Spring 2016

Analyze Large Multidimensional Datasets Using Algebraic Topology

David Le
San Jose State University

SAN JOSE STATE UNIVERSITY

MASTER'S WRITING PROJECT

Analyze Large Multidimensional Datasets Using Algebraic Topology

Author:
David A. Le

Advisor:
Dr. Tsau Young Lin

Committee Members:
Dr. Jon Pearce
Mr. James Casaletto

A writing project submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department of Computer Science

May 12, 2016

SAN JOSE STATE UNIVERSITY

MASTER'S WRITING PROJECT

Abstract

This paper presents an efficient algorithm to extract knowledge from high-dimensionality, high-complexity datasets using algebraic topology, namely simplicial complexes. Based on concept of isomorphism of relations, our method turn a relational table into a geometric object (a simplicial complex is a polyhedron). So, conceptually association rule searching is turned into a geometric traversal problem. By leveraging on the core concepts behind *Simplicial Complex*, we use a new technique (in computer science) that improves the performance over existing methods and uses far less memory. It was designed and developed with a strong emphasis on scalability, reliability, and extensibility. This paper also investigate the possibility of Hadoop integration and the challenges that come with the framework.

SAN JOSE STATE UNIVERSITY

MASTER’S WRITING PROJECT

Acknowledgments

I would like to express my deepest gratitude to my research advisor, Dr. Tsau Young Lin, for his excellent guidance, direction, and support over the past several years. His profound knowledge has helped me overcome many challenges during the research phase. With his dedicated patience and vision, I’ve learned to tackle complex problems from many different angles. Thank you Dr. Lin.

I also would like to thank my committee members, Dr. Jon Pearce and Dr. James Casaletto, for providing me with many thoughtful suggestions and feedbacks on the project. Their expertise and experience in the areas of Mathematics and Computer Science have given me a tremendous boost. I couldn’t have been more fortunate to have this amazing opportunity.

I would like to mention my colleague, my friend and my mentor, Mike Mittmann, who I’ve always looked up and run to whenever I hit a roadblock. His ability to approach challenging problems and coming up with proper solutions at lightning speed is unprecedented. He is the first person that I turned to get some ideas on how to brainstorm the possible solutions to this complicated problem.

Finally, I would like to thank my wife for being by my side and encouraged me every step of the way. It’s not possible to complete this Master’s project without her unconditional love and support. Thank you so much “ba xa”. You truly mean the whole world to me.

Table of Contents

1. Introduction

Big data, small data, structured data, unstructured data. Data is everywhere. Data is all around us. To say that we are living in the information age is an understatement. It was estimated that the total amount of stored data in the world was approaching 8 zettabytes as of 2015 (1). That is equivalent to 10^{21} bytes or 8 trillion gigabytes of data. And we constantly generate massive amount of information on a daily basis. This size of data presents many challenges as well as wonderful opportunities for the scientific community. Not only we have to deal with its enormous size, but we also have to find innovative ways to handle its ever-increasing complexity.

In this paper, we present a mathematical concept in algebraic topology to help tackle aforementioned challenges in Data Science. Our method is based upon a branch of study in algebraic topology called *Simplicial Complex*. By using the main principles behind *Simplicial Complex*, intricate relationships within high-dimensional datasets can be extracted to give us useful insights and hidden knowledge about the data that we normally would miss otherwise. But there are several big hurdles that we must overcome. The problem is not in Mathematics itself, but instead it's in Computer Science, particularly when working with high-dimensional, high-complexity datasets. Sometimes what worked in theory does not always translate well into practice. Our novel approach must take all these measures into considerations. We were able to develop a solution that can utilize resources more effectively, improve the performance, and provide a solid framework to integrate with distributed systems for scalability.

1.1 Motivation

This work, led by Dr. Tsau Young Lin, is motivated by years of research at San Jose State University in the Department of Computer Science. There are many important problems that exist in the world today which can be solved if we have proper software tools to analyze large amount of data and produce meaningful results in a timely manner.

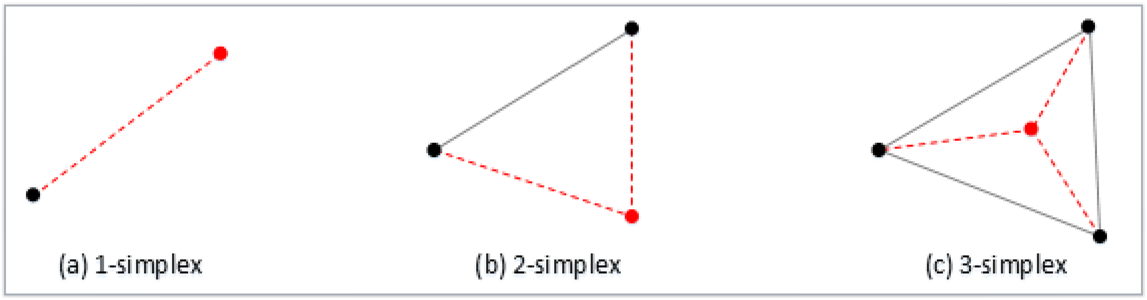
List of applicable applications:

- Genetic Diseases
- Business Data Analytics
- Computer Security
- Natural Language Processing (NLP)
- Space Exploration
- Image Recognition
- Effects of Social Media

As more Big Data are showing up everywhere, traditional systems for analyzing small datasets are fast becoming antiquated. Public sector, private sector, the government, and academia are all scrambling for better solutions to address the growing concerns as well as looking for potential opportunities. We understand and embrace these technical challenges from a humble perspective. This is the driving force behind our motivation.

1.2 Prior Work

Traditionally, simplicial homology defines a concept consisting of open simplexes of a specific dimension within a *Simplicial Complex* [1]. The classical notion of *Simplicial Complex* is to decompose a space into simplexes allowing different faces of a simplex to coincide and dropping the requirement that simplexes are uniquely determined by the vertices. The idea is very simple and straightforward. If we have a point in space, adding another point will result in a line that connects the two points. If we have a line in space, adding a point will create a triangle. And if we have a triangle in space, putting a point down will result in a tetrahedron.



Basically, all this is saying is that *n*-simplex would consist of $(n+1)$ distinct number of vertices, and that no other *n*-simplex has this same set of vertices [2]. It is fairly simple to grasp the concept; however, number of calculations that are needed to compute a high-dimensional dataset can increase exponentially. This can pose a tremendous challenge when attempting to analyze the simplex-pair associations for high-complexity datasets.

1.3 Contribution

Our research, which led by Dr. Tsau Young Lin for past several years, has been trying to come up with fresh perspectives to tackle this challenging problem by targeting it from different angles. We have developed several versions of the algorithm, with each one having certain advantages. We carefully studied the strengths of each one and tried to leverage what we've learned in the next revision. Our goal is to come up with a practical solution to the problem and then define a *Simplicial Complex* framework that can be used by the general community. Although we made a significant discovery on how to efficiently analyze and identify relationships within large datasets, there is still a lot of room for improvements in this area.

1.3.1 Our Approach

The purpose of this paper is to present a unique perspective on how to deal with the aforementioned problems. We take the basic principles that have been proven mathematically for *Simplicial Complex* and then apply the knowledge gained to address real-world data problems. We came up with a concept, called *Self-Contained Cone Construction*, to traverse the topological graph in such manner that limits the use of system's resources and boosts the overall performance of an application. Our approach is simple but effective. It was also designed to work well and can easily be integrated in a distributed computing environment to run with massive datasets. We will go over more details in the subsequent sections of this paper.

2. Simplicial Complex

2.1 Basic Definition

In algebraic topology, a simplicial complex consists of a finite set of simplexes that are glued together on a topological space to form a geometric structure in arbitrary dimensions. A simplicial complex K must satisfy the following conditions [18]:

1. Any face $\sigma \in K$ and any subset $\tau \subseteq \sigma$ implies that $\tau \in K$
2. If $\sigma_1, \sigma_2 \in K$ then $\sigma_1 \cap \sigma_2$ is either $\emptyset$ or a face of both

By definition, the empty set $\emptyset$ is a face of every simplex as stated in condition (1), and thus belongs to K. Condition (2) also allows the two faces to be completely disconnected from one another.

2.1.1 Simplexes

In geometry, a simplex represents the simplest possible polytope in any given space where the vertex of a polytope is a point in which polytope edges are joined. It generalizes the notion of a triangular or tetrahedral region of space to k dimensions. Thus a k-simplex is a k-dimensional polytope of a convex hull that consists of $(k + 1)$ vertices.

