the constructed acyclic set without loosing acyclicity. How to obtain such a function is the purpose of Section 6.

The first algorithm, referenced in the following sections as `AccST`, is an adaptation of the algorithm presented in [8] to the case of simplicial complexes. The adaptation is not straightforward, because, unlike the case of cubical sets, in the simplicial case it is not obvious how to efficiently determine the neighborhood of a simplex. For this, we use the incidence graph presented in Section 3. Another difference is that, instead of one, we construct several acyclic subsets in each connected component of $\mathcal{S}$ and then join them by a spanning tree. It allows us to construct larger acyclic subsets for certain kinds of data. To do this we need two auxiliary functions: `FindSimplexNotInAccSub` and `CreateSpanningTree`. Both are based on standard graph algorithms [1]. The first finds a simplex that has no intersection with the acyclic subset. It uses breadth-first search algorithm. If no such simplex can be found, it returns `NULL`. The other takes as an input a list of simplices, one per each constructed acyclic subset. It first constructs the shortest paths connecting the acyclic subsets. Each path is a list of one-dimensional simplices. The paths are used to build a graph in which nodes are the constructed disjoint acyclic subsets and edges are the constructed paths. Then, Kruskal algorithm [1] is applied to create a spanning tree joining the constructed acyclic subsets.

Theorem 4.1. Algorithm 4.1 always stops. Given a simplicial complex $\mathcal{S}$ on input, represented by the incidence graph $G = (V, E, C)$, it returns a component-wise acyclic complex $\mathcal{A}$ on output.

Proof. In lines 11 and 12 a simplex P is added simultaneously to $\mathcal{Q}$ and to the acyclic subset $\mathcal{A}$. Since P may be added to $\mathcal{A}$ only once and the number of simplices is finite, the inner `while` loop in line 7 always finishes. Functions `FindSimplexNotInAccSub` and `CreateSpanningTree` are respectively BFS and Kruskal algorithms [1], hence they both complete. Every simplex found by `FindSimplexNotInAccSub` in line 13 is added to the acyclic subset. Hence, the finiteness of V implies that the `while` loop in line 4 completes. Thus, since the number of simplices in C is also finite, we know that the algorithm always stops.

Algorithm 4.1. AccST(IncidenceGraph (V, E, C))

```
 1: A := ∅; Q := EmptyQueue;
 2: for all Simplex P ∈ C do
 3:       L :=EmptyList;
 4:       while P ≠ ∅ do
 5:            A := A ∪ {P};
 6:            Enqueue(Q, P);
 7:            while Q ≠ ∅ do
 8:                 Simplex Q := Dequeue(Q);
 9:                 for all Simplex S ∈ n(Q) \ A do
10:                      if AcyclicityTest(A, S) = true then
11:                           A := A ∪ {S};
12:                           Enqueue(Q, S);
13:            P :=FindSimplexNotInAccSub(V, E, P, A);
14:            if P ≠ NULL then
15:                 L := L ∪ {P};
16:       A := A ∪ CreateSpanningTree(L);
17: return A;
```

Since each simplex is acyclic, we begin the construction of the acyclic subsets of the components of $\mathcal{S}$ with the representants of the connected components of the incidence graph described in Section 3. As long as we can find a simplex acyclically intersecting $\mathcal{A}$, by Mayer-Vietoris Theorem we may add it to $\mathcal{A}$ without losing its acyclicity. If we cannot find such a simplex, we look in the same connected component for another one that has no intersection with $\mathcal{A}$ and we build acyclic subset around it as described above. We stop this procedure when there are no simplices that do not intersect $\mathcal{A}$. Due to Mayer-Vietoris Theorem acyclic subsets constructed that way cannot intersect each other. Now let us assume, we have a number of disjoint acyclic subsets of $\mathcal{A}$ and we want to connect them in order to form a larger acyclic subset. First, we need to find paths, i.e. lists of one-dimensional simplices, joining the subsets. Since all components of the constructed set $\mathcal{A}$ are contained in the same connected component of $\mathcal{S}$, we can always find a path joining any two of them. Unfortunately, the constructed paths can intersect each other or even other parts of $\mathcal{A}$ creating unwanted cycles. Nevertheless, is is not difficult to avoid this problem by joining acyclic parts step by step and adding only parts of the connecting paths so as not to lose acyclicity. □

