8-2007

A Study of the Homology of Subset Spaces and their Connection to the K-SAT Problem in Computer Science

Oliver J. Thistlethwaite
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Oliver J. Thistlethwaite entitled "A Study of the Homology of Subset Spaces and their Connection to the K-SAT Problem in Computer Science." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Mathematics.

James Conant, Major Professor

We have read this thesis and recommend its acceptance:

David Anderson, Conrad Plaut

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Oliver James Thistlethwaite entitled "A Study of the Homology of Subset Spaces and their Connection to the K-SAT Problem in Computer Science". I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Mathematics.

James Conant

Major Professor

We have read this thesis
and recommend its acceptance:

David Anderson

Conrad Plaut

Accepted for the Council:

Carolyn Hodges

Vice Provost and
Dean of the Graduate School

(Original signatures are on file with official student records.)

A Study of the Homology of Subset Spaces and their Connection to the K-SAT Problem in Computer Science

A Thesis Presented for
the Master of Science Degree
in Mathematics

The University of Tennessee, Knoxville

Oliver James Thistlethwaite

August 2007

Acknowledgments

Dr. Conant, thank you all for your help.

I would also like to thank my father, Dr. Morwen Thistlethwaite,
and the rest of the Thistlethwaite family.

Abstract

It is the purpose of this thesis to introduce an idea for studying questions of computer science via topology. We begin by describing the homology of a certain space constructed from a given subset of the power set of any finite set. We then discuss how this relates to the k-SAT problem in computer science.

We shall use computers as a tool to calculate the homology groups as well as the Euler characteristic of some of these spaces. Due to the sheer number of calculations needed, doing the necessary computations by hand is both impractical and impossible.

In addition, with inspiration from these results, we will provide several rigorous mathematical proofs detailing certain properties of the spaces produced by some input sets.

Contents

List of Tables

List of Figures

1 Introduction

1.1 The Combinatorial Questions

There are several ways we may contruct topological spaces out of a subset of the power set of any finite set. Here we begin with some definitions and examples.

Definition 1. Let $\mathbb{F}$ be a finite set and $\mathcal{A}$ be a collection of proper subsets of $\mathbb{F}$. We may define a simplicial complex $K_{\mathcal{A}}$ as follows. The vertices are the elements of $\mathcal{A}$ and there is an n-simplex $[a_0, a_1, \ldots, a_n]$ if and only if $a_0 \cap a_1 \cap \cdots \cap a_n \neq \emptyset$.

Remark. Let us only consider $\mathcal{A}$s which do not contain the entire set $\mathbb{F}$ as a subset. If we add $\{\mathbb{F}\}$ to our $\mathcal{A}$, then $K_{\mathcal{A}}$ will be contractible, as $K_{\mathcal{A}\cup\{\mathbb{F}\}}$ is a cone on $K_{\mathcal{A}}$.

Example. Define $\mathbb{F} = \{1\,,\,2\,,\,3\}$ and $\mathcal{A}$ to be the set of all proper subsets of $\mathbb{F}$. Then $K_{\mathcal{A}}$ is homotopy equivalent to S^1. This is shown in *Figure 1.*

Definition 2 *(Geometric Realization of a Poset).* Recall that a set, X, has a partial order, $<$, if $<$ is irreflexive and transitive. The pair $(X, <)$ is often called a *poset.* Any poset can be turned into a simplical complex, $|X|$, as follows. The 0-simplices are simply elements of X. The 1-simplices correspond to all pairs x, y where $x < y$. Similarly, the n-simplices correspond to $(n+1)$-tuples, $(x_0, \ldots, x_n)$, where $x_0 < x_1 < \ldots < x_n$. This simplicial complex is called the *geometric realization* of X.

Example. The set of proper subsets of $\{a, b, c\}$ is partially ordered by inclusion. The geometric realization is homeomorphic to S^1. This is shown in *Figure 2.*

Definition 3. Let $\mathbb{F}$ a finite set, and $\mathcal{A}$ a collection of proper subsets of $\mathbb{F}$, as in the first section. Let $\langle\mathcal{A}\rangle$ be the set of all nonempty intersections $a_1 \cap a_2 \cap \ldots \cap a_k$ where $a_i \in \mathcal{A}$, and define the simplicial complex $\widehat{K}_{\mathcal{A}} = |\langle\mathcal{A}\rangle|$.

For example, consider $\mathcal{A} = \{\{1, 2\}, \{2, 3\}\}$. Then
$\langle\mathcal{A}\rangle = \{\{1, 2\}, \{2, 3\}, \{2\}\}$, and the simplicial complex $\widehat{K}_{\mathcal{A}}$ is just two line segments glued at one endpoint of each.

Theorem 1. *The complexes $K_{\mathcal{A}}$ and $\widehat{K}_{\mathcal{A}}$ are homotopy equivalent.*

Proof. First note that $\mathcal{U} = \bigcup_{a\in\mathcal{A}} \mathcal{U}_a$ is a cover of $\widehat{K}_{\mathcal{A}}$, where $\mathcal{U}_a$ is the complex spanned

by all $b_i \in \langle \mathcal{A} \rangle$ such that $b_i \subset a$.

Now we see $b_i \in \mathcal{U}_a \cap \mathcal{U}_c \iff b_i \in a$ and $b_i \in c \iff b_i \in a \cap c \iff$ $b_i \in \mathcal{U}_{a \cap c}$. Therefore $\mathcal{U}_a \cap \mathcal{U}_c = \mathcal{U}_{a \cap c}$. So if this intersection exists it is a simplex, so it is contractible.

Now generalizing this argument we see,

$$\mathcal{U}_{a_1} \cap \cdots \cap \mathcal{U}_{a_n} \simeq \begin{cases} * & \text{if } a_1 \cap \cdots \cap a_n \neq \emptyset \\ \emptyset & \text{if } a_1 \cap \cdots \cap a_n = \emptyset \end{cases},$$

where $*$ is the homotopy type of a point. Note we may make our $\mathcal{U}$ cover an open cover of $\widehat{K}_{\mathcal{A}}$ by taking arbitrarily small neighborhoods around each $\mathcal{U}_a$.

Now we may apply Corollary 4.G.3 from [Hat06] to show $\widehat{K}_{\mathcal{A}}$ is homotopy equivalent to the nerve $N\mathcal{U}$. But this nerve is clearly homeomorphic to $K_{\mathcal{A}}$; so we're done. □

1.2 Relationship to Problems in Computer Science

Definition 4 *(Boolean Formula).* First fix a finite alphabet $S_n = \{x_1, \ldots, x_n\}$. A *boolean formula* is a formula in these variables and their negations $\bar{x}_1, \ldots, \bar{x}_n$, with the logical operators $\wedge$ ("AND") and $\vee$ ("OR").

Definition 5 *(k-SAT).* A boolean formula *is an element of k-SAT* if it is of the form

$$(a_{11} \vee a_{12} \vee \ldots \vee a_{1k}) \wedge (a_{21} \vee a_{22} \vee \ldots \vee a_{2k}) \wedge \cdots \wedge (a_{m1} \vee a_{m2} \vee \ldots \vee a_{mk}),$$

where each a_{ij} is an element of $S_n \cup \bar{S}_n$, and furthermore, the same variable cannot occur more than once in each pair of parentheses.

A formula is said to be *satisfiable* if there is an assignment of T or F to each variable in S_n which makes the formula true. (Such an assignment is called a *satisfaction.*) It is a theorem that there is a polynomial time algorithm for deciding if a 2-SAT formula is satisfiable, but that deciding whether a 3-SAT formula is satisfiable is an NP problem. Thus it is desirable to explore the difference between these two problems.

P and NP are important concepts in computer science. The complexity class of decision problems that can be solved on a deterministic sequential machine in polynomial time is known as P. The class of decision problems that can be verified in polynomial time is known as NP. It is one of the most significant unsolved problems

in mathematics whether or not P = NP. It is considered to be one of the "Millennium Problems" and a million dollar prize is offered for the first correct proof. The ultimate goal of this project was to find a topological invariant for distinguishing between sets of P and NP formulae, but much work still remains to be done.

Definition 6.
1. Let $\mathcal{T}_n$ denote the set of all truth assignments of the variables in S_n. Thus $\mathcal{T}_n$ has 2^n elements.
2. Let ϕ be a Boolean formula. Define $x_\phi \in \mathcal{P}(\mathcal{T}_n)$ by letting x_ϕ be the set of all satisfactions of the formula ϕ.
3. Now let $X_{n,k} \subset \mathcal{P}(\mathcal{T}_n)$ be defined as

$$X_{n,k} = \{x_\phi : \phi \text{ is a } k\text{-SAT formula}\} \setminus \{\emptyset, \mathcal{T}_n\}$$

4. Let $\widehat{K}_{n,k}$ be the simplicial complex $|X_{n,k}|$.

The connection between this question and the first section is the following observation.

Theorem 2. *$\widehat{K}_{n,k} = \widehat{K}_{\mathcal{A}}$, where $\mathcal{A} = \{x_\phi : \phi = a_1 \vee \cdots \vee a_k\} \setminus \{\emptyset\}$, and $\mathbb{F} = \mathcal{T}_n$.*

Proof. Notice that $x_\phi \cap x_\psi = x_{\phi \wedge \psi}$. Thus $\langle \mathcal{A} \rangle$ is equal to the set of all x_ϕ where ϕ is a set of formulas from $\mathcal{A}$ which are "and"ed together. That is, x_ϕ is a k-SAT formula. □

Example. Let's work out $\widehat{K}_{2,2}$. The only possible atomic formulae are
$\phi_1 = x_1 \vee x_2$ $\phi_2 = x_1 \vee \bar{x}_2$ $\phi_3 = \bar{x}_1 \vee x_2$ $\phi_4 = \bar{x}_1 \vee \bar{x}_2$.
Now
$x_{\phi_1} = \{\{T,T\},\{T,F\},\{F,T\}\}$,
$x_{\phi_2} = \{\{T,T\},\{T,F\},\{F,F\}\}$,
$x_{\phi_3} = \{\{T,T\},\{F,T\},\{F,F\}\}$,
$x_{\phi_4} = \{\{F,T\},\{T,F\},\{F,F\}\}$.