Let S be a finite set of $(k + 1)$ points $\{u_0, u_1, \ldots, u_k\}$ in k-dimensional Euclidean space $\mathbb{R}^{k+1}$ such that $\{u_1\text{-}u_0, u_2\text{-}u_0, \ldots, u_k\text{-}u_0\}$ are linearly independent. That is they are independent in an affine space of a geometric structure. Then a convex hull of these points is a k-simplex or Δ^k and is defined as follows:

$$C = \{\, \lambda_0 u_0 + \lambda_1 u_1 + \ldots + \lambda_k u_k \mid \sum_{i=0}^{k} \lambda_i = 1 \text{ where } \lambda_i \geq 0 \text{ for all } i \,\}$$

The boundary of this k-simplex has $(k+1)$ 0-faces or polytope vertices, $\left(\frac{k(k+1)}{2}\right)$ 1-faces or polytope edges, and $\binom{k+1}{i+1}$ i-faces where $\binom{n}{k}$ is a binomial coefficient. The number of i-faces of a k-simplex can also be located on Pascal's triangle with $(i+1)$ columns and $(k+1)$ rows, excluding the left diagonal.

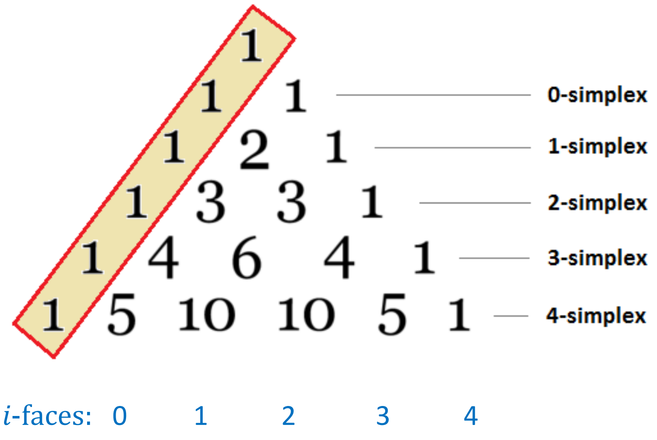


Figure 2.1.2: Pascal's triangle showing the number of i-faces up to 4-simplex.

2.1.1.1 Valid Simplexes

By definition described in 2.1.1, the standard *k*-simplex, or Δ^k, is the subset of $\mathbb{R}^{k+1}$ given by

$$\Delta^k = \{ (\lambda_0, \lambda_1, \dots, \lambda_k) \in \mathbb{R}^{k+1} \mid \sum_{i=0}^{k} \lambda_i = 1 \text{ where } \lambda_i \geq 0 \text{ for all } i \}$$

The simplex Δ^k lies in the affine hyperplane that can be obtained by removing the restriction of $\lambda_i \geq 0$ in the above equation [1]. The coefficients λ_i are also called the barycentric coordinates of a point in Δ^k. Thus this simplex is often referred to as an affine *k*-simplex. Figure 2.1.1.1 below shows a list of possible valid simplexes that form the basis of our *Self-Contained Cone Construction* simulation later.

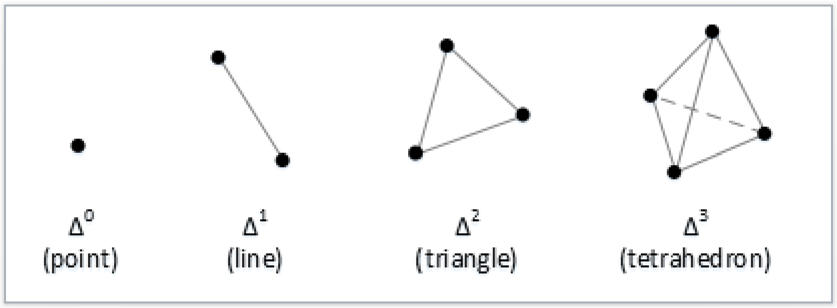


Figure 2.1.1.1: Dimensional space for valid simplexes starting from left: $\mathbb{R}^1$, $\mathbb{R}^2$, $\mathbb{R}^3$, $\mathbb{R}^4$.

2.1.1.2 Invalid Simplexes

Due to strict requirements of the algorithm that is being presented in this paper, we only operate on the premise that all simplexes must be labeled as valid. If a simplex somehow violates one or more conditions as stated in the above definition, it will be classified as invalid and thus will not be considered in the experiments.

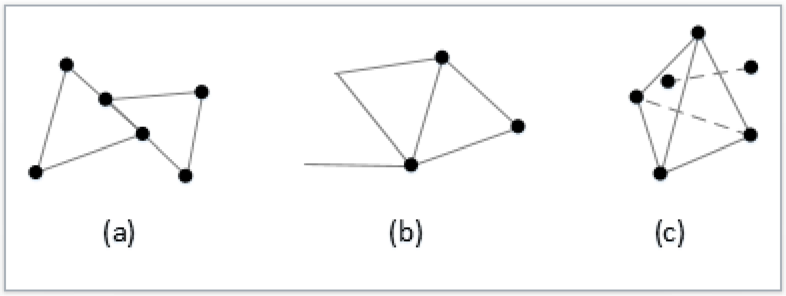


Figure 2.1.1.2: Examples of invalid simplexes that violate the definition of a Simplex.

We can observe that example (a) contains an overlapping edge that is not part of either of two triangles. The middle segment (edge) does not belong to either of the triangles. This clearly violates the first condition that was described in Section 2.1, *Basic Definition*, and so it cannot form a valid simplex. In example (b), there are two missing vertices and an edge connecting them. This also is not allowed. Lastly in example (c), the edge of a line is crossing the tetrahedron at an interior segment. This segment is neither an empty set nor it is a face of both, thus violating the second condition.

2.2 Algorithm

2.2.1 Data Mining

Data mining is a process of gathering, analyzing, and discovering of interesting patterns within large datasets. Data mining tools can be used to predict trends and behaviors, allowing data scientists and companies to make proactive, knowledge-driven decisions. This is why data mining is also referred to as knowledge discovery.

2.2.1.1 Feature Extraction

The process of transforming large complex datasets into a reduced set of desirable features is called *Feature Extraction* [1]. The purpose is to narrow down our focus to the information that is relevant for a particular area of interest. Analyzing large datasets with high variability will typically result in more memory usage and slower performance, not to mention overfitting that could lead to poorer statistical models. To illustrate what feature extraction is and how it can be applied to our *Simplicial Complex* model, we will take look at Table 2.2.1.1, which contains a list of 10 selected features. These features could be used to represent a variety of subjects. For example, they could be any of the followings:

- Words for understanding of human-computer interactions in NLP
- List of selected genes to study genetic diseases
- Stock symbols to analyze the strength of financial market
- Fortune-500 companies and their successful investments

		0	1	2	3	4	5	6	7	8	9
		A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**
T1	Tetrahedron 1	1	1	0	1	1	0	0	0	0	0
T2	Tetrahedron 2	0	1	1	1	1	0	0	0	0	0
T3	Tetrahedron 3	1	0	0	0	1	1	0	1	0	0
T4	Tetrahedron 4	1	0	0	0	0	1	1	1	0	0
T5	Tetrahedron 5	0	0	0	0	1	0	0	1	1	1
		.	.	.	.	.	.	.	.	.	.
		.	.	.	.	.	.	.	.	.	.
		.	.	.	.	.	.	.	.	.	.

Table 2.2.1.1: Example of a list of selected features extracted and projected onto a 2D table.

Each feature or column is assigned to a vertex in our *Simplicial Complex* model. This is equivalent to 0-simplex (Δ^0) on a topological space with no dimension. In other words, it is just a point. To keep things simple, we use a small dataset that contains only five rows of data. Each row is a tetrahedron, which is equivalent to 3-simplex (Δ^3) on a 3-dimensional topological space.

2.2.1.2 Simplex Representation

Given that each column in Table 2.2.1.1 is a vertex or a point in space, we can construct a graphic representation of that simple dataset with edges linking the vertices. This is depicted in Figure 2.2.1.2. The mapping results of those data points are considered isomorphic since they both describe the same object model but in different manners.