Algorithm 4.2, referenced in the following sections as `AccIG`, constructs simultaneously the incidence graph and a component-wise acyclic complex. For certain kinds of data it provides faster and more memory efficient way of constructing component-wise acyclic subcomplex than Algorithm 4.1. The nodes of the resulting graph G are these simplices in $S_{max}(\mathcal{S})$ which are not in the acyclic subset. The algorithm uses the general graph functions `AddToGraph` and `RemoveFromGraph`, respectively adding a simplex to or removing a simplex from

Algorithm 4.2. AccIG(MaximalSimplexList $S_{max}(\mathcal{S})$, VertexHash H)

```
1: V := ∅; E := ∅; A := ∅; Q := EmptyQueue;
2: for all Simplex P ∈ S_max(S) do
3:     if P ∉ V and P ∉ A then
4:         A := A ∪ {P};
5:         EnqNeighb(P, H, Q);
6:         while Q ≠ ∅ do
7:             Simplex current := Dequeue(Q);
8:             if AcyclicityTest(A, current) = true then
9:                 A := A ∪ {current};
10:                EnqNeighb(current, H, Q);
11:                if current ∈ V then
12:                    RemoveFromGraph(current, V, E);
13:            else if current ∉ V then
14:                AddToGraph(current, V, E, H);
15:                EnqNeighb(current, H, Q);
16: return Graph(V, E), A;
```

a given graph. It also uses function `EnqNeighb`, which enqueues all neighbors of given simplex that are not yet in the queue nor in the acyclic subset.

Theorem 4.2. Algorithm 4.2 stops and returns a component-wise acyclic complex $\mathcal{A}$ and the incidence graph $G = (V, E)$ whose nodes are the maximal simplices in $S_{max}(\mathcal{S}) \setminus \mathcal{A}$.

Proof. A simplex may be added to $\mathcal{Q}$ only if it does not belong to the acyclic subset and its neighbor has been added to the incidence graph or the acyclic subset. Since each simplex can be added to the graph or the acyclic subset at most once, the algorithm stops.

We start building a new acyclic component of the set $\mathcal{A}$ by finding a simplex that has not been added yet neither to the graph nor to the acyclic subset. Thus, it is not a neighbor of any simplex already processed. It means it represents new connected component of $\mathcal{S}$ in which we can start build new acyclic subset $\mathcal{A}$. We extend it only by adding those maximal simplices that have acyclic intersection with $\mathcal{A}$. For every simplex in $\mathcal{Q}$ we either add it to the acyclic subset or to the incidence graph, which means that nodes of the created incidence graph are all maximal simplices of $S_{max}(\mathcal{S})$ that have not been added to $\mathcal{A}$. □

5 Distributed Computations

In this section we show how the algorithms that compute a component-wise acyclic subcomplex for a given simplicial complex can be used in distributed computations. The idea is to divide the initial complex into small parts, then construct a component-wise acyclic subcomplex and an incidence graph for each part and finally combine the results into a component-wise acyclic subcomplex and incidence graph for the initial complex. However, we need to ensure that

after combining the results from the individual computations the obtained space is component-wise acyclic, i.e. we do not make cycles while connecting the acyclic subsets from the different parts. Moreover, we need a way to connect individual incidence graphs into the incidence graph of the initial complex. The whole procedure is very technical and resembles what we do in Algorithm 4.1 but in a more global scale. Let us emphasize that distributed computations involve only the construction of the incidence graph and the acyclic subset for each part. After combining the results from the individual reductions we create one complex for which we can perform homology computations just like in the non-distributed case.