Then $\{x_{\phi_1}, x_{\phi_2}, x_{\phi_3}, x_{\phi_4}\}$ forms the set $\mathcal{A}$, and the poset $\widehat{X}_{2,2}$ that we are interested in is formed by taking all intersections of elements of A, excluding the empty set.

Abbreviating notation, we get

$$\begin{aligned}\widehat{X}_{2,2} = \{&\{TT,TF,FT\},\{TT,TF,FF\},\{TT,FT,FF\},\{FT,TF,FF\},\\ &\{TT,TF\},\{TT,FT\},\{FT,TF\},\{TT,FF\},\{TF,FF\},\{FT,FF\},\\ &\{TT\},\{TF\},\{FT\},\{FF\}\}\ .\end{aligned}$$

Definition 7. One can also define a complex $K_{n,k}$ which has a vertex for every $x_\phi \in \mathcal{A}$ as above, and which has an m-simplex $[a_0, \ldots, a_m]$ whenever $a_0 \cap \cdots \cap a_m \neq \emptyset$.

Remark. For the rest of this paper when referring to k-SAT with n variables, let us actually mean the simplicial complex $K_{n,k}$, unless the original meaning is implied in the text. Also from now on we will be just studying $K_{n,k}$ complexes and not $\widehat{K}_{n,k}$ ones.

Example. Let's work out $K_{2,2}$. We already calculated

$$\mathcal{A} = \{\{TT, TF, FT\}, \{TT, TF, FF\}, \{TT, FT, FF\}, \{FT, TF, FF\}\} \ .$$

Then $K_{2,2}$ has the simplices

$$[1]\,,\ [2]\,,\ [3]\,,\ [4]\,,\ [12]\,,\ [13]\,,\ [14]\,,\ [23]\,,\ [24]\,,\ [34]\,,\ [123]\,,\ [124]\,,\ [134]\,,\ [234] \ .$$

By Theorem 4, this yields a 2-sphere: $K_{2,2} \simeq S^2$.

2 Literature Review

Due to the unique nature of this project, there was not much literature available for review. However, there were several interesting articles on related topics.

Michael Freedman has written several papers illustrating a possible connection between the P / NP problem and techniques of topology. First we discuss some of these papers.

In [Fre98b], Freedman discusses the P / NP problem and [notes that] in 25 years only technical progress on the problem has been made. He makes note of new evidence obtained from field theory that some physical system might be manipultated to solve NP and even #P-hard problems in polynomial time. The most significant idea presented is that "ultrafilter limits" might be applied to the P / NP problem to convert it into a logicial problem of decidability. These "ultrafilter limits" are a topological technique developed by Gromov. The concept is that in such a limit P would become decidable, so a problem with an undecidable limit would be shown to be outside of P.

In our next Freedman article, [Fre99], he defines infinite generalizations of *2*-SAT and *3*-SAT which are respectively algorithmic and undecidable. In these generalizations, decidability distinguishes between *2*-SAT and *3*-SAT. While this doesn't prove *2*-SAT and *3*-SAT are distinct, it is an important step in that direction.

The final Freedman article we make mention to is [Fre98a]. Here he further delves into "ultrafilter limits" idea of a way of distinguishing P and NP. He applies these "ultrafilter limits" to the classical Turing machine model of computation and develops a paradigm for distinguishing P from NP as a logical problem of decidability. His goal was to find an appropriate limit to prove the problems in P are decidable and hence any problem that is undecidable must lie outside of P.

Now we move onto [KB00], an article by J. Kouneiher and A.P.M. Balen. Here they outline a definition for a propositional manifold and logical cohomology. A propositional manifold is a generalization of Boolean algebras of propositions. The logical cohomology can be defined as the cohomology of these propositional manifolds. Finally it is shown that if two Boole algebras of mathematicial propositions are nonequivalent then their cohomologies are not isomorphic. The topology for spaces of formulae defined in [KB00] is the order topology relating to logical implication. This inspired the definition of the complexes $\widehat{K}_{\mathcal{A}}$ in this thesis.

3 Materials and Methods

3.1 How the Homology Groups are Calculated

To calculate the homology of $K_{\mathcal{A}}$ we can use the equation,

$$\mathbb{Q} \otimes H_i(K_{\mathcal{A}}) \cong \frac{\mathbb{Q}^{dim(C_i)-rank(\partial_i)}}{\mathbb{Q}^{rank(\partial_{i+1})}}.$$

It is important to note that we aren't actually calculating H_i but rather $\mathbb{Q} \otimes H_i$. Tensoring with $\mathbb{Q}$ kills off all the torsion in H_i. While $\mathbb{Q} \otimes H_i$ isn't quite as powerful a topological invariant as H_i, the computations are greatly simplified by using it.

Now we shall prove the above isomorphism.

Proposition 2.

$$\mathbb{Q} \otimes H_i(K_{\mathcal{A}}) \cong \frac{\mathbb{Q}^{dim(C_i)-rank(\partial_i)}}{\mathbb{Q}^{rank(\partial_{i+1})}}.$$

Proof. First, from the definition of H_i we have,

$$H_i(K_{\mathcal{A}}) \cong \frac{Ker(\partial_i : C_i \to C_{i-1})}{Im(\partial_{i+1} : C_{i+1} \to C_i)}.$$

Now if we tensor $H_i(K_{\mathcal{A}})$ with $\mathbb{Q}$ using the fact that $\mathbb{Q} \otimes H_i(K_{\mathcal{A}}) \cong H_i(\mathbb{Q} \otimes C_*)$, we get

$$\mathbb{Q} \otimes H_i(K_{\mathcal{A}}) \cong \frac{Ker(\partial_i : C_i \otimes \mathbb{Q} \to C_{i-1} \otimes \mathbb{Q})}{Im(\partial_{i+1} : C_{i+1} \otimes \mathbb{Q} \to C_i \otimes \mathbb{Q})}.$$

The above is isomorphic to

$$\frac{\mathbb{Q}^{dim(ker\partial_i)}}{\mathbb{Q}^{dim(im\partial_{i+1})}},$$

as tensoring with $\mathbb{Q}$ is equivalent to killing off the torsion. Now, using the elementary linear algebra property that the rank plus the nullity is equal to the dimension of the column space, we find,

$$\frac{\mathbb{Q}^{dim(ker\partial_i)}}{\mathbb{Q}^{dim(im\partial_{i+1})}} \cong \frac{\mathbb{Q}^{dim(C_i)-rank(\partial_i)}}{\mathbb{Q}^{rank(\partial_{i+1})}}.\square$$

Finally, it is worth mentioning that the rank of a homomorphism $f : G \otimes \mathbb{Q} \to H \otimes \mathbb{Q}$ of finitely generated modules is determined by first finding a basis of $\beta_G =$

$\{g_1\,,\,\ldots\,,\,g_n\}$ of $G \otimes \mathbb{Q}$ and a basis $\beta_H = \{h_1\,,\,\ldots\,,\,h_m\}$ of $H \otimes \mathbb{Q}$. Once bases are known, we can find the coefficients c_{ij} such that $f(g_i) = \sum_{j=1}^{n} c_{ij}h_j$ for each i. Now, to find the rank of f, we simply need to calculate the rank of the matrix,

$$\begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1m} \\ c_{21} & c_{22} & \ldots & c_{2m} \\ \vdots & \vdots & & \vdots \\ c_{n1} & c_{n2} & \ldots & c_{nm} \end{bmatrix}.$$

3.2 How the Euler Characteristic is Calculated

The Euler characteristic χ, is a topological invariant which we can use to describe an aspect of the shape or structure of a simplicial complex. While the Euler characteristic is not as powerful an invariant as the homology type, it is much easier to compute. For a finite simplicial complex $K_{\mathcal{A}}$, the Euler Characteristic $\chi(K_{\mathcal{A}})$ is defined by:

$$\chi(K_{\mathcal{A}}) = c_0 - c_1 + c_2 - \ldots$$

This is an alternating sum where each c_i is dimension of the C_{i-th} chain group. Then, c_i is equal to the number of i-cells in the complex. So, we simply need to find the number of i-cells in $K_{\mathcal{A}}$ for each i and insert these numbers into the above to find χ.

Note, each c_i is the number of nonempty intersections of $i+1$ elements in some subset $\mathcal{A}$ of the power set of a finite set $\mathbb{F}$. So, as $\mathcal{A}$ is finite, we only need to find up to $c_{|\mathcal{A}|-1}$.

3.3 Program Design

We begin by taking as input a file containing a collection of subsets of some finite set $\mathbb{F}$. For simplification we will only be considering sets $\mathbb{F}$ of form $\mathbb{F} = \{1\,,\,2\,,\,3\,,\,\ldots\,,\,n\}$ where $n \in \mathbb{N}$. The subsets will be stored as a list of numbers of F with 0s to specify the gaps between subsets. For example $\mathcal{A} = \{\{1\}\,,\{2\,,\,3\}\,,\,\{3\}\}$ would need to be stored as 1 0 2 3 0 3 0 in the input file.

The next step is to convert this into a more convenient format. First we read through the data to determine both the size of $\mathbb{F}$ and size of $\mathcal{A}$, our input set. Now we initialize an integer array of size $|F| * |\mathcal{A}|$ and store each element of $\mathcal{A}$ in a block of numbers of size $|F|$ with 0s for padding. For instance, $\mathcal{A} = \{\{1\}\,,\{2\,,\,3\}\,,\,\{3\}\}$ would be stored as 1 0 0 2 3 0 3 0 0. This format is necessary so we can quickly move

between elements without having to calculate the length of an element of $\mathcal{A}$ before we can move onto the next one, as would be required in the original format in the input file. Note it is also faster to use a continuous block of memory such as this array than it would be to use a linked list. A linked list would require more memory access to obtain pointers to elements, whereas in this format all that is required is some pointer arithmetic to obtain the address of the next element.

Now that we have our input set $\mathcal{A}$ in a convenient format, the next step is to calculate the chain groups for $K_{\mathcal{A}}$. That is to find each C_k, or kth chain group, we need to know all the nontrivial intersections of $k+1$ and fewer elements of $\mathcal{A}$.

Note we only are required to calculate the first $|\mathbb{F}|-1$ chain groups to find the homology groups.