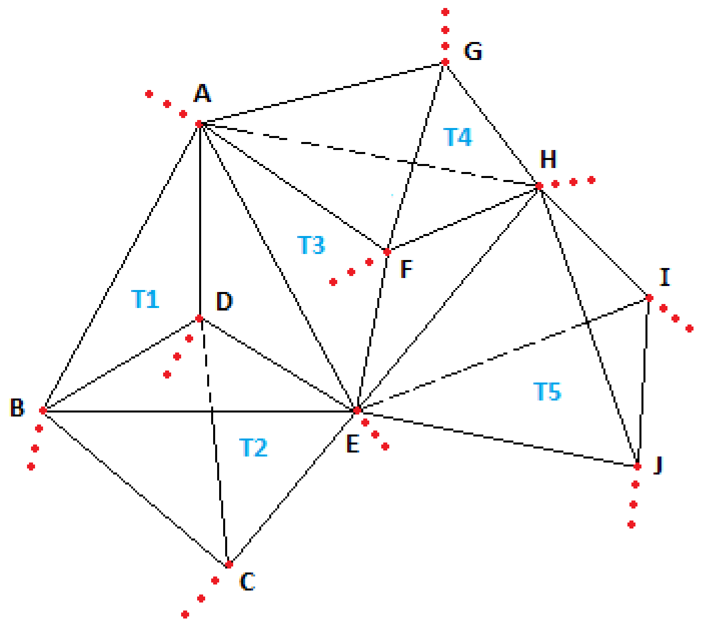


Figure 2.2.1.2: A graphical representation of Table 2.2.1.1.

The number of dimensions is directly proportional to the number of features or vertices on the graph. So the higher the dimension, the more difficult it is to analyze and identify the related patterns within the dataset. We defined each data row as a tetrahedron for simplicity. We can observe that there are a total of five 3-simplex objects that are labelled as **T1**, **T2**, **T3**, **T4** and **T5**. They are linked to each other by a number of different i-faces. For instance, **T1** and **T2** are connected by a 3-face element or triangle **BDE**, and similarly **T3** and **T5** are connected by a 2-face element or line **EH**. As one can imagine, the number of dimensions for each data row in Table 2.2.1.1 can be as high as the column size minus 1, which is 9 in this case. Since humans generally prefer to see objects in 3 dimensions or less, we purposely created each row of data as Δ^3. However in any practical dataset, the number of features can grow as large as required in order to generate a more robust statistical model, but the underlining principles of *Simplicial Complex* remain the same. Also in our simple example, each vertex consists of a maximum of 5 data rows or tetrahedrons. A typical real-world dataset would have a much larger number of rows, thus each vertex would contain large number of edges and that can be hard to visualize. This can be addressed by grouping the data points in each vertex into clusters and stretching them out to different areas on the graph. This was indicated by (• • •) in Figure 2.2.1.2.

2.2.1.3 Inverted Index

To help improve the speed of finding possible edges and links on the constructed graph, we use an inverted index table that was anchored at the record level. What this allows us to do is to quickly retrieve not only the data points that are associated with a given vertex, but also access the unified simplexes that occur during cone construction. We will dive into more details of what cone-construction process really is in the latter sections. Figure 2.2.1.3 shows how an inverted index was created for Table 2.2.1.1. The inverted-index structure is very crucial in how we will define our data model in the next section. Another step that utilizes this indexing table is *Vertex Unification* in order to track the associations among them.

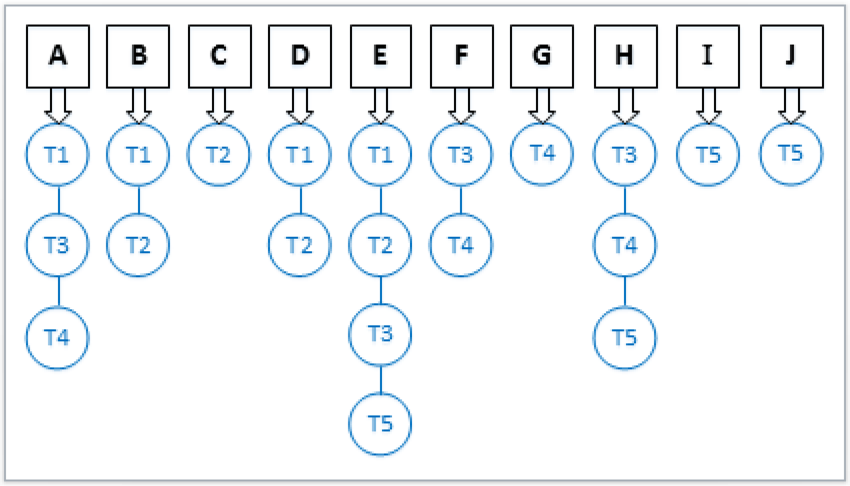


Figure 2.2.1.3: Inverted index structure that was constructed based on Table 2.2.1.1.

2.2.1.4 Data Structures

When developing a complex algorithm, data structures play a critical role and have tremendous impact on the efficiency of the algorithm. Different data structures are better suited for different types of applications while others are more specialized for specific tasks. From a mathematical perspective, *Simplicial Complex* is a proven concept with many solid principles to support the idea. It is not always easy to convert and translate that into applications without proper data structures.

We are introducing a unique data structure that was designed to address the many challenges such as resource allocations and performance. We will explain what each component is and how it was used at different stages of the algorithm. The key thing to take away from this data structure is that it was not built to store all the data in memory, but instead it was designed to handle the processing of just a single vertex and all of its dependencies at any given moment. What that means is during the cone construction, each vertex is not aware of the information that was previously computed for the visited vertices. This may seem counter-intuitive at first, but we will see later on that this process is remarkably efficient when it comes to data storage and memory management. Additionally, we will discuss more how algorithm was able to take advantage of this simplex structure to traverse the graph in such way that allows it to compute relationships among the vertices very quickly.

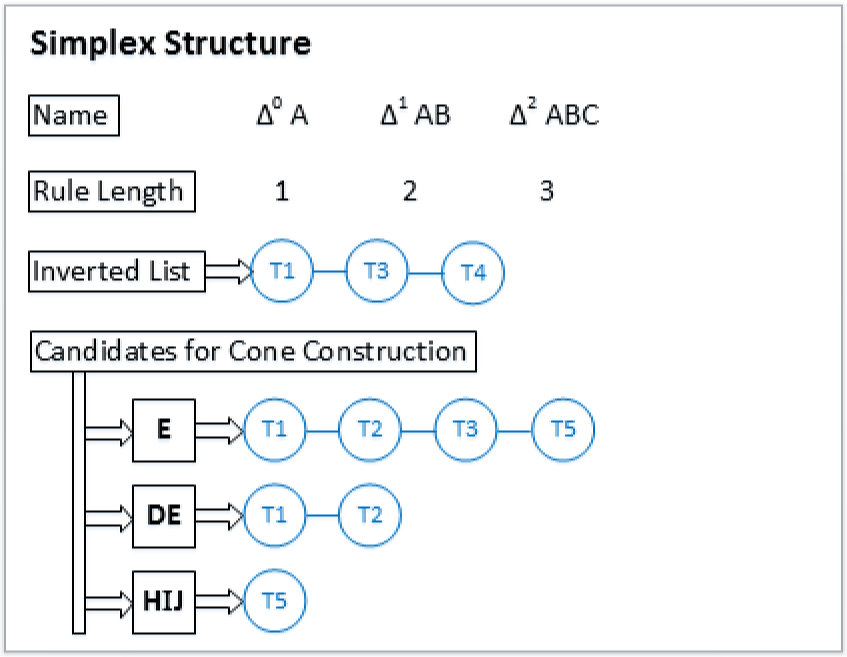


Figure 2.2.1.4: Layout structure describing inner functions and dependencies of a simplex.

In simplex structure above, the name of a simplex has multiple representations but carry the same function. It can represent just a single vertex or a unified group of vertices depending on what phase of cone construction that is being processed. It was designed to support up to (n-1)-simplex where n is number of selected features, thus giving it the flexibilities to work well with high-dimensional datasets.

The second component in this structure is the number of association rules for the simplex. This is somewhat a redundant field since we can extract the number of rules from simplex name itself, but for our convenience, we just pre-calculate and populate it using a 16-bit integer.

The next field, inverted list, is a vector of inverted indices that is associated with each simplex. For a simplex of single vertex, it is simply a list of all data rows with 1's in Table 2.2.1.1. As for a simplex of unified group of vertices, it is a subset of data rows that represents all connected links among the vertices. The higher the dimension, the smaller this subset will get. This is an important point to understand since how we order and traverse the vertices really affects the overall performance. We will cover more on this topic in the next section, *Building Simplexes*, especially regarding to the part about ranking of vertices.

The last component contains a list of possible candidates (simplexes) that are connected to the target simplex. Each element inside the list consists of two parts: (1) Simplex name of the candidate and (2) An inverted list of 1's that belongs to the candidate. The purpose of this list of linking candidates is to speed up the process of identifying simplex-pair associations during the vertex-unification step. If the dataset is complicated with lots of extracted features, the application could potentially divide the work into chunks of vertices for parallel processing, and then consolidate the results when all jobs have been completed. This is not possible without the flexible design of this last component that can provide the necessary support to accomplish such feat. It is important to note that this simplex structure does not memorize all previously constructed objects, but only the immediate and relevant ones.

