The Topological Structure of the Renaming Problem

Paavan Bhavsar

Background

The renaming problem in distributed systems tackles the problem of "renaming" a finite set of processors using a finite set of "names". The specific problem that is dealt with in this paper is as follows: suppose there are n processors each with a unique "name" $k_i \in K$ where $|K| >> n$. Instead of wasting much of K, we instead wish for each processor to draw from a set of only $n+1$ names. Various renaming algorithms exist which allow the n processors to asynchronously rename themselves such that each processor chooses a unique name. Instead of attempting to develop a competing algorithm, we instead interest ourselves with the topological structure of this problem, using simplicial homology theory.

Let us start by considering the renaming problem for 3 processors. Each processor will be assigned a number between 1 and 3 to serve as its invariant id. Then each computer will choose a name, either A, B, C, or D. Suppose that an algorithm is applied and processor 1 choose A, processor 2 chooses B, and processor 3 chooses C. This is a possible final state of the renaming problem. We can represent this final state in a 2-simplex as depicted in Figure 1. We use the following convention: a vertex labeled $1A$ is interpreted as "processor 1 has taken name A".

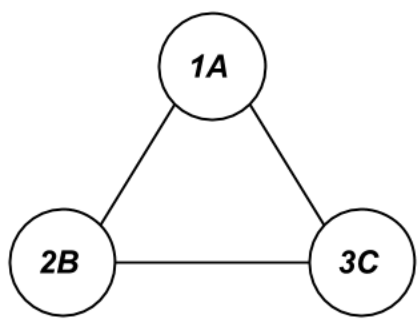

Figure 1: Basic 2-Simplex

Now suppose that instead of B, processor 2 instead chooses to take the name D. We can represent this final state by "flipping" the vertex $2B$ over the 1-simplex ($1A$ $3C$). This results in mesh depicted in Figure 2. This mesh represents two final states: one with $1A$, $2B$, and $3C$, and another state with $1A$, $2D$, and $3C$.

It is then trivial to see how we can "unfold" the structure of this problem by repeatedly flipping vertices over available axes. After all possible final states have been represented once, we can be sure that we have a mesh that represents the complete structure of the renaming problem. However,

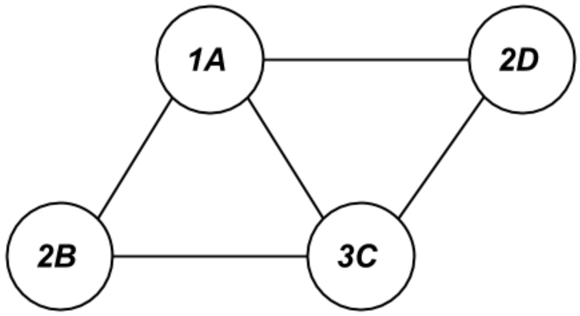

Figure 2: Basic Mesh

in order to represent the boundaries of this mesh, we must also take into account the orientations of its simplices. If we always notate simplices in processor-order, then we find that calculating orientations is trivial. If a simplex has orientation $a \in \{1, -1\}$, then all simplices adjacent to it will have orientation $-a$. That is, every time we "flip" a vertex, the resulting new simplex will take the opposite orientation of the original simplex. This property is depicted in Figure 3.

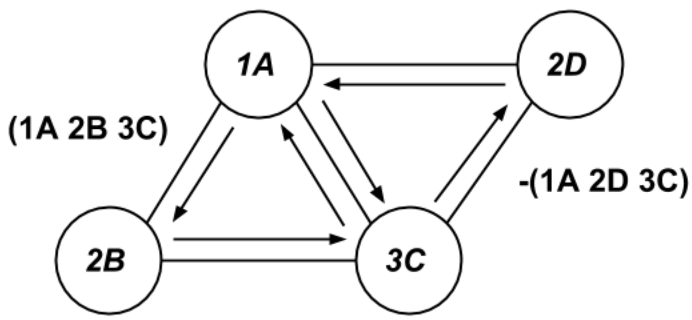

Figure 3: Basic Oriented Mesh

With this logic in place we can represent the renaming problem for any number of processors in a simplicial mesh.

Algorithm

The algorithm works by creating the boundary matrices for each dimension of the mesh, and then manipulating them to derive the homology groups and Betti numbers of the structure. This is done at a high level in *calculate_homologies*.

In our earlier example we had each processor identified by a number and take a letter. In the actual implementation of the algorithm, each vertex is identified by just one number that encompasses both of these ideas (instead of letters we use numbers for names). The intuition is simple: the n^{th} processor is correlated with the n^{th} group of $n+1$ vertex numbers. For example, in the case of 3 processors, the first processor is correlated with vertex numbers 1, 2, 3, and 4; each of these correspond to processor 1 taking the names 1, 2, 3, and 4, respectively. Processor 2 takes vertices 5-8, and processor 3 takes vertices 9-12. This way we have an easily extensible way to handle any number of processors, while making it easy to decompose a vertex's number into its corresponding processor number and name. See Figure 4 for an example of a mesh using this numbering system.