Lemma 1. *$H_i(K_{\mathcal{A}})$ is trivial for all $i > |\mathbb{F}|-2$.*

Proof. By Theorem 1, $K_{\mathcal{A}}$ is homotopy equivalent to $\widehat{K}_{\mathcal{A}}$.

Each simplex in $\widehat{K}_{\mathcal{A}}$ corresponds to an n-tuple $(x_1, \ldots, x_n)$ of $x_i \in \langle \mathcal{A} \rangle$, where $x_1 \subset x_2 \subset \ldots \subset x_n$. Note since $x_1 \subset x_2$, x_2 must be at least a 2 element subset of $\mathbb{F}$. Likewise, x_n must be at least an n element subset of $\mathbb{F}$. So the largest possible simplex in $\widehat{K}_{\mathcal{A}}$ is an $(|\mathbb{F}|-2)$-simplex, as we are excluding $\mathcal{A}$s that contain $\mathbb{F}$ as a subset. Recall,

$$H_i(\widehat{K}_{\mathcal{A}}) \cong \frac{Ker(\partial_i : C_i \to C_{i-1})}{Im(\partial_{i+1} : C_{i+1} \to C_i)}.$$

So it is clear $H_i(\widehat{K}_{\mathcal{A}})$ is trivial for all $i > |\mathbb{F}|-2$. Therefore $H_i(K_{\mathcal{A}})$ is also. $\square$

We start by initializing three arrays of size $|\mathbb{F}|$ of cells, where each cell is stored as a pointer to some element of $\mathcal{A}$. Let us call them a, p, and I. a will serve as a placeholder for cells to add to a simplex, p_k will represent the next element after a_k for each k, and finally I will be used to store intersections of various cells.

First we read in the first element of $\mathcal{A}$ into p_0, the first position of p. We then set $a_0 = p_0$ and increment p_0. We are starting with 0-simplices so, as we don't need to calculate any intersections, we just set $I_0 = a_0$. We then test to see if there are any more elements left in $\mathcal{A}$. If there are and if $i < |F|-1$ we set $p_1 = p_0$ and increment i.

In the general case, we first set $a_i = p_i$ and calculate the intersection of a_i and I_{i-1}, which we will store in I_i. If this intersection is nonempty and $i < |\mathbb{F}|-1$, we still have some slots remaining in our arrays, so we set $p_{i+1} = p_i$ and increment i, and the process is repeated. Otherwise, if $i = |\mathbb{F}|-1$, we have run out of slots in

our arrays, so we just repeat the above on the next element of $\mathcal{A}$. Finally when we have run out of elements in $\mathcal{A}$, we first decrement i and start the process over with the next element after a_i.

Note that every time a nonempty cell is added to, say the kth slot in I, we store from the beginning to the kth slots of a as k-simplex. Also note the simplices are stored in a dynamic linked list structure, so we may store them on our first iteration, without having to first calculate the storage needed.

Finally, we know the algorithm has completed if we try to move to a negative slot of our arrays; if $i < 0$. Simply put, we are going through all the possible k-simplices of $K_{\mathcal{A}}$ in lexicographic order. The pseudocode for the algorithm is shown in *Figure 3.*

Now that we have calculated the generators for all the chain groups, the next step is to calculate the rank of the boundary operations between them. For example, say we are trying to find the rank of the boundary map $\partial_n : C_n \to C_{n-1}$. The first step is to determine what generators of C_n get mapped to in the form of generators of C_{n-1}. The boundary map $\partial_n : C_n \to C_{n-1}$ is defined by the formula:

$$\partial_n(\sigma) = \sum_i (-1)^i \sigma|_{[v_0, \ldots, \widehat{v_i}, \ldots, v_n]}$$

This is an alternating sum of $(n-1)$-simplices, each i^{th} simplex formed by removing the i^{th} cell from σ. So given some simplex σ of C_n, it is trivial to determine $\partial_n(\sigma)$. The difficulty arises in that we would like to determine the coefficients x_i needed to write $\partial_n(\sigma)$ in the form:

$$\partial_n(\sigma) = \sum_i x_i b_i$$

where b_i are the ordered $(n-1)$-simplices of $\mathcal{A}$, the generators of C_{n-1}.

One's first thought is to simply write an algorithm that parses through all the $(n-1)$-simplices of $K_{\mathcal{A}}$ until a match is found for each summand in the above. However, when the ranks of the chain groups get large, this approach is very inefficient. For instance, if $K_{\mathcal{A}}$ contains 20,000 $(n-1)$-simplices, we must parse through this set of 20,000 elements each time we wish to find one of the x_i in the above.

A much more efficient way of doing this is to store the basis of C_{n-1}, the $(n-1)$-simplices, in an ordered tree data structure. We shall define the tree as follows. Each node will have an integer value as well as a set of size $|\mathbb{F}| + 1$ of pointers to other nodes, let us call this set *branches*.

To store a simplex, first we move to the head of the tree and then look at the first number in the simplex's first cell, say it is j. We then move to the node *branches*[j] is pointing to. If no such node exists there, we create one there and move to it.

We continue in this way until we reached the end of a cell. When this happens we move to the node *branches*[0], that the current node is pointing to. Finally when we have reached the end of our simplex, we change the integer value stored in our current cell to that simplices position in the set of $(n-1)$-simplices we calculated earlier.

After the tree has been constructed, finding what position a simplex has in the list of $(n-1)$-simplices is simply done by scrolling through the tree in the same way as earlier and looking at the value of the node we end up in. Both the contruction of the tree and finding the positions of elements can be done in linear time with respect to n and the total number of $(n-1)$-simplices.

Illustrated in *Figure 4* is a diagram of how the set of simplices
$\{\{1\}\ ,\ \{1\ ,\ 2\}\}\ ,\ \{\{2\}\ ,\ \{2\ ,\ 3\}\}\ ,\ \{\{3\}\ ,\ \{3\ ,\ 4\}\}$
is stored in tree form.

Now we have all the information we need to construct our boundary matrix. We will store this information in a matrix of size $\dim(C_n)$ by $\dim(C_{n-1})$. Now all that remains is to calculate the rank of this matrix.

Note that these matrices quickly become very large and very sparse as we use larger $\mathcal{A}$s, so using traditional rank finding algorithms becomes impossible without requiring a massive amount of computer memory. A much more efficient method is to use a sparse matrix rank algorithm. While these sparse algorithms may be slower than their nonsparse counterparts, their lack in speed is certainly made up by requiring only a tiny fraction of the memory needed by the nonsparse variety.

To determine rank, we will use the LinBox C++ library. This library is appropriate, as it is one of the only sparse matrix libraries designed to handle integer matrices. More information can be found at [Lin07]. Most sparse matrix computation today is done with matrices whose entries are decimal approximations, which arise often in engineering related fields.

So now after we have calculated the ranks of all the necessary boundary maps $\partial_n : C_n \to C_{n-1}$, we may determine $\mathbb{Q} \otimes H_i(K_{\mathcal{A}})$ for each i. This is calculated by

$$\mathbb{Q} \otimes H_i(K_{\mathcal{A}}) \cong \frac{\mathbb{Q}^{dim(C_i)-rank(\partial_i)}}{\mathbb{Q}^{rank(\partial_{i+1})}}.$$

Finally, it is worth mentioning that the algorithm to find the Euler characteristic is very similar to the algorithm mentioned earlier to calcuate the chain groups. The key difference being that we must now parse through ALL the chain groups and not just the first $|\mathbb{F}|-1$. In addition, now we are no longer required to store the cells, we only need to count how many i-cells there are for each i so we may calculate the alternating sum:

$$\chi(K_{\mathcal{A}}) = c_0 - c_1 + c_2 - \dots \quad ,$$

where each c_i is the number of i-cells in the complex.

Example. Now let us try using the method outlined in Section 3.1 of this text on the complex illustrated in *Figure 1.* Note $\mathcal{A} = \{\{1\}\ ,\ \{1\ ,\ 2\}\ ,\ \{1\ ,\ 3\}\ ,\ \{2\}\ ,\ \{2\ ,\ 3\}\ ,\ \{3\}\}$. Working out all the simplices we get:

$$
\begin{aligned}
C_0 &= \langle \{1\}\ ,\ \{1\ ,\ 2\}\ ,\ \{1\ ,\ 3\}\ ,\ \{2\}\ ,\ \{2\ ,\ 3\}\ ,\ \{3\} \rangle \\
C_1 &= \langle \{\{1\}\ ,\ \{1\ ,\ 2\}\}\ ,\ \{\{1\}\ ,\ \{1\ ,\ 3\}\}\ , \\
&\qquad \{\{1\ ,\ 2\}\ ,\ \{1\ ,\ 3\}\}\ ,\ \{\{1\ ,\ 2\}\ ,\ \{2\}\}\ , \\
&\qquad \{\{1\ ,\ 2\}\ ,\ \{2\ ,\ 3\}\}\ ,\ \{\{1\ ,\ 3\}\ ,\ \{2\ ,\ 3\}\}\ , \\
&\qquad \{\{1\ ,\ 3\}\ ,\ \{3\}\}\ ,\ \{\{2\}\ ,\ \{2\ ,\ 3\}\}\ ,\ \{\{2\ ,\ 3\}\ ,\ \{3\}\} \rangle \\
C_2 &= \langle \{\{1\}\ ,\ \{1\ ,\ 2\}\ ,\ \{1\ ,\ 3\}\}\ ,\ \{\{1\ ,\ 2\}\ ,\ \{2\}\ ,\ \{2\ ,\ 3\}\}\ , \\
&\qquad \{\{1\ ,\ 3\}\ ,\ \{2\ ,\ 3\}\ ,\ \{3\}\} \rangle\ .
\end{aligned}
$$

$\partial_1 : C_1 \to C_0$ corresponds to the matrix

$$
\begin{bmatrix}
-1 & 1 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & 0 & 0 \\
0 & -1 & 1 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 0 & 0 \\
0 & -1 & 0 & 0 & 1 & 0 \\
0 & 0 & -1 & 0 & 1 & 0 \\
0 & 0 & -1 & 0 & 0 & 1 \\
0 & 0 & 0 & -1 & 1 & 0 \\
0 & 0 & 0 & 0 & -1 & 1
\end{bmatrix}
$$

which has rank 5.