2.2.2 Building Simplexes

In this section we will discuss how the simplex model is constructed, processed, and refined to rapidly identify key relationships and isomorphic patterns within the graph. There are four phases that are required for building simplexes as well as traversing the graph effectively.

- Use dimensionality reduction to reduce the complexity of input datasets
- Rank and order the vertices to minimize time to connect related vertices
- Apply a new technique called *Self-Contained Cone Construction*
- Traverse the graph through each vertex using a recursive function

For each phase, we will cover more about the special functions and behaviors in following subsections. We will also provide step-by-step illustrations and detailed explanations of what makes the algorithm very capable and highly efficient.

2.2.2.1 Dimensionality Reduction

Dimensionality reduction is a process that is often used in machine learning and statistics to reduce the number of random variables without impacting the final result much. By lowering the number of dimensions, it allows us to perform calculations faster and identify patterns more easily. To illustrate how our algorithm eliminates less dependable vertices, e.g. insufficient data, we apply an artificial barrier of 0.4 to the qualifying threshold. This essentially requires a vertex to have a minimum of 2 data points to be selected for further consideration. So the topological space in Figure 2.2.1.1 is now reduced to six qualified vertices: **A**, **B**, **D**, **E**, **F** and **H** (as depicted in Figure 2.2.2.1). Lower number of vertices also makes it simpler for us to go over the important concept of *Topological Cone Construction* in the next several sections.

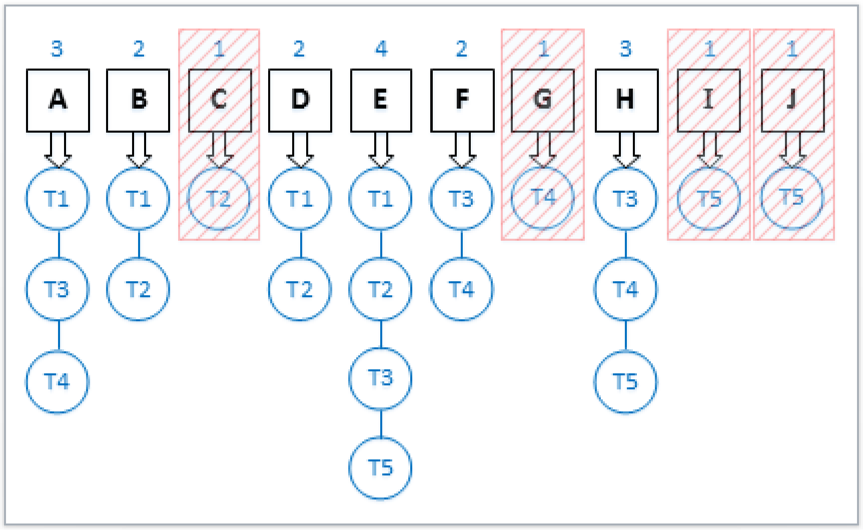


Figure 2.2.2.1: Eliminate vertices with insufficient data by applying artificial threshold of 0.4.

Our Simplicial Complex algorithm defines the qualifying threshold to be a percentage of the maximum # of data rows. In this example, that is (0.4×5) or a minimum of 2 data points. Next we will look at how the vertices are ranked and ordered, and the compelling reason behind such motive.

2.2.2.2 Ranking Vertices

In graph theory, vertex ranking is generally useful for computing vertex-pair similarities. The method that we used for navigating the graph to find connected vertices is a recursive, semi-repetitive procedure. This turns out to be quite significant in terms of performance when deciding what vertex to process next. Thus we need to rank and place the qualified vertices in a certain order.

Let's take a hypothetical example where vertex **A** has an inverted list of {**T2, T5**}, and vertex **B** has an inverted list of {**T1**, **T2**, **T3**, **T4**}. Now imagine we want to find joined edges between them. We can solve this simple problem in two different ways with both yielding the same result; however, by no means they are considered identical. Approach #1: For each indexed row in vertex **A**, we check if it also exists in vertex **B**. The result is an intersected list of {**T2**} after performing two checking operations. Approach #2: For each indexed row in vertex **B**, we check if it also exists in vertex **A**. The result is also an intersected list of {**T2**} after performing four checking operations. From this observation, we can conclude that approach #1 is more effective than approach #2, especially if the number of edges is vastly different between two vertices.

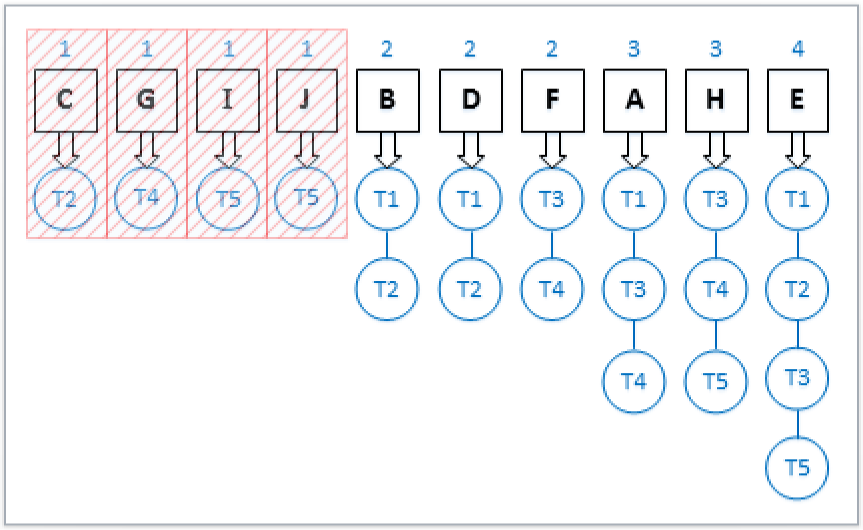


Figure 2.2.2.2: Ranked vertices by number of data points from highest (right) to lowest (left).

Figure 2.2.2.2 shows a reduced table with qualified vertices that are ranked from highest (right) to lowest (left) by the number of data rows and then by the column index. We keep this right-to-left orientation for the convenience of simulating *Self-Contained Cone Construction* in the next section. The order in which the qualified vertices will be visited is strictly defined as follow:

$$\mathbf{E}_0, \mathbf{H}_1, \mathbf{A}_2, \mathbf{F}_3, \mathbf{D}_4, \mathbf{B}_5$$

Even though the algorithm does not traverse from one neighbor to the next, it will eventually have to arrive at every stop listed above. It must do so in order to identify all vertex-pair relationships on the graph. The key point to take away the *Cone Construction* mechanism is that the algorithm should make exactly one stop at each of the vertices, run computations, and then move on and never return.

2.2.3 Graph Traversal

Graph traversal is a process of visiting, checking, and updating each node or vertex on a particular graph. This process is commonly known as graph search in Computer Science. Graph traversal allows some vertices to be visited more than once. That means redundancy is acceptable but an algorithm must do so with extra care. As the graph becomes denser, redundancy can immensely increase the computation time and thus rendering it unusable. So sometimes it is necessary to remember previously visited vertices while other times it becomes impractical due to system limitations. When it comes to graph traversal, it is often not possible to develop a highly efficient algorithm without proper design and support of robust data structures underneath. The opposite can also be true. A lousy algorithm design can also ruin the best data structure out there. What we are basically saying is that algorithm and data structure must always go hand in hand.

2.2.3.1 Topological Cone Construction

The idea of a cone construction can be described as building an object on top of what has already been constructed. It is frequently used in combinatorial topology to connect barycenters of the faces in a specific way. In geometry, the barycentric subdivision is a standard way of dividing an arbitrary convex polytope into simplexes with same dimension [1]. Simply put, we can construct a tetrahedron by adding a point to an existing triangle and creating three edges connecting new vertex to three vertices on the triangle.

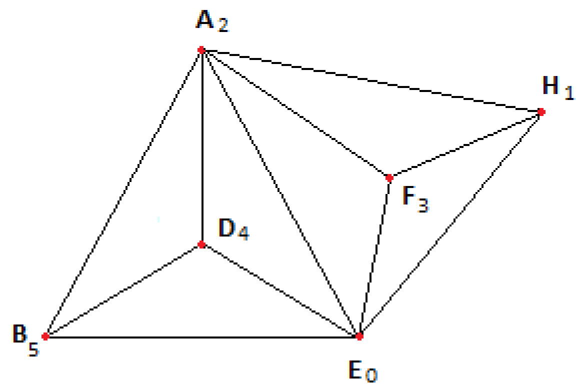
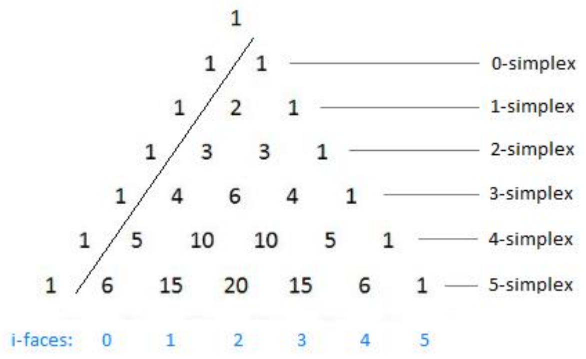


Figure 2.2.3.1a: Visiting order for vertices on a reduced graph in *Cone Construction*.