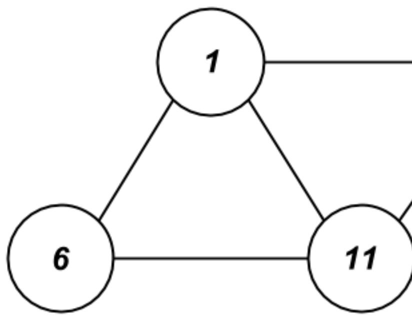

Figure 4: Basic mesh equivalent to Figure 2, with vertices labeled using the numbering system instead of id-name combinations.

Creating the Boundary Matrices

In order to populate ∂_k, the algorithm first calculates the full list of k and $(k-1)$-simplices that will be used to key the columns and rows of the boundary matrix. Since we wish to depict every possible final state in this matrix, we must include every possible k-simplex with its boundary. This is done in *simplex_generator*, which, given k and the number of processors, outputs a matrix with every possible (valid) k-simplex. By valid, we mean to say that each vertex in a k-simplex corresponds to a unique processor taking a unique name. Thus, simplices such as (1 5 9) do not appear (the issue in that simplex is that all three processors are taking the first name).

The output of *simplex_generator* is a matrix in which each row is a valid k-simplex. The function is used to create two matrices called *columnSimplices* and *rowSimplices*, which are the lists of k-simplices and $(k-1)$-simplices, respectively. These two matrices will be parallel to ∂_k's columns and rows: the first column of ∂_k is associated with the first row of *columnSimplices*, and the first row of ∂_k is associated with the first row of *rowSimplices*.

Given n processors, there are two cases to consider when populating ∂_k. When $k = n-1$, we must also take into account the orientation of the boundaries. This boundary matrix is more costly to compute, since it must be incrementally generated by repeatedly flipping vertices.

Generating the Oriented Surface

In order to create the boundary matrix for the oriented surface, we create two more matrices called *axes* and *orientations*.

- *axes* is parallel to *rowSimplices*, and will be used to quickly find available "axes" for vertices to flip over. For example, in Figure 3, the vertex $2B$ was flipped over the axis represented by the simplex $(1A\ \ 3C)$.

- *orientations* is parallel to *columnSimplices*, and simply stores the orientation of its associated simplex in *columnSimplices* (1, 0, or -1).

With this framework in place, we can begin to populate ∂_k, where we have $n = k+1$ processors.

First, we prepopulate the matrix with the starting k-simplex, representing the final state in which the first processor takes the first name, the second processor takes the second name, etc. To do this, we find the column that is associated with this k-simplex and use the *getBoundary* function to find the $(k-1)$-simplices in its boundary and their orientations. Since the starting simplex has orientation 1, we simply populate the entry at the appropriate row and column with the orientation of that $(k-1)$-simplex in the boundary of the k-simplex.

Since each $(k-1)$-simplex represents a possible axis to flip over, we add its orientation to the *axes* matrix at the appropriate row.

Now we can enter a loop. Every nonzero entry in *axes* represents a $(k-1)$-simplex that we can flip over. We pick one such $(k-1)$-simplex and scan its row in ∂_k to find the k-simplex it is a part of (since it is an available axis, there is only one such k-simplex). We then compare the two simplices to determine which processor number is "left out"; that is, which processor is in the k-simplex but not in the $(k-1)$ simplex. This is the processor that will flip over the axes. In a similar fashion, we determine which name this processor will take. With these two pieces of information, we can determine the final vertex number after the flip is done, and therefore the final k-simplex that we will add to ∂_k. We use *orientations* to find the orientation of the original k-simplex so that the new k-simplex can have the opposite oriention. Then, we add this new k-simplex to ∂_k using *getBoundary* and update *orientations* and *axes*. Note that *axes* effectively stores the orientation of each $(k-1)$-simplex in the mesh. Since the axis we flipped over now appears twice in ∂_k with opposite orientations, its value in *axes* is 0 and we need not worry about considering it again.

Let's consider an example to make this more clear. Again, we will suppose we have 3 processors and 4 names. We first add (1 6 11) to the boundary matrix, giving (1 6) orientation 1, (6 11) orientation 1, and (1 11) orientation -1. These are also the values in the *axes* matrix at the corresponding rows. By checking *axes*, we find that the row corresponding to (1 6) is 1 and so we choose it as a flipping axis. By checking the boundary matrix we find that (1 6) is currently used in the 2-simplex (1 6 11). Comparing (1 6) and (1 6 11), we discover that we are going to flip vertex 11. Simple arithmetic tells us that vertex 11 is processor 3, and that it will take name 4 (since (1 6 11) only uses names 1, 2, and 3). Using these two numbers (number 3 and name 4), we find that the new vertex number after flipping is going to be 12. Thus, the new simplex to add is (1 6 12).

Since (1 6 11) had orientation 1, (1 6 12) will have orientation -1. We add this simplex to the boundary matrix, and update *orientations* and *axes*. In this fashion we repeat until *axes* is all zero (indicating that every possible flipping axis has been used).