$\partial_2 : C_2 \to C_1$ corresponds to the matrix

$$
\begin{bmatrix}
1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 1
\end{bmatrix}
$$

which has rank 3.

Now using the formula:

$$\mathbb{Q} \otimes H_i(\mathcal{A}) \cong \frac{\mathbb{Q}^{dim(C_i) - rank(\partial_i)}}{\mathbb{Q}^{rank(\partial_{i+1})}},$$

we find $\mathbb{Q} \otimes H_0(K_{\mathcal{A}}) \cong \mathbb{Q}$ and $\mathbb{Q} \otimes H_1(K_{\mathcal{A}}) \cong \mathbb{Q}$. All the other homology groups are trivial. So this suggests $K_{\mathcal{A}}$ is homotopy equivalent to S^1.

4 Results and Discussion

4.1 The Homology of Some Spaces Generated by Small Input Sets

Let us start by calculating the homology of $K_{\mathcal{A}}$ for some $\mathcal{A}$s that are contained in the power sets of some small sized $\mathbb{F}$s. Before we start, let us prove a helpful lemma.

Lemma 2. *If we append any $\{f\}$ to $\mathcal{A}$ where $f \in \mathbb{F}$ and $\mathcal{A}$ is a subset of the power set of $\mathbb{F}$, the homology of $K_{\mathcal{A}}$ is affected in one of two ways.*

1. *A new contractible path component is added to $K_{\mathcal{A}}$.*
2. *The homology of $K_{\mathcal{A}}$ is left unchanged.*

Proof. Case 1: Suppose $\{f\} \cap a = \emptyset$ for all $a \in \mathcal{A}$. Then it is clear $\{f\}$ forms no simplices with any other $a \in \mathcal{A}$. Therefore it must be a disjoint vertex in $K_{\mathcal{A}}$, hence a new contractible path component.

Case2: Suppose $\{f\} \cap a \neq \emptyset$ for some $a \in \mathcal{A}$. Define S_f to be the set of all points of $\mathcal{A}$ that intersect $\{f\}$. It is clear S_f is a simplex, as the intersection of all its points contains at least $\{f\}$. Note $S_f \cup \{f\}$ is also a simplex. To show $K_{\mathcal{A}\cup\{f\}}$ is homotopy equivalent to $K_{\mathcal{A}}$, we may simply retract $S_f \cup \{f\}$ to S_f. $\square$

The lemma tells us that nothing really interesting happens to $K_{\mathcal{A}}$ by appending elements of $\mathbb{F}$ to $\mathcal{A}$; so let us only consider $\mathcal{A}$s without any elements from $\mathbb{F}$. Now, let us work out the homology of $K_{\mathcal{A}}$ for all $\mathcal{A}$s corresponding to $\mathbb{F}$ of size 3 and $\mathbb{F}$ of size 4. The results are shown in *Table 1*.

In *Table 1*, each row represents a particular configuration of H_i. For instance the first row would be just $H_0 = \mathbb{Q}$ and all other H_i trivial. It is interesting to note when $|\mathbb{F}| = 3$, $K_{\mathcal{A}} \simeq S^1$ only occurs if we take $\mathcal{A}$ to be the set of all subsets of $\mathbb{F}$ of size 2. Also when $|\mathbb{F}| = 4$, $K_{\mathcal{A}} \simeq S^1 \vee S^1 \vee S^1$ only occurs if we take $\mathcal{A}$ to be the set of all subsets of $\mathbb{F}$ of size 2. We actually prove these $\mathcal{A}$s produce $K_{\mathcal{A}}$s of these types in Theorem 4.

Note it is impossible to compute tables like this for any $\mathbb{F}$ where $|\mathbb{F}| > 4$, as there are far too many possible $\mathcal{A}$s. So the best we can do is take a random sample. Also the computations required to find the homology increase as well, as we choose larger $\mathbb{F}$s. Now let us see what the homology looks like for some random $K_{\mathcal{A}}$s corresponding to $\mathbb{F}$ of order 5.

From the information in *Table 2*, it would appear that $K_{\mathcal{A}} \simeq S^3$ for a lot of $\mathcal{A}$s taken from the power set of $\mathbb{F}$ where $|\mathbb{F}| = 5$, similar to how it appeared were a lot of $K_{\mathcal{A}} \simeq S^1$ for $|\mathbb{F}| = 4$. It is possible to work out random $K_{\mathcal{A}}$ for sets of higher order, but the time needed for the computations increases exponentially.

4.2 Proof that any Finite Simplicial Complex can be Realized

The following theorem is important because it shows the study of $K_{\mathcal{A}}$ is actually equivalent to the study of finite simplicial complexes in general.

Theorem 3. *Any finite simplicial complex $\mathcal{X}$ is equivalent to some $K_{\mathcal{A}}$ for some set $\mathbb{F}$ and some subset $\mathcal{A}$ of the power set of $\mathbb{F}$.*

Proof. Let $\mathcal{X}$ be a finite simplicial complex with i-simplices $e^i_{\alpha_i}$, where $1 \leq \alpha_i \leq n_i$ for each i.

Define $\mathbb{F} = \{e^0_1, \ldots, e^0_{n_0}, \ldots, e^m_1, \ldots, e^m_{n_m}\}$. So $\mathbb{F}$ is the set of ALL the simplices of $\mathcal{X}$. Now define v_i to be the set of all the simplices of $\mathcal{X}$ that contain e^0_i for each i. Let $\mathcal{A} = \{v_1, \ldots, v_{n_0}\}$.

So we want to show $K_{\mathcal{A}}$ is equivalent to $\mathcal{X}$.

Let $f : \mathcal{X} \to K_{\mathcal{A}}$ be a map where each simplex $e^i_{\alpha_i}$ is sent to $[v_{\beta_1}, \ldots, v_{\beta_n}]$ where v_{β_j} are the v_j that contain $e^i_{\alpha_i}$. It is clear the image of $e^i_{\alpha_i}$ is a simplex as each v_{β_j} contains at least $e^i_{\alpha_i}$, so their intersection is nonempty.

Now define $g : K_{\mathcal{A}} \to \mathcal{X}$, where each simplex $[v_{\beta_1}, \ldots, v_{\beta_n}]$ is sent to $e^k_{\alpha_k}$, where $e^k_{\alpha_k}$ is the smallest simplex contained in $\bigcap_{j=1}^{n} v_{\beta_j}$. Note the smallest simplex in $\bigcap_{j=1}^{n} v_{\beta_j}$ is an $(n-1)$-simplex and there can only be one $(n-1)$-simplex on n vertices in a simplicial complex, so it is unique and our g map is well defined.

First we want to show f and g are simplicial maps.

Suppose σ is face of τ. Then $f(\tau) = [v_{\beta_1}, \ldots, v_{\beta_n}]$, where v_{β_i} are the v_j that contain τ. Also $f(\sigma) = [v_{\alpha_1}, \ldots, v_{\alpha_n}]$, where v_{α_i} are the v_j that contain σ. But all the v_j that contain σ also contain τ; so we see $f(\sigma)$ is indeed a face of $f(\tau)$.

Now consider a simplex $[v_{\beta_1}, \ldots, v_{\beta_n}]$ in $K_{\mathcal{A}}$. Let $[v_{\alpha_1}, \ldots, v_{\alpha_k}]$ be a face of that simplex. Note $g([v_{\beta_1}, \ldots, v_{\beta_n}])$ is the $(n-1)$-simplex on the vertices $e^0_{\beta_1}, \ldots, e^0_{\beta_n}$ and $g([v_{\alpha_1}, \ldots, v_{\alpha_k}])$ is the $(k-1)$-simplex on the vertices $e^0_{\alpha_1}, \ldots, e^0_{\alpha_k}$. Since $\{e^0_{\alpha_1}, \ldots, e^0_{\alpha_k}\}$ is a subset of $\{e^0_{\beta_1}, \ldots, e^0_{\beta_n}\}$, we see $g([v_{\alpha_1}, \ldots, v_{\alpha_k}])$ is indeed a face of $g([v_{\beta_1}, \ldots, v_{\beta_n}])$.

Now we need to show their compositions are the identity.

Suppose $f(e^i_{\alpha_i}) = [v_{\beta_1}, \ldots, v_{\beta_n}]$. Note $e^i_{\alpha_i}$ is contained in each v_{β_j} and $e^i_{\alpha_i}$ is an $(n-1)$-simplex. Thus $g([v_{\beta_1}, \ldots, v_{\beta_n}]) = e^i_{\alpha_i}$.

Now suppose $g([v_{\beta_1}, \ldots, v_{\beta_n}]) = e^k_{\alpha_k}$. Note $e^k_{\alpha_k}$ is the smallest simplex contained in the intersection of all the v_{β_j}. Note $e^k_{\alpha_k}$ is an $(n-1)$-simplex. $f(e^k_{\alpha_k})$ is a list of all the v_j that contain $e^k_{\alpha_k}$. There are n-1 of these and they are precisely the vertices of our original simplex $[v_{\beta_1}, \ldots, v_{\beta_n}]$. So we see $f(e^k_{\alpha_k}) = [v_{\beta_1}, \ldots, v_{\beta_n}]$.

So we have shown $\mathcal{X}$ and $K_{\mathcal{A}}$ are equivalent simplicial complexes. $\square$

4.3 The Space Generated by the Set of all Subsets of Size k

Here we prove an interesting theorem about the space $K_{\mathcal{A}}$ generated by the set of all subsets of some $\mathbb{F}$ of size k. This will become useful later.

Theorem 4. *If we take $\mathcal{A}$ to be the set of all subsets of $\mathbb{F}$ of size k where $|\mathbb{F}| = n$, then $K_{\mathcal{A}}$ is homotopy equivalent to a wedge of $(k-1)$-spheres. Also the number of these spheres is determined by $\binom{n-1}{k}$.*

Proof. Let $\mathcal{A}$ be the set of all subsets of size k. It is clear

$$\langle \mathcal{A} \rangle = \begin{cases} \text{size } k \text{ subsets} \\ \vdots \\ \text{size 1 subsets} \end{cases}.$$

$\widehat{K}_{\mathcal{A}}$ decomposes as a union of simplices; one i-simplex Δ_x for each subset x of size i. Note Δ_x is a subcomplex of $\widehat{K}_{\mathcal{A}}$ spanned by subsets of x.