The above topological graph is a reduced version of Figure 2.2.2.2 with the intent to help speed up the algorithm and maintain its truthfulness. All it does is to filter out unnecessary noise from the initial input data. As a result, we have a robust representation of the graph where each vertex is weighted depending on its content. Note that the original version of *Cone Construction* does not care much about what is actually inside of each vertex and so it treats all vertices as equal. There is no specific or preferred order of traversal. One possible path to navigate the graph could just be in alphabetical order. That is to say: **A**, **B**, **D**, **E**, **F**, and then **H**. However, as explained in one of the earlier sections, this could not have been more unfavorable when searching and identifying simplex-pair associations. The present of the data points in each vertex (or a column of data) is used as part of the calculation to decide how significant the vertex is with respect to the rest of vertices on the graph. Now let us take a closer look to observe what *Cone Construction* is all about and how we can leverage its key concepts to design a more efficient algorithm to deal with large multidimensional datasets.

Before we dive into the simulation, there are a few points that are worth mentioning first. The purpose of a *Cone Construction* is to visit each vertex only once and never look back. All computations that were completed by the preceding vertices will be remembered. As new vertex arrives, it will rely on the prior calculated information in order to construct new simplexes. In essence, the construction of each stage is being built on top of previous stages, hence the name *Cone Construction*. That is to say Δ^1 can only be constructed if the information for Δ^0 is available, Δ^2 if both Δ^1 and Δ^0 are available, and so on.

Below is the simulation for Cone Construction of six selected vertices in Figure 2.2.3.1a. We can follow the color-coded faces to get an idea of how different stages (colors) use the information from prior stages to construct new simplexes with higher dimensions.



0-simplex (Δ^0) has a total of **6** 0-faces:
E_0, H_1, A_2, F_3, D_4, B_5

1-simplex (Δ^1) has a total of **15** 1-faces:
EH_1,
EA_2, HA_2,
EF_3, HF_3, AF_3,
ED_4, HD_4, AD_4, FD_4,
EB_5, HB_5, AB_5, FB_5, DB_5

2-simplex (Δ^2) has a total of **20** 2-faces:
EHA_2,
EHF_3, EAF_3, HAF_3,
EHD_4, EAD_4, HAD_4, EFD_4, HFD_4, AFD_4,
EHB_5, EAB_5, HAB_5, EFB_5, HFB_5, AFB_5, EDB_5, HDB_5, ADB_5, FDB_5

3-simplex (Δ^3) has a total of **15** 3-faces:
$EHAF_3$,
$EHAD_4$, $EHFD_4$, $EAFD_4$, $HAFD_4$,
$EHAB_5$, $EHFB_5$, $EAFB_5$, $HAFB_5$, $EHDB_5$, $EADB_5$, $HADB_5$, $EFDB_5$, $HFDB_5$, $AFDB_5$

4-simplex (Δ^4) has a total of **6** 4-faces:
$EHAFD_4$,
$EHAFB_5$, $EHADB_5$, $EHFDB_5$, $EAFDB_5$, $HAFDB_5$

5-simplex (Δ^5) has a total of **1** 5-faces:
$EHAFDB_5$

Note that the total number of i-faces elements for each simplex up to Δ^5 (or 6 vertices) coincides with 6th row on Pascal's triangle that excludes the left diagonal. We can also observe from Pascal's triangle that adding each dimension to the system increases the total number of faces by a large portion. So for very high dimensional datasets, the amount of resources that are required to keep all previously constructed stages will be unthinkable. This is a major flaw in this *Cone Construction* process. It is just not a practical solution to use for real-world applications. To better gauge the impact of this problem and how we can address it, let's examine it more closely.

The following formula can be applied to compute the number of i-faces elements for k-simplex (Δ^k).

$$i\text{-faces } elements = 2^{k+1} - 1 \text{ for } \Delta^k$$

As you can imagine, the number of i-faces elements can grow exponentially as the number of vertices or dimensions increases (Figure 2.2.3.1b).

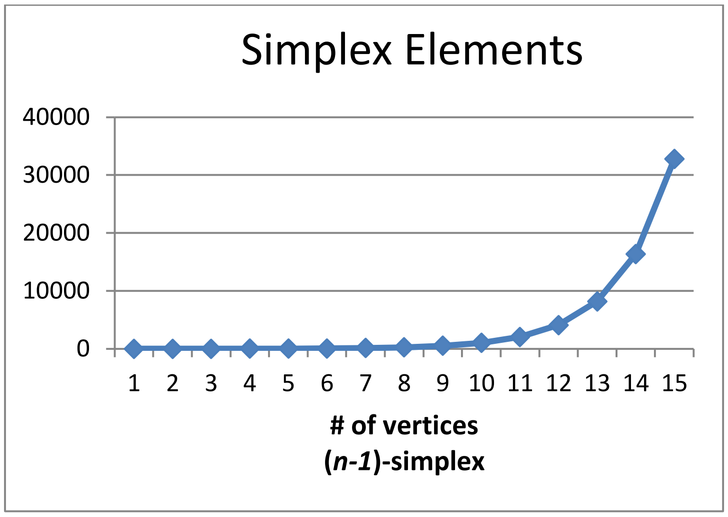


Figure 2.2.3.1b: Chart depicting exponential grow for # of i-faces elements.

The graph above is basically saying that the analysis of the dataset can become very complicated, very fast. This is a vital point in terms of the amount memory required to run and the effects it has on the overall performance. This also brings us back to the significance of Simplex data structure that we have designed for this *Simplicial Complex* algorithm (Figure 2.2.1.4). The aforementioned challenges are the driving causes behind our efforts to develop a novel method called *Self-Contained Cone Construction*, which leads us to the next section.

2.2.3.2 Self-Contained Cone Construction

It has been proven in the earlier section that amount of memory can growth exponentially if we were to design our system based on traditional cone-construction method. For a much smaller dataset, this old method works quite well because of the limited memory that is required to run. However it is not very practical to handle real-world problems because of its inability to scale effectively. We must look for a more appropriate solution to address this challenging problem. Our research work has led us to an improved technique called *Self-Contained Cone Construction*. The key concept behind this particular technique is the ability to process each vertex independently without having any prior knowledge of computed vertex-pair relationships. So within context of the processing vertex, connected edges of previously visited vertices must be re-examined to determine if their associations are still viable. How can this redundant behavior in an algorithm actually improve the overall performance? Inefficient use of physical resources can lead to slower performance of an application, even more so if the system relies heavily on virtual memory (disk) to complete the work. *Self-Contained Cone Construction* allows the algorithm to run with just small amount of physical memory and eliminates the need of virtual memory, thus giving us a huge boost in performance.

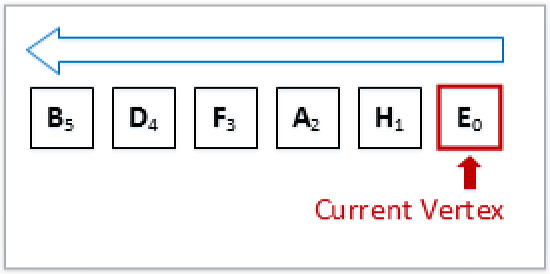
Self-Contained Cone Construction has four key requirements:

1. Current vertex must have less edges than preceding vertices
2. Must compute all simplex-pair associations with preceding vertices
3. Vertex traversal must follow from right-to-left order
4. Vertex unifications must merge rightward until none remains

Now let's go over the graphical simulation of this technique so we can get a better understanding of what is happening at each critical step in the process. We will continue to use Figure 2.2.2.2 and Figure 2.2.3.1a as example to demonstrate our point.

Simulation Step #1:

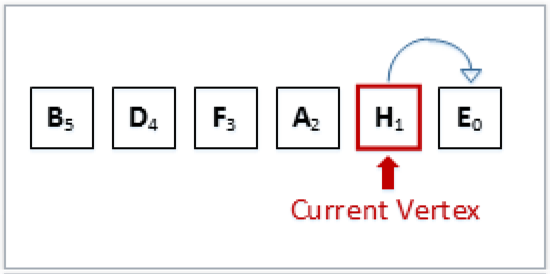
We start out at vertex $\mathbf{E_0}$ since it has the highest number of edges. This will ensure that rules #1 and #3 will not be violated. Since this is the first vertex, we only need to compute all associations up to 0-simplex or Δ^0. For Δ^0 $\mathbf{E_0}$, this is {T1, T2, T3, T5} and it satisfies rule #2. Condition #4 does not apply for Δ^0 so we are done with vertex $\mathbf{E_0}$.