Generating Non-Oriented Boundary Matrices

For values of $k \neq n-1$, populating ∂_k is fairly straightforward. After creating *rowSimplices* and *columnSimplices*, we simply iterate over each column simplex and add its boundary to the matrix ∂_k. There is no need to keep track of the orientation of the simplices, since each column simplex will be treated as if it has orientation 1. The only edge cases arise when $k = 0$ and $k = n$.

- When $k = 0$, ∂_k represents the boundary matrix mapping 0-simplices (vertices) to -1-simplices. Since 0-simplices have no boundary, ∂_0 is a $0 \times m$ matrix, where m is the number of 0-simplices.
- When $k = n$, ∂_k represents the boundary matrix mapping n-simplices to $(n-1)$-simplices. Since there are no n-simplices, ∂_n is a $m \times 0$ matrix, where m is the number of $(n-1)$-simplices.

Cycle Groups and Boundary Groups

Once we have calculated the boundary matrices, it is trivial to find the cycle groups and boundary groups. We will denote the cycle group of k-simplices by Z_k, and the boundary group of k-simplices by B_k. We use the following formula:

$$Z_k = ker(\partial_k)$$
$$B_k = Im(\partial_{k+1})$$

That is, the matrix for the cycle group is the kernel of ∂_k, and the matrix for the boundary group is the image space of ∂_{k+1}.

Homology Groups

We define the k^{th} homology group of our mesh by the following relation:

$$H_k = Z_k / B_k$$

The *homology* function takes the two matrices Z_k and B_k, calculates the quotient, and then runs a simple algorithm to interpret the resulting matrix into a string describing the abelian group's structure.

First, we create a larger matrix $K = [Z_k \ B_k]$. Then, we integer-row-reduce K into echelon form using the function $irref$ (we do not allow division as this destroys information). Once this is done, there might be rows that are zero on the Z_k half of K; these rows are deleted from K. Each row is then divided by the values along the diagonal of Z_k half of K. Finally, we take the B_k half of K

and integer row-reduce it into a matrix M. At this point M is a diagonal matrix representing the group structure of the k^{th} homology group. Each element along the diagonal represents a generator for the group in the following way: if a value on the diagonal is a, then the corresponding generator is $\mathbb{Z}/a\mathbb{Z}$.

Betti Numbers

There are several different ways to calculate Betti numbers for a mesh. We decided to use a method independent of the homology group in order to get some degree of independent confirmation of the correctness of our results. Thus, we decided to use the following definition:

$$\begin{aligned} B_k &= \alpha_k - \gamma_k - \gamma_{k-1} \\ \alpha_k &= \text{The number of k-simplices} \\ \gamma_k &= rank(\partial_{k+1}) \end{aligned}$$

Results

Due to the exponential blow-up of the problem, we were only able to generate results for up to 5 processors. The findings are included below, organized by the number of processors.

One Processor

$$H_0 = \mathbb{Z} + \mathbb{Z} \qquad\qquad B_0 = 2$$

With only one processor and two names, we stumble upon a curious edge case in which the resulting mesh is simply two disjoint vertices. This is trivial to check.

Two Processors

$$\begin{aligned} H_0 &= \mathbb{Z} & \qquad B_0 &= 1 \\ H_1 &= \mathbb{Z} & \qquad B_1 &= 1 \end{aligned}$$

The resulting surface is topologically equivalent to the circle S_1, which is easy to see by hand-drawing the mesh.

Three Processors

$$\begin{aligned} H_0 &= \mathbb{Z} & \qquad B_0 &= 1 \\ H_1 &= \mathbb{Z} + \mathbb{Z} & \qquad B_1 &= 2 \\ H_2 &= \mathbb{Z} & \qquad B_2 &= 1 \end{aligned}$$

Here we find that the final state mesh for three processors is very neat: it is topologically equivalent to the torus.

Four Processors

The previous three meshes were initially done by hand and used to check the correctness of the algorithm. The four-processor mesh consists of 3-simplices folding over in 4-space, and therefore could not be reasonably imagined. The algorithm provides the only insight into the structure of the mesh.

$H_0 = \mathbb{Z}$	$B_0 = 1$
$H_1 = 0$	$B_1 = 0$
$H_2 = 20\mathbb{Z}$	$B_2 = 20$
$H_3 = \mathbb{Z}$	$B_3 = 1$

Unfortunately, this mesh does not seem as neat as the previous three meshes. We were concerned that the homology groups were strange due to a bug, but the Betti numbers seem to corroborate these strange properties. It seems that there are twenty “holes” in four-space, but the fundamental group in 3-space is trivial.

Five Processors

The mesh for five processors is even more strange

$H_0 = \mathbb{Z}$	$B_0 = 1$
$H_1 = 0$	$B_1 = 0$
$H_2 = 0$	$B_2 = 0$
$H_3 = 152\mathbb{Z}$	$B_3 = 152$
$H_4 = \mathbb{Z}$	$B_4 = 1$

Conclusions

Although the final state meshes for up to three processors are known surfaces, it seems that this order breaks down for higher-level meshes. The pattern set by four and five processors seems to suggest that meshes for 5+ processors will have trivial fundamental groups up until the full surface itself, which which will instead be riddled with an arbitrary number of holes.