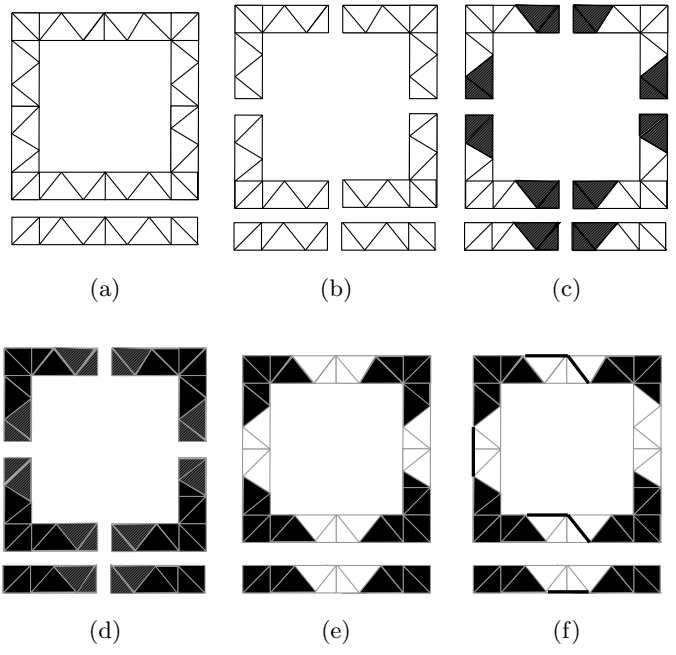

Fig. 1. (a) the initial simplicial complex, (b) the initial complex splitted into smaller, partial complexes, (c) the boundary simplices in the partial complexes, (d) the acyclic subsets in the partial complexes (black), (e) the combined results, (f) the acyclic subsets joined with a spanning forest (black)

The first step is to split the initial list of maximal simplices of $\mathcal{S}$ (Figure 1a) into lists $\mathcal{P}_i$, $i \in \{1, 2, ..., n\}$ in such a way that $\bigcup_i \mathcal{P}_i = S_{max}(\mathcal{S})$ (Figure 1b) and $\mathcal{P}_i \cap \mathcal{P}_j = \emptyset$ if $i \neq j$. For every $\mathcal{P}_i$ we define two sets: $BV_i := \bigcup_{i \neq j} \{S_0(\mathcal{P}_i) \cap S_0(\mathcal{P}_j)\}$ and $BS_i := \{Q \,|\, Q \in \mathcal{P}_i \wedge S_0(Q) \cap BV_i \neq \emptyset\}$. The elements of BS_i are referred to as the *boundary simplices* - simplices which have neighborhood contained in other packages. (Figure 1c). In the process of constructing the

acyclic subset $\mathcal{A}_i$ for each $\mathcal{P}_i$ we consider only those simplices that are not boundary simplices (Figure 1d). To do so, we need to change a little Algorithms 4.1 and 4.2 so they include such restriction. We will not present them here, but it is easy for the reader to do such modification. In our example (Figure 1d) acyclic subset is constructed from all simplices that are not boundary simplices, but in general case this is not true. Computations of lists of both incidence graphs G_i and acyclic subsets $\mathcal{A}_i$ may be performed sequentially or in a distributed manner. In both cases we gain profits from lower memory usage, because list of simplices for which computations are performed are much smaller than the initial one. In the second case computations are performed much faster. Moreover, after constructing the acyclic subsets $\mathcal{A}_i$ we can discard all simplices contained in $\mathcal{A}_i$ from the incidence graph and construct a new acyclic subset which is the intersection of $\mathcal{A}_i$ with the lower dimensional faces of simplices which are left in the incidence graph. In the latter case, we save additional memory needed to store redundant simplices. Finally, after combining the results (Figure 1e) into one incidence graph we obtain a structure analogous to the one in Algorithm 4.1. We then create a spanning forest in which nodes are disjoint parts of the acyclic subset and edges are lists of one-dimensional simplices connecting them (Figure 1f).

Theorem 5.1. The family of simplices $\mathcal{A}$ constructed as above is a componentwise acyclic subcomplex of the initial simplicial complex $\mathcal{S}$.