Claim: Δ_x is an i-simplex.

In fact, Δ_x is the Barycentric subdivision of the simplex whose vertices are elements of x.

Suppose Δ is a simplex whose vertices are elements of x. To subdivide, note there is vertex in $b(\Delta)$ for every face of Δ. The faces of Δ correspond to subsets of x. This gives us the vertices of Δ_x.

Note, v_1 is a face of v_2 if and only if the corresponding subsets are contained in each other. So simplices $v_1\,,\,\ldots\,,\,v_l$ in $b(\Delta)$ occur when $x_{v_1} \subset x_{v_2} \cdots \subset x_{v_l}$. This means these simplices are in $\Delta_x \subset \widehat{K}_{\mathcal{A}}$.

We see $\widehat{K}_{\mathcal{A}} = \bigcup \Delta_x$. Now we want to collapse some of the Δ_xs. In general we may collapse a pair Δ_x and Δ_y if Δ_x is a k-simplex and Δ_y is a $(k-1)$-simplex that is a face of Δ_x and all the faces of Δ_x except for Δ_y have been crushed.

Note in *Table 3*, σ represents arbitrary numbers that can occur. First we may collapse the pair $\Delta_2 \subset \Delta_{12}$ as Δ_{12} has faces Δ_1 and Δ_2, and Δ_1 is already crushed since it is just a vertex. We may also crush the pair $\Delta_3 \subset \Delta_{13}$ and continue in this way until $\Delta_n \subset \Delta_{1n}$ is crushed. So we have crushed 1σ and the vertices 2 to n.

Similarly in the general case we may use 1σ in the set of subsets of size i to crush all the $2\sigma\,,\,\ldots n\sigma$ in the set of subsets of size $i-1$.

So after this process is completed, we see all we have left is the vertex 1 and $2\sigma\,,\,\ldots\,,\,\{n-k+1\,,\,\ldots\,,\,n\}$ in the subsets of size k. Note if we completely crush the boundary of a $(k-1)$-simplex to a point, we get a $(k-1)$-sphere. As all the simplices have their boundaries crushed to the same point, we see $\widehat{K}_{\mathcal{A}}$ is indeed a wedge of $(k-1)$-spheres. And the number of subsets of size k in $2\sigma\,,\,\ldots\,,\,\{n-k+1\,,\,\ldots\,,\,n\}$ is $\binom{n-1}{k}$. Finally using Theorem 1, we know $\widehat{K}_{\mathcal{A}}$ is homotopy equivalent to $K_{\mathcal{A}}$; so we're done. $\square$

Corollary. The set of proper subsets of a set with n elements has a geometric realization homotopy equivalent to S^{n-2}.

4.4 The Homology of *k*-SAT

In *Table 4* is most of the data we have been able to deduce about the complexes produced by k-SAT. Note by H_i we really mean $\mathbb{Q} \otimes H_i$, and H_0 is assumed to be $\mathbb{Q}$ for all cases. Where actual topological spaces are listed, the implied meaning is the corresponding complex is homotopy equivalent to it. The homotopy type of the *2*-SAT with 3 variables complex was computed using a by hand calculation on the nerve produced by the method outlined in Section 4.4.3.

Computing the Homology and Euler characteristics of k-SAT complexes is a very difficult problem. Even generating this much of the table required many techniques. Computing the homology is difficult as we must deal with very large sparse matrices. While the computations needed to find the Euler characteristic are not as bad, they too quickly become impractical.

After computing the first couple values in this table, it seems as there is an obvious pattern, where the Euler characteristic of *2*-SAT with k variables is $2^{2^{k-2}}$ and *3*-SAT with k variables is $2^{3^{k-3}}$. But this pattern is clearly broken at *2*-SAT with 5 variables, which is 106 instead of 256, and *3*-SAT with 5 variables, which is 58 and not 512.

Conjecture. $K_{n,2}$ is homotopy equivalent to a wedge of $(2n-3)!!$ spheres of dimension $2n-2$. Also $K_{n,3}$ is homotopy equivalent to a wedge of $(2n-3)!!-(2n-4)!!$ spheres of dimension $4n-6$.

Now we shall prove some theorems about various properties of k-SAT.

4.4.1 Homology of *1*-SAT

Note in *Figure 5* each vertex is labeled as its elementary formula instead of as a set of truth assigments.

Theorem 5. *$K_{n,1}$, the simplicial complex corresponding to 1-SAT with n variables, is homeomorphic to a $(k-1)$-sphere.*

Proof. *1*-SAT with n variables has elementary formulae,

$$x_1\,,\ \bar{x}_1\,,\ \ldots\,,\ x_n\,,\ \bar{x}_n.$$

And *1*-SAT with $n+1$ variables has elementary formulae,

$$\phi_1 = x_1\,,\ \phi_2 = \bar{x}_1\,,\ \ldots\,,\ \phi_{2n-1} = x_n\,,\ \phi_{2n} = \bar{x}_n\,,\ \phi_{2n+1} = x_{n+1}\,,\ \phi_{2n+2} = \bar{x}_{n+1}.$$

Note the set of satisfactions for each ϕ_i where $1 \leq i \leq 2n$ is the same as in the *1*-SAT with n variables case, except now this set is twice as large, as each satisfaction must be appended with T or F for the x_{n+1} variable. This however does not change how the x_{ϕ_i} for $1 \leq i \leq 2n$ intersect with each other, so the complex generated by them is the same as $K_{n,1}$, the complex corresponding to *1*-SAT with n variables.

Now we want to investigate the effects of adjoining $\phi_{2n+1} = x_{n+1}$ and $\phi_{2n+2} = \bar{x}_{n+1}$. Note x_{n+1} is satisfied by any $t \in \mathcal{T}_{n+1}$ which has x_{n+1} set to true. So $x_{\phi_{2n+1}}$ forms a $(k+1)$-simplex on every k-simplex in the complex generated by x_{ϕ_i} for $1 \leq i \leq 2n$. Similarly, $x_{\phi_{2n+2}}$ also forms a $(k+1)$-simplex on every k-simplex in that complex. Finally, we note $\phi_{2n+1} = x_{n+1}$ and $\phi_{2n+2} = \bar{x}_{n+1}$ have no mutual satisfactions, so there is no edge between $x_{\phi_{2n+1}}$ and $x_{\phi_{2n+2}}$.

Therefore it is clear *1*-SAT with $n+1$ variables is homeomorphic to the suspension of *1*-SAT with n variables.

It is easy to show the complex corresponding to *1*-SAT with 1 variable is S^0 as it just has elementary formulae $\phi_1 = x_1$ and $\phi_2 = \bar{x}_1$ which have satisfactions $x_{\phi_1} = \{T\}$ and $x_{\phi_2} = \{F\}$. These are just two disjoint vertices.

Therefore by induction and the fact that the suspension of an n-sphere S^n is homeomorphic to S^{n+1}, we see the complex corresponding to *1*-SAT with n variables is homeomorphic to S^{n-1} for each n. □

4.4.2 Homology of k-SAT with k variables

The following theorem tells us what the homology of the first k-SAT with k variables for each k.

Theorem 6. *$K_{k,k}$, the simplicial complex corresponding to k-SAT with k variables, is homotopy equivalent to a $(2^k - 2)$-sphere.*

Proof. The elementary formulae for k-SAT with k variables are of the form $x_1 \vee x_2 \vee \cdots \vee x_n$, where some of the variables are negated. This only excludes one truth assignment, in this case $x_1 = F, x_2 = F, \ldots, x_n = F$, when there are no negated variables. Furthermore every truth assignment is excluded by one such formula. In our case $|\mathbb{F}| = 2^k$, so $\mathcal{A}$ consists of all subsets of $\mathbb{F}$ of size $2^k - 1$. By Theorem 4, this is homotopy equivalent to a sphere of dimension $2^k - 2$. □

4.4.3 Other Properties of k-SAT

There is a technique we may use to reduce almost every k-SAT complex near the beginning of the table into a simpler homotopy equivalent complex. We start by defining a cover $\mathcal{U}$ of the $K_{n,k}$ complex defined by $\mathcal{U} = \bigcup_{t \in \mathcal{T}_n} \mathcal{U}_t$, where $\mathcal{U}_t$ is the simplex formed by all the vertices that correspond to elementary formulae that are satisfied by t.

Note all the $\mathcal{U}_t$ have either empty or contractible intersections since simplices in simplicial complexes always do. Similar to in the proof of Theorem 1, we may take arbitrarily small neighborhoods around the $\mathcal{U}_t$ to make them open. Now we can apply Corollary 4.G.3 from [Hat06] to show $K_{n,k}$ is homotopy equivalent to the nerve $N\mathcal{U}$.

Using this nerve greatly simplifies things in most cases. For instance, in $K_{4,3}$ the largest C_i dimension is $410,894,304$, but the corresponding nerve complex had a largest C_i dimension of only $12,868$.

Also the nerve complexes have an interesting structure. The nerve of $K_{n,k}$ is actually homeomorphic to the $K_{\mathcal{A}}$ simplicial complex you get by taking $\mathbb{F}$ to be set $k-1$ dimensional faces of an an n dimensional octahedron and $\mathcal{A}$ to be the set of complements of $n-1$ dimensional faces. Unfortunately, it is still very difficult to calculate the homotopy type of these spaces.

In *Table 5*, we compare the times for computing the Euler characteristics of the original and nerve complexes corresponding to the given k-SAT complexes.

5 Conclusions and Recommendations

In conclusion, the study of computer science via topology appears to be an intriguing area of study. By studying the simplicial complexes we produced from k-SAT, it may be possible to gain new understanding into other areas of computer science such as whether or not P = NP. Also these k-SAT complexes are interesting in their own right, as at the current time there appears to be no clear pattern between them.

The greatest difficulty encountered in studying this material is the sheer need for computing power. Even as little as 15 years ago, a lot of the results reached would have been unattainable.

To any reader who is interested in continuing this research, I would encourage them to become proficient in a middle-level programming language such as C++, as high-level languages such as Mathematica and Maple are too cumbersome to quickly carry out the complicated computations necessary. Also, the "brute force" approach to programming doesn't really work here, as from my experience in the project, inefficient algorithms are likely to produce no interesting results.