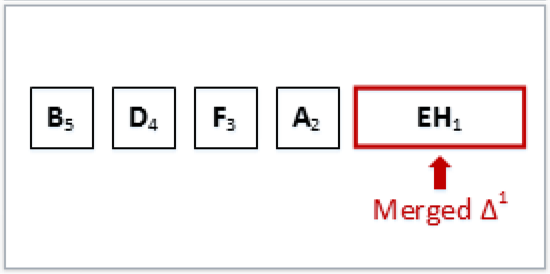


Simulation Step #2:

Now that we completed the processing of *Self-Contained Cone Construction* for vertex $\mathbf{E_0}$, and before we move to the second vertex $\mathbf{H_1}$, there are a couple of important points to remember:

a. All simplex-pair relationships that were computed from the preceding vertices, e.g. vertex $\mathbf{E_0}$ in simulation step #1, are gone.
b. Only edges from preceding vertices that are shared with the current simplex or subsequent unified simplexes will be selected for further considerations.
c. Starting from the current vertex, the unification of simplexes can only be combined in a rightward direction with respect to its current position until there is none left to merge.





With the above notes out of the way, let's move onto vertex $\mathbf{H_1}$, which contains the second-highest number of edges. This is the second vertex in the ordered-traversal list so we need to compute all associations up to 1-simplex or Δ^1. The 1-way associations of $\mathbf{H_1}$, or Δ^0 $\mathbf{H_1}$, are {T3, T4, T5}. Next we will attempt to unify $\mathbf{E_0}$ and $\mathbf{H_1}$ by intersecting these two vertices,

$$\Delta^0\ \mathbf{E_0} \cap \Delta^0\ \mathbf{H_1} = \Delta^1\ \mathbf{EH_1} \text{ or } \{T3, T5\}$$

By connecting vertex $\mathbf{E_0}$ and vertex $\mathbf{H_1}$, we can identify two common edges {T3, T5} that are shared between them. These edges or lines are 1-dimensional objects so the resulting merged vertex is a unified 1-simplex (Δ^1), thus we can label them as Δ^1 $\mathbf{EH_1}$. With Δ^1 $\mathbf{EH_1}$ being in the rightmost position, condition (c) has been activated and we are officially done with self-contained processing of vertex $\mathbf{H_1}$.

Simulation Step #3:

In this step we can observe that the traversing order must be strictly enforced to ensure that all simplex-pair associations are thoroughly covered as well as operations and computations are optimal. Keep in mind that the number of i-faces elements for k-simplex can still be computed using this formula below:

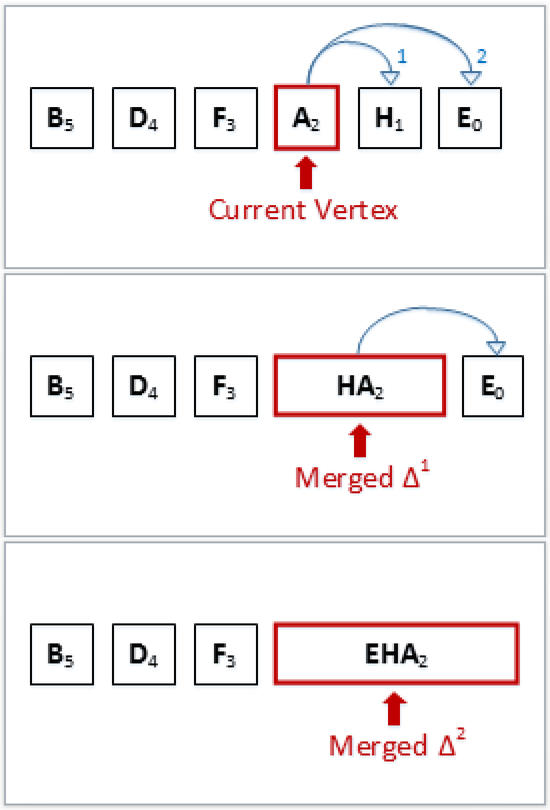
$$i\text{-faces } elements = 2^{k+1} - 1 \text{ for } \Delta^k$$

Let's break this step down into multiple iterations and inspect each of iterations more closely to see what is going on.

Iteration #1:
Δ^1 **HA**$_2$ = {**T3**, **T4**}
Δ^1 **EA**$_2$ = {**T1**, **T3**}

Iteration #2:
Δ^2 **EHA**$_2$ = {**T3**}

Iteration #3:
None left to merge (done)



Notice that the four requirements that were listed at beginning of the simulation must be applied to each of the three iterations. This also holds true for the conditions (a), (b) and (c) that were mentioned in Simulation Step #2. By now we can observe that at outer-most layer of this algorithm, the current "processing" vertex is moving from right-to-left direction where number of edges is in a decreasing manner. In Section 2.2.2.2, *Ranking Vertices*, we have shown that identifying simplex-pair associations can greatly improve the performance if the edges are connected from low-to-high count. This is indicated by (1) and (2) curved arrows in Iteration #1 diagram above.

Simulation Steps #4, #5 and #6:

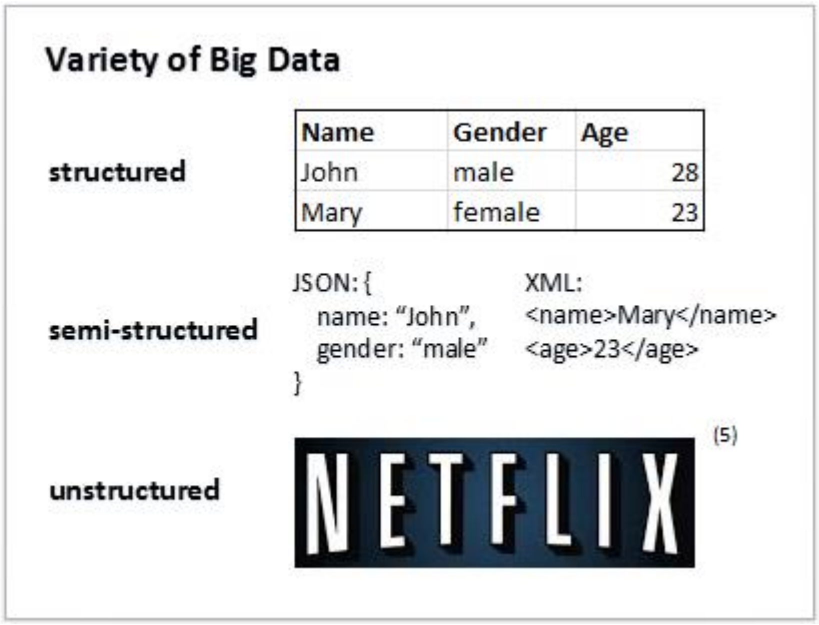
As the number of vertices increases, there will be more iteration steps that will be required in order to fully complete the simulation step. It will be very difficult to describe them in such a way that is not convoluted. But in case you're interested of seeing more, an illustration of how to complete the processing of vertex **F**$_3$ using *Self-Contained Cone Construction* can be found in Simulation Step #4 of Appendix Section (ii), *Simulations*, (1) *Simulation Step #4 of Self-Contained Cone Construction*. It is essentially an extension of what was presented in Step #2 and Step #3 but with higher dimensional simplexes and more associative connections. It should provide more clarity for the algorithm.

As for steps #5 (**D**$_4$) and #6 (**B**$_5$), the same procedure can be repeated as long as the rules of engagement and stated conditions remain intact. Not only that, we can also expand this proposed concept to much higher dimensional datasets and the underlining principles still work effortlessly. This is one of the true benefits of this algorithm design in that it contributes great efficiency, flexibility and scalability. And in the world of Big Data, this is a big plus.

3. Parallel Computing

3.1 Big Data

These days Big Data and parallel computing are synonymous. For a very simple reason, we cannot talk about one topic without directly referring to the other. Big Data is data whose scale, diversity, and complexity require new architectures, techniques, and algorithms to manage and extract the hidden knowledge from it [11]. Analyzing data has become tantalizingly difficult due to tremendous growth in volume, velocity, and variety of the data. We are no longer dealing with gigabytes and terabytes of data, but more like with petabytes (10^{15}) and exabytes (10^{18}). And in rare instances, we are approaching the zettabytes (10^{21}) territory. A research from University of Southern California reported that in 2007, humankind successfully sent 1.9 zettabytes of information through broadcasting networks and GPS devices. It has been estimated that the Internet traffic alone could reach 1.3 zettabytes by 2016 (2). Not only volume and velocity of data are problematic, the data can come in various forms. It can be in structured form, which is what we were traditionally familiar with. Another form that it can take shape is semi-structured. This is what we are trying to adapt and use in many of our current technologies. JSON and XML formats are perfect examples of what semi-structured data would be like. The third type, unstructured, carries the most weight since it does not have to be in any particular form. Video streaming, weather forecast, and universe observations can generate data in such unstructured form.