Proof. By restricting the acyclic subset algorithm to these simplices that are not boundary simplices we are sure that the acyclic subsets in the respective parts do not create cycles after combining them. The rest of the proof is analogous to the proof of Theorem 4.1. □

6 Acyclicity Tests

The `AcyclicityTest` function is a tool allowing to decide whether we can add a simplex to the constructed acyclic subset. The function takes two arguments: the already constructed acyclic subset $\mathcal{A}$ and a simplex P. We distinguish two types of acyclicity tests:

- a *full test* – it returns `true` if and only if $\mathcal{A} \cap cl(P)$ is acyclic
- a *partial test* – if it returns `true`, then $\mathcal{A} \cap cl(P)$ is acyclic but `false` on output denotes a failure to prove that $\mathcal{A} \cap cl(P)$ is acyclic.

Acyclicity tests in the setting of cubical sets, both full and partial, are proposed in [8].

The main limitation for quick acyclicity tests is the dimension of the complex. The full tests both in the cubical [8] and in the simplicial case are based on the idea of *tabulated configurations* for boundary elements. The number of configurations is 2^{3^d-1} for a d-dimensional cube and $2^{2^{d+1}}$ for a d-dimensional simplex. This makes the method prohibitive for $d > 3$ in the case of cubical sets [8] and for

$d > 4$ in the case of simplicial complexes [3]. The universal full test that works for every dimension is the computation of the homology of $\mathcal{A} \cap cl(P)$. However, this method is computationally very expensive. Therefore, in the dimensions where the tabulated configurations cannot be used, the use of quick partial tests is of interest. An acyclic subset algorithm based on partial acyclicity tests remains correct and often provides acyclic subsets which are not substantially smaller or even the same size as the algorithms based on full tests.

In the rest of this section we introduce few partial tests for the simplicial case. Given an acyclic subspace $\mathcal{A}$ and a d-dimensional simplex P we set $\mathcal{I} := \mathcal{A} \cap cl(P)$. The first test is based on the investigation of the maximal simplices of $\mathcal{I}$. It is straightforward to check that if the number of maximal simplices of dimension $d-1$ in $\mathcal{I}$ is less than or equal to d and there are no maximal simplices of other dimensions in $\mathcal{I}$, then $\mathcal{I}$ is acyclic. This proves the following theorem.

Algorithm 6.1. AccTestCoDim1(Set $\mathcal{A}$, Simplex P)

```
1: I := MaximalSimplices(A ∩ cl(P));
2: d :=Dim(P); i := 0;
3: for all Simplex Q ∈ I do
4:       if Dim(Q) = d − 1 then
5:             i++;
6:       else
7:             return false;
8: if i > 0 and i <= d  then
9:       return true;
10: else
11:      return false;
```

Theorem 6.1. Given a set $\mathcal{A}$ and simplex P on input, if Algorithm 6.1 returns `true`, then $\mathcal{A} \cap cl(P)$ is acyclic. However, `false` on output denotes a failure to decide whether $\mathcal{A} \cap cl(P)$ is acyclic.

Algorithm 6.2 tries to find a vertex of P which is a common face of all maximal simplices in $\mathcal{I}$. If it is able to do so, it means that $\mathcal{I}$ forms a topology of a star and therefore is acyclic. Analogous theorem as for Algorithm 6.1 can be stated for Algorithm 6.2.

Two more partial tests will be introduced without presenting suitable algorithms and theorems. First of them uses the list of maximal simplices $\mathcal{I}$ for construction of an acyclic subcomplex $\mathcal{I}'$ of $\mathcal{I}$. The whole procedure is exactly the same as presented in this paper. For testing acyclicity it uses itself recursively. At the bottom of recursion we only need to determine if the intersection of two one dimensional simplices is acyclic. In fact it is true if and only if simplices share common vertex, which is trivial to check. If constructed acyclic subcomplex $\mathcal{I}'$ is the same as the initial complex $\mathcal{I}$, then $\mathcal{I}$ is acyclic.

The last test constructs simplicial complex from $\mathcal{I}$ and performs coreductions [7] on it. If the resulted complex is fully reduced, it means that $\mathcal{I}$ is acyclic.