5.1 C++ versus Mathematica

During the course of this project, both C++ and Mathematica versions of the programs were made. The project was started in Mathematica, but due to speed and memory problems, the code had to be ported to C++. In *Table 6* we compare the times taken by each program.

Note that C++ outperforms Mathematica in every case except for the 10 random $K_{\mathcal{A}}$ for $|\mathbb{F}| = 5$ one. The reason for this is the matrices produced for this case are just small enough to be stored entirely in computer memory, so Mathematica is able to efficiently run its nonsparse matrix rank algorithm. C++, on the other hand, uses a sparse algorithm, which is slower. The value of the sparse rank algorithm becomes apparent in the next case, however, which Mathematica is unable to complete.

References

[Fre98a] Michael H. Freedman. Limit, logic, and computation. *Proceedings of the National Academy of Sciences*, pages 95–97, 1998.

[Fre98b] Michael H. Freedman. Logic, P/ NP, and the quantum field computer. *Proceedings of the National Academy of Sciences*, pages 98–101, 1998.

[Fre99] Michael H. Freedman. K-sat on groups and undecidability. *Symposium on Theory of Computing*, pages 572–576, 1999.

[Hat06] Allen Hatcher. *Algebraic Topology*. Cambridge University Press, New York, seventh edition, 2006.

[KB00] J. Kouneiher and A.P.M. Balan. Propositional manifolds and logical cohomology. Festschrift in honor of Newton C. A. da Costa on the occasion of his seventieth birthday. *Synthese*, pages 147–154, 2000.

[Lin07] Project Linbox: Exact computational linear algebra. http://www.linalg.org, 2007.

[Sch95] Herbert Schild. *C: The Complete Reference*. Osborne McGraw-Hill, California, third edition, 1995.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Massachusetts, special edition, 2000.

[Wol91] Stephen Wolfram. *Mathematica: A System for doing Mathematics by Computer*. Addison Wesley, Massachusetts, second edition, 1991.

Appendices

Appendix A: Tables

Table 1: Homology calculated for $K_{\mathcal{A}}$ where $|\mathbb{F}| = 3$ or 4.

Homology	occurrences when $\lvert\mathbb{F}\rvert = 3$	occurrences when $\lvert\mathbb{F}\rvert = 4$
$H_0 = \mathbb{Q}$	6	610
$H_0 = \mathbb{Q}^2$	0	3
$H_0 = \mathbb{Q}$ $H_1 = \mathbb{Q}$	1	307
$H_0 = \mathbb{Q}$ $H_1 = \mathbb{Q}^2$	0	38
$H_0 = \mathbb{Q}$ $H_1 = \mathbb{Q}^3$	0	1
$H_0 = \mathbb{Q}$ $H_2 = \mathbb{Q}$	0	64

Table 2: Homology calculated for $K_{\mathcal{A}}$ for some random $\mathcal{A}$s with $|\mathbb{F}| = 5$.

Homology	$\|\mathbb{F}\| = 5$
$H_0 = \mathbb{Q}$	53
$H_0 = \mathbb{Q}$ $H_1 = \mathbb{Q}$	3
$H_0 = \mathbb{Q}$ $H_2 = \mathbb{Q}$	9
$H_0 = \mathbb{Q}$ $H_3 = \mathbb{Q}$	34
$H_0 = \mathbb{Q}^2$	1

Table 3: Subsets of $\widehat{K}_{\mathcal{A}}$.

subsets of size k	1σ	2σ	$\dots$	$\{n-k+1\,,\dots,\,n\}$
subsets of size $k-1$	1σ	2σ	$\dots$	$\{n-k\,,\dots,\,n\}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
subsets of size 2	1σ	2σ	$\dots$	$\{n-1\,,\,n\}$
subsets of size 1	1	2	$\dots$	n

Table 4: k-SAT.

# of Vars	*1*-SAT	χ	*2*-SAT	χ	*3*-SAT	χ	*4*-SAT	χ
1	S^0	2						
2	S^1	0	S^2	2				
3	S^2	2	$S^4 \vee S^4 \vee S^4$	4	S^6	2		
4	S^3	0	$H_6 = \mathbb{Q}^{15}$	16	$H_{10} = \mathbb{Q}^7$	8	S^{14}	2
5	S^4	2	?	106	?	58	?	11

Table 5: Original and nerve Euler characteristic computation times.

Test	Original Time (sec)	Nerve Time (sec)
2-SAT with 3 variables	0.004	0.003
2-SAT with 4 variables	1.288	0.024
2-SAT with 5 variables	18013.410	385.956
3-SAT with 4 variables	1009.635	0.064

Table 6: Mathematica and C++ homology computation time comparison.

Test	C++ time (sec)	Mathematica time (sec)
2-SAT with 3 variables	0.344	19.781
All $K_{\mathcal{A}}$ for $\lvert\mathbb{F}\rvert = 4$	1.040	58.360
10 random $K_{\mathcal{A}}$ for $\lvert\mathbb{F}\rvert = 5$	604.634	350.234
1 random $K_{\mathcal{A}}$ for $\lvert\mathbb{F}\rvert = 6$	6775.567	OUT OF MEMORY!

Appendix B: Figures

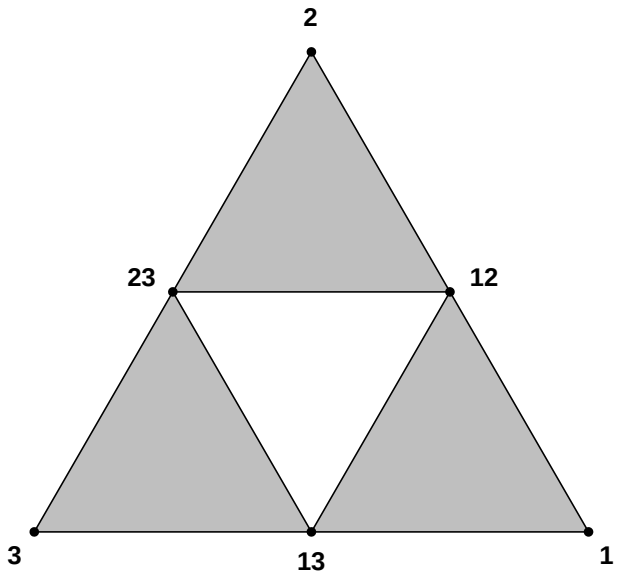

Figure 1: The simplicial complex $K_{\mathcal{A}}$ generated by all the proper subsets of a set of order 3.

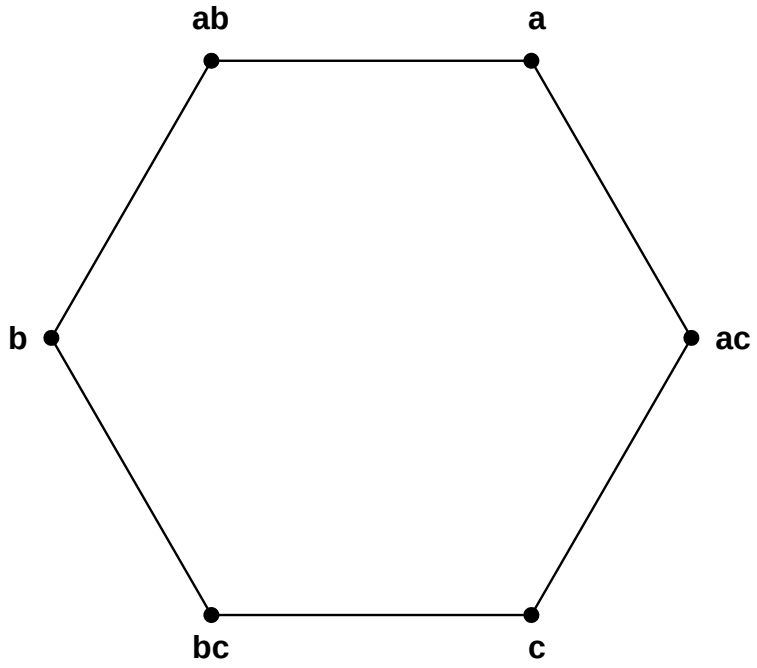

Figure 2: The geometric realization of the set of proper subsets of a 3 element set.

```
a = new array of size |𝔽|;
p = new array of size |𝔽|;
I = new array of size |𝔽|;


p_0 = first element of 𝒜;
i = 0;
while(i ≥ 0)
{
   a_i = p_i;
   p_i = element of 𝒜 after a_i;

   if(i > 0) I_i = I_{i-1} ∩ a_i;
   else I_i = a_i;
   if(I_i ≠ ∅)
   {
      add a_0 ... a_i as an i simplex;
      if(p_i ≠ ∅ and i < |𝔽| − 1)
      {
         p_{i+1} = p_i;
         i = i + 1;
      }
   }
   if(p_i = ∅) i = i − 1;
}
```

Figure 3: Pseudocode for the algorithm.

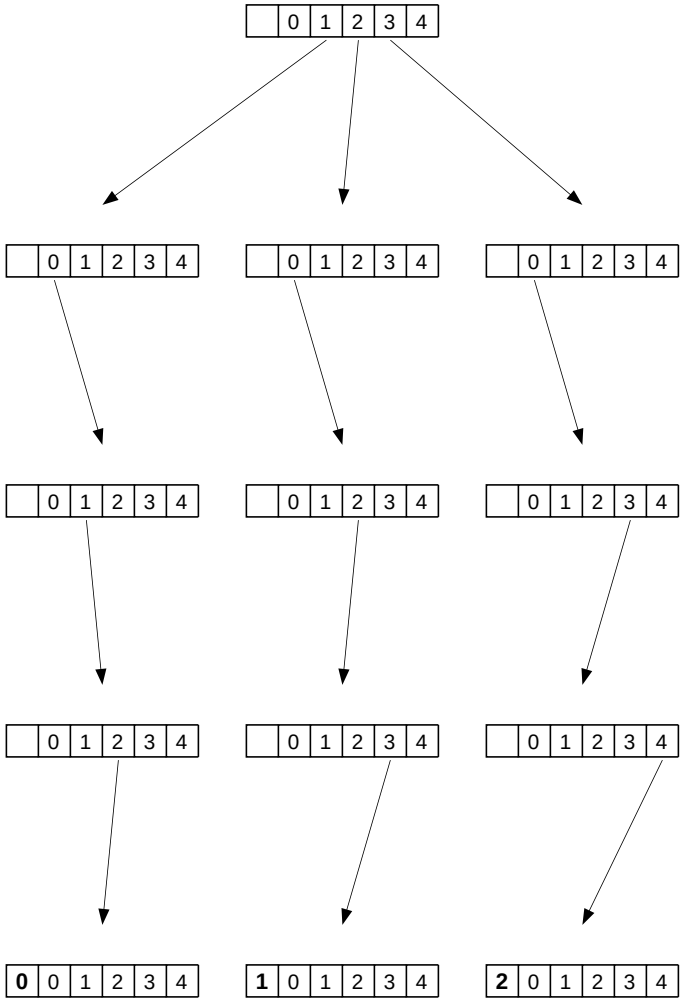

Figure 4: Chain group tree storage example.