Variety of Big Data

structured

Name	Gender	Age
John	male	28
Mary	female	23

semi-structured

```
JSON: {
  name: "John",
  gender: "male"
}
```

```
XML:
<name>Mary</name>
<age>23</age>
```

unstructured

(5)

In this section, we will discuss how to take the important concepts that we absorbed from Simplicial Complex and apply them to Parallel Computing environment. We will also describe the current methodology in MapReduce paradigm and how it can be trivially integrated with *Simplicial Complex*.

3.2 Embarrassingly Parallel

In parallel computing, embarrassingly parallel is a technique that requires minimal efforts to break the problem up into smaller chunks and run them in a parallel manner. The computation can be divided into a number of independent parts, each of which can be executed by a separate process. Each separate process is an exact clone of the other and is strictly assigned to work on a specific portion of the original dataset. This parallel technique is also known as *Data Partitioning*. The idea is to have as many of these subtasks running simultaneously as possible. This model actually fits and works quite well with Simplicial Complex algorithm as we will see later on.

Here is a graphical illustration of what an embarrassingly parallel problem consists of. See Figure 3.2 below for more details.

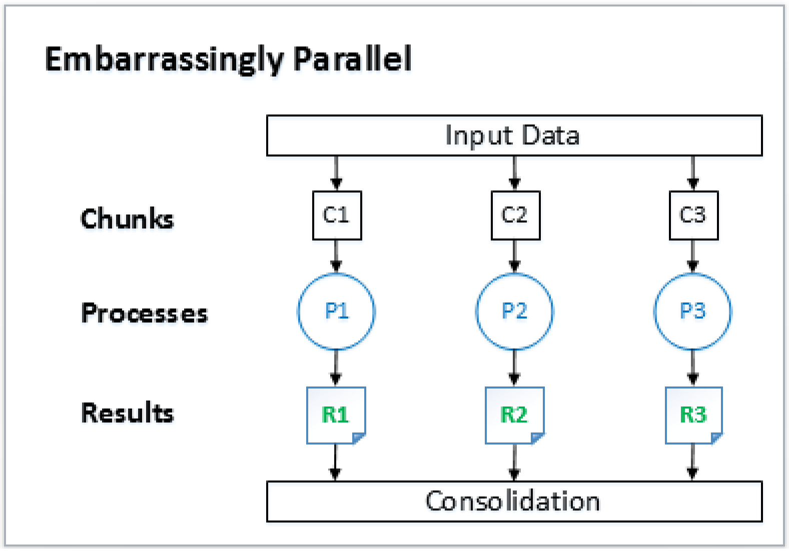


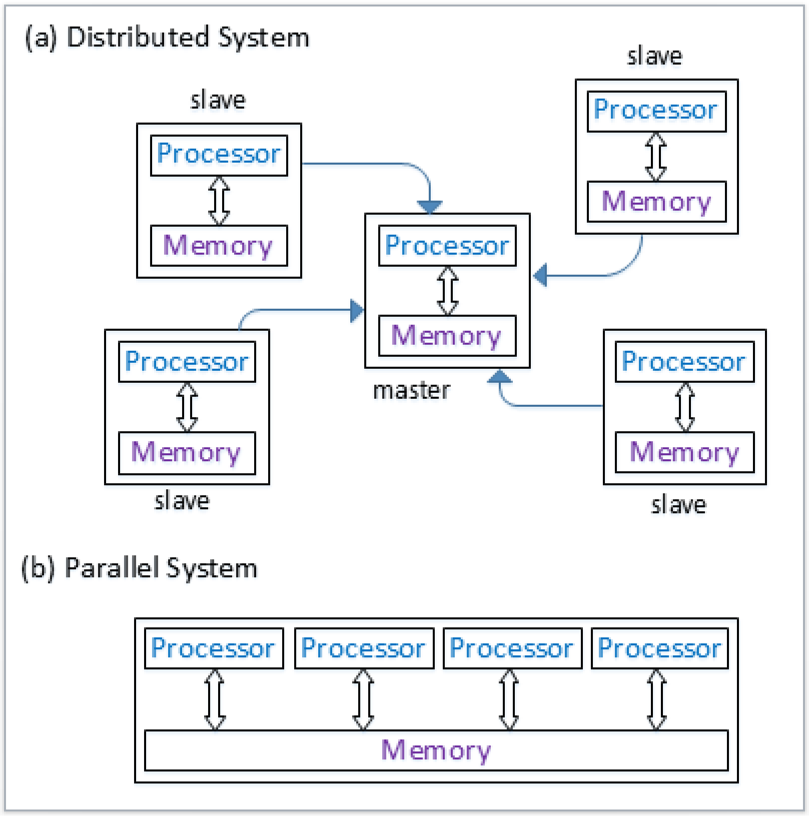
Figure 3.2: Embarrassingly parallel tasks that are divided up to run with one piece of data.

One convenience about this technique is that the subtasks can run with various speeds. Each subtask is not bound to complete within a certain timeframe. Although it should not take an indefinite amount of time to run, or it will be at risk of being excluded from consolidation step due to timeout. Let's take an example of a program that counts word occurrences in a given book. One way to approach this problem is to have a single program that can scan the entire book and maintain a hash table to track the number of occurrences for each word encountered. This may seem like an easy fix but let's stretch our imagination a little bit. Let us assume that this is no ordinary book but a colossal one. It is so big in size that even a supercomputer would have trouble tracking all generated results, not to mention, it would be very slow to run. Well let's say we can divide the book into chapters, or even pages, and then assign each chapter or page to the same program that can run on a separate computer system. Each computer system would then generate a small map table that contains the counts of all word occurrences within that chapter or page. Once you have lots of these page results, we just need to write a merge tool that can consolidate all of them into one final outcome. This is what we commonly refer to as *Distributed Computing*, which we will cover more in the next section. In addition to that, we will also go over Hadoop ecosystem and MapReduce programming model, originally designed by Google, and how *Self-Contained Cone Construction* will behave in such platforms. It is important to note that not all Hadoop challenges will be addressed in this paper, especially the section regarding to dimensionality reduction in *Simplicial Complex*.

3.3 Distributed Systems

Distributed system is a model in which components of a software system are shared among multiple computers to improve the efficiency and performance of the overall system. These nodes inside a computer network communicate and coordinate with each other using a mechanism called *Message Passing Interface* or MPI. Many distributed systems are designed to solve complex computational problems. One of the key characteristics of a distributed system is the independent failure of components [1]. The failure of one or more nodes does not hinder progress of the remaining nodes. Referring back to our word-count example, if one page was ripped out and its assigned node was unable to process it, the overall system would still continue to work and behave normal, albeit the result from missing page will not be accounted for in the final outcome.

The inner working of a distributed system (a) is different from that of a parallel system (b) as we can see in the figure to the right. The major distinction between the two systems is whether the memory space is shared among the processors. Parallel computing is actually a tightly-coupled form of a distributed computing. And on the other hand, distributed computing can be thought as a loosely-coupled form of parallel computing. For the purpose of this paper on Simplicial Complex, we will only focus in the architecture of a distributed system to process high dimensional, high complexity datasets using *Self-Contained Cone Construction* technique. Simplicial Complex.



3.3.1 Hadoop Framework

Today one of the popular distributed computing platforms is the Hadoop framework, which consists of Hadoop core libraries, MapReduce and Hadoop Distributed File System (HDFS). For the past decade, Hadoop has evolved very rapidly. When starting out a new Big Data project, you can always build your own Hadoop system; however, a better approach would be to leverage one of the existing Hadoop distributions that are currently available on the market. It will help speed up the research and development work.

Here is a list of Hadoop distributions (1):

- **Apache Hadoop** – Open-source distribution maintained by Apache Software Foundation.
- **Map-R Technologies** – Hadoop distribution with some unique features such as NFS mounting.
- **Cloudera** – Leading distributor of Hadoop, built on top of Apache Hadoop.
- **Horton Works** – Also built on top of Apache Hadoop with enterprise support.
- **AWS Elastic MapReduce** – Hadoop distribution by Amazon Web Services

There is also an entire ecosystem that resolves around Hadoop technologies. There are literally countless projects and tools that are available to software developers and data scientists to make use of when working with Hadoop and MapReduce. Many of these tools were designed to tailor to specific needs of individual projects or systems. We will briefly cover what these tools are and how they are used in different scenarios. It is important to have a high-level understanding of what is being provided by these projects; however, we will not go into the details of each one.