Algorithm 6.2. AccTestStar(Set $\mathcal{A}$, Simplex P)

```
1: I := MaximalSimplices(A ∩ cl(P));
2: for all Vertex v ∈ P do
3:       ok := true;
4:       for all Simplex Q ∈ I do
5:            if v ⊊ Q then
6:                 ok := false;
7:                 break;
8:       if ok then
9:            return true;
10: return false;
```

7 Numerical Experiments

All algorithms presented in this paper have been implemented in `C++`. The code will be available as a part of RedHom [11] library. To provide a communication between processes during distributed computation `MPI` [4] was used. Both local and distributed approaches were compared with the coreduction homology algorithm [7], denoted in the following table by `CoRed`. `AccIG` and `AccST` are the algorithms introduced in Section 4. `DAccIG` and `DAccST` denote the outcome of distributed computations using `AccIG` and `AccST` algorithms respectively for a local construction of an acyclic subspace. Column `size` denotes the number of maximal simplices used as input. Value in column `s` is the total time in seconds needed for building the incidence graph, performing reductions (which could be either removal of acyclic subset or coreductions [7]), creating the simplicial complex from the list of maximal simplices and computing Betti numbers for the complex [5]. Column `MB` contains the total amount of memory in megabytes needed for performing computations. Distributed computations were performed on 4 nodes (1 master and 3 slaves) simultaneously and values in this case denotes the maximum running time in seconds over all nodes and maximum amount of memory that single node needs. Computing generators after reduction of acyclic subset is still an open problem.

		CoRed		AccIG		AccST		DAccIG		AccST	
Space name	Size	s	MB	s	MB	s	MB	s	MB	s	MB
Bjorner	3079k	174	2330	203	1035	375	2968	154	568	229	1008
Dunce Hat	4758k	273	3603	327	1647	598	4525	224	821	390	1527
Proj. Plane	2799k	158	2180	189	943	323	2638	143	437	190	900

The second table contains comparison of efficiency of acyclicity test algorithms presented in Section 6. Algorithm denoted as `Tab` is acyclicity test that uses tabulated configurations. `CoDim1` and `Star` are respectively Algorithms 6.1 and 6.2. `Rec` is recursive test, `Hom` is full test that uses homology computations and `Cored` is test based on coreductions. For each algorithm column `s` denotes total

Space name	Size	Tab		CoDim1		Star		Rec		Hom		Cored	
		s	#	s	#	s	#	s	#	s	#	s	#
Bjorner	513216	33	513215	49	513215	34	513215	39	513215	166	513215	124	513215
Dunce Hat	793152	53	789279	73	789060	50	789279	55	789279	253	789279	192	789279
Proj. Plane	466560	29	464511	42	464609	31	464511	32	464511	150	464511	111	464511

running time in seconds needed for performing computations, just as described above, using `AccIG` algorithm and selected acyclicity test. Column `#` denotes number of maximal simplices in constructed acyclic subcomplex.

Acknowledgments. P.B. and M.M. are partially supported by Polish MNSzW, Grant N N201 419639. P.D. is partially supported by grant Nr IP 2010 046370.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press and McGraw-Hill (1990)
2. Dłotko, P., Specogna, R.: Efficient Cohomology Computation for Electromagnetic Modeling. Computer Modeling in Engineering & Sciences 60(3), 247–277 (2010)
3. Dłotko, P.: Acyclic configurations for boundary elements of 3 and 4 dimensional simplices, `http://www.ii.uj.edu.pl/~dlotko/accconf.html`
4. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press (1990)
5. Kaczynski, T., Mischaikow, K., Mrozek, M.: Computational homology, Appl. Math. Sci., vol. 157. Springer, New York (2004)
6. Kaczynski, T., Mrozek, M., Ślusarek, M.: Homology computation by reduction of chain complexes. Computers and Math. Appl. 35, 59–70 (1998)
7. Mrozek, M., Batko, B.: Coreduction Homology Algorithm. Discrete and Computational Geometry 41, 96–118 (2009)
8. Mrozek, M., Pilarczyk, P., Żelazna, N.: Homology algorithm based on acyclic subspace. Computers and Mathematics with Applications 55, 2395–2412 (2008)
9. Munkres, J.R.: Elements of Algebraic Topology. Perseus Publishing, Cambridge (1984)
10. Computer Assisted Proofs in Dynamics, `http://capd.wsb-nlu.edu.pl`
11. The RedHom homology algorithms library, `http://redhom.ii.uj.edu.pl`