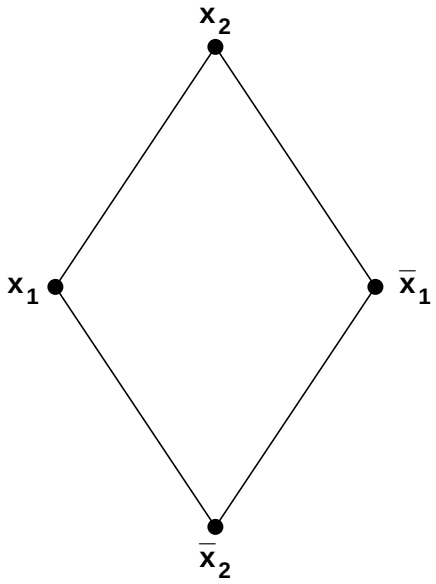

Figure 5: $K_{2,1}$ shown to be the suspension of $K_{1,1}$.

Appendix C: C++ Source Code

findCi.cpp

```
#include <cstdio>

#include "homology.h"

ciNode **Ci;
ciNode **CiPointer;

ciNode **makeCi(int sizeF, int sizeA, int *A)
{
  /* This is the initalization stage */
  int i; bool intersPrev;
  int **a = new int*[sizeF];

  for(i=0; i<sizeF; i++)
  {
    a[i] = NULL;
  }

  int **aPointer = new int*[sizeF];
  for(i=0; i<sizeF; i++)
  {
    aPointer[i] = NULL;
  }

  int **inters = new int*[sizeF];
  for(i=0; i<sizeF; i++)
  {
    inters[i] = NULL;
  }

  initCi(sizeF);

  int level;

  level = 0;
  aPointer[0] = A;

  /* This while loop will calculate chain groups of A. */
  while(1)
  {

    /* First we grab an element from aPointer.  */
    a[level] = aPointer[level];
    /* Now increment aPointer.  */
    aPointer[level] = aPointer[level] + (sizeF+1);

    intersPrev = true;

    /* Here we calculate the intersection of a with all the previous as.  */
    if(level > 0)
    {
      if(inters[level] != NULL) delete [] (inters[level]);

      inters[level] = intersection(inters[level-1],a[level], sizeF);
    }
    else
      inters[0] = a[0];
```

```
    if (inters[level][0] == 0) intersPrev = false;


    /* If a does intersect all the previous as:  */
    if(intersPrev)
    {
      addtoCi(a, level);

      if(aPointer[level][0] != -1 && level + 1< sizeF)
      {
        aPointer[level+1] = aPointer[level];
        level++;
      }
    }

    /* If we have reached the end of A we go back a level.  */
    if(aPointer[level][0] == -1)
    {
      level--;
    }

    /* Finally we break if level goes below 0.  */
    if(level < 0) break;
  }

  /* Finally we do some garbage collection.  */
  for(i=1; i<sizeF; i++)
  {
    if(inters[i] != NULL) delete [] inters[i];
  }
  delete [] a;
  delete [] aPointer;
  delete [] inters;

  return(Ci);
}

void initCi(int sizeF)
{
  Ci = new ciNode*[sizeF];
  CiPointer = new ciNode*[sizeF];


  for(int i=0; i < sizeF; i++)
  {
    Ci[i] = new ciNode(i+1);
    CiPointer[i] = Ci[i];
  }
}
void addtoCi(int **a, int level)
{

  for(int i=0; i <= level; i++)
    (CiPointer[level]->complex)[i] = a[i];

  CiPointer[level] -> next = new ciNode(level+1);
  CiPointer[level] = CiPointer[level] -> next;
}
```

```
/* This function calculates the intersection of cells a and b.  */
int *intersection(int *a, int *b, int sizeF)
{
  int i; int j; int k;
  int *inter; int min;

  inter = new int[sizeF+1];

  i = 0; j = 0; k = 0;
  while(a[i] != 0 && b[j] != 0)
  {
    if(a[i] < b[j]) i++;
    else if (a[i] > b[j]) j++;
    else
    {
      inter[k] = a[i];
      i++; j++; k++;
    }
  }
  inter[k] = 0;

  return(inter);
}
```

findHomology.cpp

```
#include <cstdio>
#include "homology.h"

/* The needed Linbox libraries.  */
#include "linbox/field/gf2.h"
#include "linbox/field/gmp-integers.h"
#include "linbox/blackbox/sparse.h"
#include "linbox/blackbox/zero-one.h"
#include "linbox/solutions/rank.h"

using namespace LinBox;
using namespace std;

/* Prototypes */
SparseMatrix<PID_integer> *findBMatrix(int i, int im1, int sizeF, ciNode **Ci, int *CiLength);

/* This will return the homology of A. */
long unsigned int *findHomology(int sizeF, int sizeA, int *A)
{
  int i, j; ciNode **Ci; int *CiLength;
  SparseMatrix<PID_integer> *BMatrix;
  long unsigned int *homology = new long unsigned int[sizeF];

  commentator.setMaxDetailLevel (-1);
  commentator.setMaxDepth (-1);
  commentator.setReportStream (std::cerr);

  /* Now we calculate the chain groups.  */
  Ci = makeCi(sizeF, sizeA, A);
  CiLength = findCiLengths(Ci, sizeF);

  long unsigned int BRank[sizeF];
  for(i=0; i<sizeF; i++)
  {
    BRank[i] = 0;
    homology[i] = -1;
  }


  /* Here we calculate the boundary matrices and find their ranks using Linbox.  */
  for(i=1;i<sizeF;i++)
  {
    BMatrix = findBMatrix(i-1,i, sizeF, Ci, CiLength);

    if(BMatrix != NULL)
    {
      rank (BRank[i], *BMatrix);
      delete BMatrix;
    }
    delete Ci[i-1];

  }
  delete Ci[sizeF-1];

  /* Finally we compute the homology.  */
  for(i=0; i<sizeF-1; i++)
    homology[i] = CiLength[i] - BRank[i] - BRank[i+1];
```

```
  delete [] CiLength;

  return(homology);
}


/* This function contructs the boundary matrix between two chain groups.  */
SparseMatrix<PID_integer> *findBMatrix(int im1, int i, int sizeF, ciNode **Ci, int *CiLength)
{
  int **Matrix;
  int **complex;
  int *temp[im1];
  ciNode *p;
  int j, k, m, n, pos;

  if(CiLength[i] == 0 || CiLength[im1] == 0) return(NULL);

  PID_integer ZZ;
  SparseMatrix<PID_integer> *BMatrix = new SparseMatrix<PID_integer>(ZZ, CiLength[i], CiLength[im1]);

  treeNode *Tree = makeTree(im1, sizeF, Ci[im1]);

  p = Ci[i];
  for(j=0; p->next != NULL; j++)
  {
    complex = p->complex;

    for(k=0; k<=i; k++)
    {
      n=0;
      for(m=0; m <= i; m++)
        if(m != k)
        {
          temp[n] = complex[m];
          n++;
        }

      pos = findinTree(im1, temp, Tree);

      if(k        BMatrix->setEntry(j, pos, 1);
      else
        BMatrix->setEntry(j, pos, -1);
    }

    p = p->next;
  }

  Tree->delTree(im1);
  delete Tree;
  return(BMatrix);
}

/* This function turns Cim1 into a tree for easier access to finding elements positions in it.  */
treeNode *makeTree(int im1, int sizeF, ciNode *C)
{
  int pos;
  treeNode *Tree = new treeNode(sizeF);

  ciNode *p = C;
```

```
  for(pos=0;p->next != NULL; pos++)
  {
    addtoTree(im1, pos, sizeF, Tree, p);
    p = p->next;
  }
  return(Tree);
}

/* Here we search Tree and return the position of complex in it.  */
int findinTree(int im1, int **complex, treeNode *Tree)
{
  int i, j;
  treeNode *Treep = Tree;
  int *cell;

  for(i=0; i <= im1; i++)
  {
    cell = complex[i];

    for(j=0; cell[j] != 0; j++)
    {
      Treep = (Treep->branch)[ cell[j] ];
    }

    if(i != im1) Treep = (Treep->branch)[0];
  }
  return(Treep->val);
}

/* This adds a complex to our Cim1 tree.  */
void addtoTree(int im1, int pos, int sizeF, treeNode *Tree, ciNode *p)
{
  int i, j;
  treeNode *Treep = Tree;
  int *cell;

  for(i=0; i<=im1; i++)
  {
    cell = (p->complex)[i];

    for(j=0; cell[j] != 0; j++)
    {
      if( (Treep->branch)[ cell[j] ] == NULL)
        (Treep->branch)[ cell[j] ] = new treeNode(sizeF);

      Treep = (Treep->branch)[ cell[j] ];
    }

    if(i != im1)
    {
      if( (Treep->branch)[0] == NULL)
        (Treep->branch)[0] = new treeNode(sizeF);

      Treep = (Treep->branch)[0];
    }
  }
  Treep->val = pos;
}
```

main.cpp

```
#include <cstdio>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstring>
#include "homology.h"

using namespace std;

int sizeF; int sizeA;

int main(int argc, char *argv[])
{
  int *A; int i; int j; int x; sizeA = 0;
  int counter; int ccounter = 0;
  streampos sp;
  ifstream inFile;
  long unsigned int *homology;



  if(argc != 2)
  {
    printf("Usage is:  findHomology filename\nwhere filename is a file containing A.\n");
    exit(1);

  }

  inFile.open(argv[1]);

  if (!inFile) {
    cout << "Error:  Unable to open file.\n";
    exit(1); // Terminate with error.
  }


  counter = 0;
  sp = inFile.tellg();
  /* This loop enables us to read in multiple As.  */
  while(inFile >> x)
  {
    /* Here we find the number of cells in A and the size of F. */
    sizeA = 0; sizeF = 0;

    while(x != -1)
    {
      if(x > sizeF) sizeF = x;
      if(x == 0) sizeA++;

      inFile >> x;
    }

    inFile.seekg(sp);

    /* Now we initalize A. */
    A = new int[sizeA * (sizeF+1) + 1];
    A[sizeA * (sizeF+1)] = -1;
```

```
    for(i=0; i < sizeA * (sizeF+1); i++)
      A[i] = 0;


    /* Finally we put the cells in it.  */

    i=0;j=0;

    inFile >> x;
    while(x != -1)
    {
      A[j*(sizeF+1)+i] = x;


      i++;
      if(x==0)
      {
        i = 0; j++;
      }
      inFile >> x;
    }


    /* Now we run the program on A. */
    homology = findHomology(sizeF, sizeA, A);

    printA(A);
    for(i=0; i < sizeF-1; i++)
    {
      printf("H%d is Q%d\n", i, homology[i]);
    // fprintf(stderr, "H%d is Q%d\n", i, homology[i]);

    }

    sp = inFile.tellg();
    delete [] A;
    delete [] homology;
  }

  return(0);
}

void printA(int *A)
{

  /* Here we print A */
  printf("A = ");
  for(int i=0; i < sizeA; i++)
  {
    printf("{");
    printCell(A+ i * (sizeF+1) );
    printf("}");
    if(i+1<sizeA) printf(" , ");

  }
  printf("\n");
}

/* This function contructs A from a file.  */
```

```
int *makeA(char *filename)
{
  int i, j; int *A;
  ifstream inFile;

  inFile.open(filename);
  if (!inFile) {
    cout << "Error:  Unable to open file.\n";
    exit(1); // Terminate with error.
  }

  /* Here we read in an A. */

  int x; i = 0; sizeA = 0;

  /* Here we find the number of cells in A and the size of F. */
  sizeA = 0; sizeF = 0;
  while(inFile >> x)
  {
    if(x > sizeF) sizeF = x;
    if(x == 0) sizeA++;
  }

  inFile.close();

  /* Now we initalize A. */
  A = new int[sizeA * (sizeF+1) + 1];
  A[sizeA * (sizeF+1)] = -1;

  for(i=0; i < sizeA * (sizeF+1); i++)
    A[i] = 0;

  /* Finally we put the cells in it.  */
  inFile.open(filename);

  i=0;j=0;
  while(inFile >> x)
  {
    A[j*(sizeF+1)+i] = x;

    i++;
    if(x==0)
    {
      i = 0; j++;
    }

  }

  inFile.close();

  return(A);
}
```

misc.cpp

```
#include <cstdio>
#include "homology.h"

void printCell(int *a)
{
  int i;
  if(a != NULL) {
    for(i=0; a[i] != 0;i++)
    {
      printf("%d",a[i]);
      if(a[i+1] != 0) printf(" ");
    }
  }
  else printf("NULL");
}

int *findCiLengths(ciNode **Ci, int sizeF)
{
  int i;
  int *CiLength = new int[sizeF];
  int length;
  ciNode *p;

  for(i=0; i<sizeF;i++)
  {
    p = Ci[i];
    for(length=0; p->next != NULL; length++) p = p->next;
    CiLength[i] = length;
  }
  return(CiLength);
}

void printCi(ciNode **Ci, int sizeF)
{
  ciNode *p;
  int i, j;

  for(i=0; i < sizeF; i++)
  {
    p = Ci[i];
    if(p->next != NULL) printf("\nC%d is:  ", i);

    while(p->next != NULL)
    {
      printf("{");
      for(j=0; j<=i; j++)
      {
        printCell((p->complex)[j]);
        if(j!=i) printf(",");
      }
      printf("}");
      if(p->next->next != NULL)printf(" , ");
      p = p->next;
    }
    printf("\n");
  }
}
```

homology.h

```
/* The Ci Tree node class.  */
class treeNode
{
  public:
    treeNode(int size)
    {
      int i;
      branch = new treeNode*[size+1];

      for(i=0; i<size+1; i++)
        branch[i] = NULL;

      val = -1;
    }
    void delTree(int size)
    {
      int i;

      for(i=0; i<size+1; i++)
        if(branch[i] != NULL)
      {
        branch[i]->delTree(size);
        delete branch[i];
      }

      delete [] branch;
    }

    treeNode **branch;
    int val;
};

/* the node for the linked list each Ci will be stored in.  */
class ciNode
{
  public:
    ciNode(int size)
    {
      int i;
      complex = new int*[size];

      for(i=0; i<size; i++)
        complex[i] = NULL;

      next = NULL;
    }
    ciNode()
    {
      delete [] complex;
      if(next != NULL) delete next;
    }

    int **complex;
    ciNode *next;
};
```

```
/*Function prototypes */
void printA(int *A);
int *makeA(char *filename);
long unsigned int *findHomology(int sizeF, int sizeA, int *A);
void initCi(int sizeF);
void addtoCi(int **a, int level);
void printCi(ciNode **Ci, int sizeF);
int *findCiLengths(ciNode **Ci, int sizeF);
ciNode **makeCi(int sizeF, int sizeA, int *A);
int *intersection(int *a, int *b, int sizeF);
void printCell(int *a);
void addtoTree(int im1, int pos, int sizeF, treeNode *Tree, ciNode *p);
treeNode *makeTree(int im1, int sizeF, ciNode *C);
int findinTree(int im1, int **complex, treeNode *Tree);
```

findEuler.cpp

```
#include <cstdio>
#include "homology.h"

/* This function calculates the Euler characteristic of A. */
long int findEuler(int sizeF, int sizeA, int *A)
{
  /* This is the initalization stage */
  int i; bool intersPrev;

  int **a = new int*[sizeA];
  for(i=0; i<sizeA; i++)
  {
    a[i] = NULL;
  }

  int **aPointer = new int*[sizeA];
  for(i=0; i<sizeA; i++)
  {
    aPointer[i] = NULL;
  }

  int **inters = new int*[sizeA];
  for(i=0; i<sizeA; i++)
  {
    inters[i] = NULL;
  }

  long unsigned int *CiCounter0 = new long unsigned int[sizeA];
  long unsigned int *CiCounter1 = new long unsigned int[sizeA];
  for(i=0; i<sizeA; i++)
  {
    CiCounter0[i] = 0;
    CiCounter1[i] = 0;
  }

  int level;

  level = 0;
  aPointer[0] = A;


  /* This while loop will calculate chain groups of A. */
  while(1)
  {

    /* First we grab an element from aPointer.  */
    a[level] = aPointer[level];
    /* Now increment aPointer.  */
    aPointer[level] = aPointer[level] + (sizeF+1);

    intersPrev = true;

    /* Here we calculate the intersection of a with all the previous as.  */
    if(level > 0)
    {
      if(inters[level] != NULL) delete [] (inters[level]);
```

```
    inters[level] = intersection(inters[level-1],a[level], sizeF);
  }
  else
    inters[0] = a[0];

  if (inters[level][0] == 0) intersPrev = false;

  /* If a does intersect all the previous as:  */
  if(intersPrev)
  {
    CiCounter0[level]++;
    if(CiCounter0[level] == 1000000)
    {
      CiCounter1[level]++;
      CiCounter0[level] = 0;
    }


    if(aPointer[level][0] != -1 && level + 1< sizeA)
    {
      aPointer[level+1] = aPointer[level];
      level++;
    }
  }

  /* If we have reached the end of A we go back a level.  */
  if(aPointer[level][0] == -1)
  {
    level--;
  }

  /* Finally we break if level goes below 0.  */
  if(level < 0) break;
}

/* Here we calculate the Euler characteristic */
long int X0 = 0;
long int X1 = 0;

for(i=0; i<sizeA; i++)
{

  printf("C%d has dim %d %d.\n",i, CiCounter1[i], CiCounter0[i]);
  if(i    {
    X0 = X0 + CiCounter0[i];
    X1 = X1 + CiCounter1[i];
  }
  else
  {
    X0 = X0 - CiCounter0[i];
    X1 = X1 - CiCounter1[i];
  }
}
printf("\nThe 1st counter is the number of millions.\nThe 2nd counter is the remainder.\n");


/* Finally we do some garbage collection.  */
for(i=1; i<sizeA; i++)
{
  if(inters[i] != NULL) delete [] inters[i];
```

```
  }

  delete [] a;
  delete [] aPointer;
  delete [] CiCounter0;
  delete [] CiCounter1;
  delete [] inters;

  /* Returns the Euler characteristic.  */
  return(X1*1000000 + X0);
}

/* This function calculates the intersection of cells a and b.  */
int *intersection(int *a, int *b, int sizeF)
{
  int i; int j; int k;
  int *inter; int min;

  inter = new int[sizeF+1];

  i = 0; j = 0; k = 0;
  while(a[i] != 0 && b[j] != 0)
  {
    if(a[i] < b[j]) i++;
    else if (a[i] > b[j]) j++;
    else
    {
      inter[k] = a[i];
      i++; j++; k++;
    }
  }
  inter[k] = 0;

  return(inter);
}
```

Vita

Oliver Thistlethwaite was born in London, England, but moved to the United States as a child. He first attended the University of Tennessee of Knoxville as an undergraduate in Computer Science, but would later change his major to Mathematics.

In his last year, he was first introduced to topology in a senior level class taught by Dr. James Conant, who would later go on to become his thesis advisor. After graduating in spring 2005 with a major in Mathematics and a minor in Computer Science, Oliver enrolled as a graduate student in Mathematics. After completing several more topology courses, he began writing his thesis and assisting Dr. Conant in research.