Related projects and tools in Hadoop ecosystem (4):

- **Avro**: Data serialization framework that allows arbitrary schemas to be defined.
- **Pig**: Analyzing large datasets using high-level language called Pig Latin.
- **Hive**: Storing and retrieve Hadoop data using high-level SQL called HiveSQL.
- **HBase**: Column-oriented, scalable distributed data storage based on Hadoop.
- **Mahout**: Scalable machine learning (ML) library built for Hadoop.
- **YARN**: Next-generation of MapReduce, capable of running over very large clusters.
- **Ozzie**: Workflow scheduler system to create and manage MapReduce jobs.
- **Flume**: Reliable service for collecting and aggregating log data in HDFS.
- **Sqoop**: Simplify the transferring of data between Hadoop and relational databases.
- **Cascading**: Framework for building Hadoop applications.
- **Spark**: Pipeline for large-scale data processing, supports multiple programming languages.

Now that we've slightly covered some of these topics for Hadoop framework, let us shift our focus to MapReduce and the integration of *Simplicial Complex* algorithm.

3.3.2 MapReduce Paradigm

MapReduce is a distributed computational model that was introduced by Google in 2004. It is a programming model for processing and generating large datasets using a distributed algorithm running on a cluster of computers. The idea is to exploit the locality of reference by bringing the compute module to where the data resides and not vice versa. Figure 3.3.2a depicts the transitioning from traditional method of using relational databases to a more robust model of MapReduce.

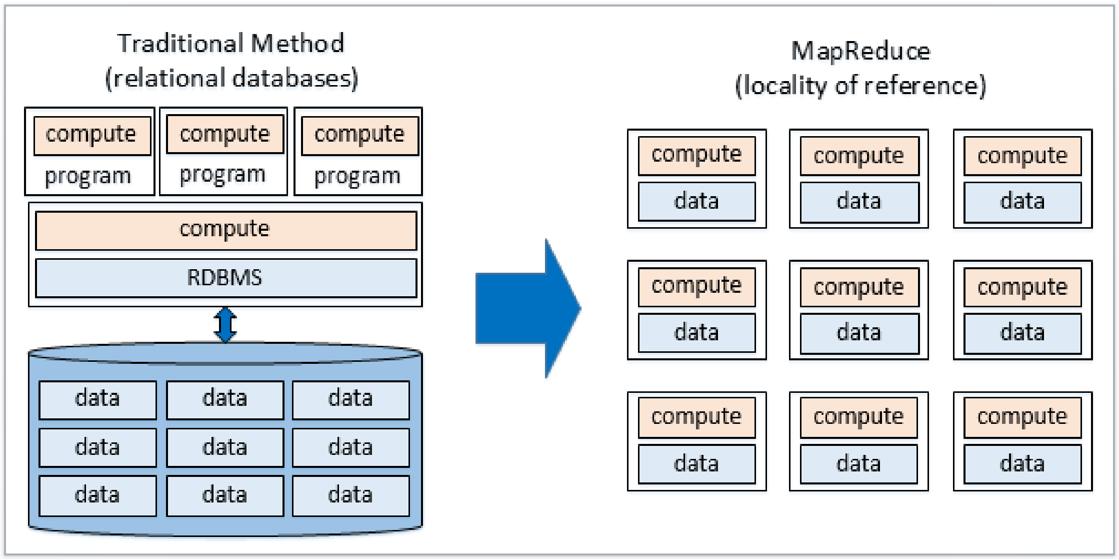


Figure 3.3.2a: Moving away from traditional method (RDBMS) to MapReduce paradigm.

In the past, we normally would have to import the data to wherever the computer system is located. We can copy or import it to a database server that is residing on the local area network (LAN) before attempting to run the computations on local computer. This was quite a common practice in the old days because the size of data was somewhat manageable. It has become increasing difficult to keep up with this antiquated process as the data grew rapidly over the years. To eliminate this headache, MapReduce paradigm suggests that the compute module should be copied or transported to where the data actually resides. This is the very definition of *Locality of Reference*. This is the single-most important aspect of MapReduce that allows it to scale the data processing with ease.

The three main problems that MapReduce paradigm addresses:

1. **Parallelization** –divide a complex problem into subtasks and run them in parallel.
2. **Distribution** – how to distribute the data across the cluster.
3. **Fault-tolerance** –handle failure of components gracefully.

MapReduce programming model consists of two key phases. The first is called the mapping phase. The purpose of this stage is to process and filter the input data. This could be in form of a file or a directory of files that is stored in HDFS. The input file is passed to the mapping function line by line. If the input data is enormous, it can be split into multiple pieces and pass along to multiple mapper jobs. Once completed, the mapper will generate small chunks of data that is usually group by their keys. That brings us to our second phase, the reducer. This stage is a combination of the shuffle stage and the reduce stage. The purpose of this stage is to process the data that are generated by the mapper jobs. In our word-count example, it would simply aggregate the total count for each unique word. So one reducer job could potentially generate total counts for words starting with letter “A”, and another mapper jobs would generate the results for words starting with “B”, and so on. Depending on each project’s needs, the number of mappers and reducers can vary. Figure 3.3.2b is a typical MapReduce workflow.

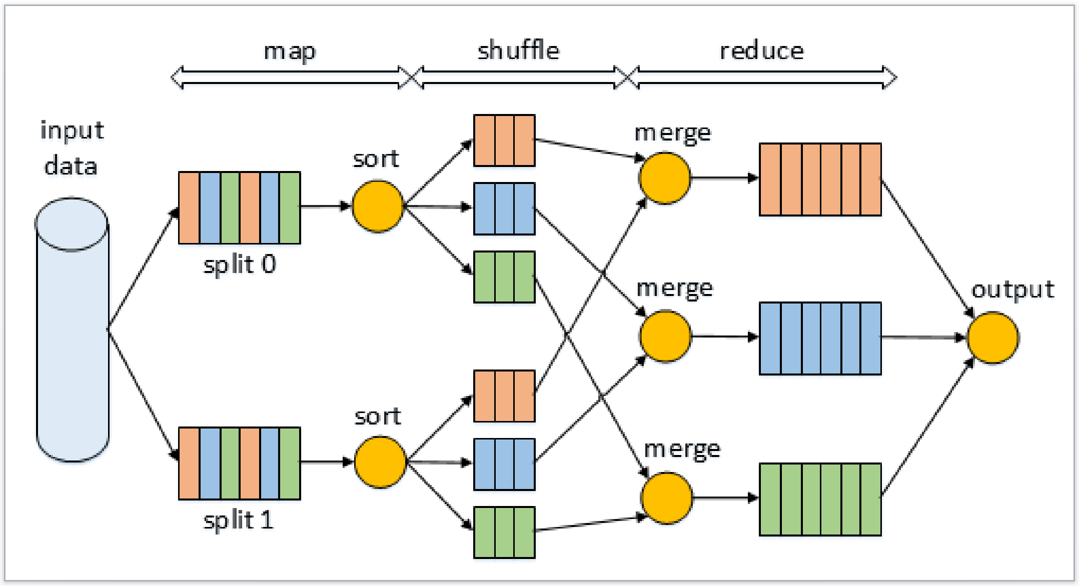


Figure 3.3.2b: MapReduce workflow that describes the job functions at different stages.

The major advantage of MapReduce programming model is its scalability over multiple large number of computing nodes. Another advantage is that much of the hard work has already been handled by the Hadoop framework. This can mostly be done by changing the settings in configuration file. Developers only have to worry about implementing the functions for mapper and reducer jobs, and let the framework magically take care of the scaling problem.

3.4 Simplicial Complex on Hadoop

3.4.1 Initial Concept

In this section, we would like to present the initial concept of putting Simplicial Complex on Hadoop platform and leveraging the MapReduce paradigm. The initial approach of *Simplicial Complex* algorithm was designed to run efficiently on a single system, thus it was not designed for scalability. The components inside the system were tightly coupled. This was a major bottleneck. With the introduction of *Self-Contained Cone Construction* technique, we can now safely and reliably scale the system to run with enormous datasets by taking advantage of the proven principles behind MapReduce model. Figure 3.4.1 describes the general idea of our proposed workflow for putting Simplicial Complex on Hadoop cluster and how the components interact with each other. One can imagine that additional mapper and reducer nodes can readily be inserted into the cluster to increase the overall throughput. In the subsequent sections, we will go over the plan to scale horizontally, split the input data, implement mapper and reducer jobs, and complete with final consolidation step.

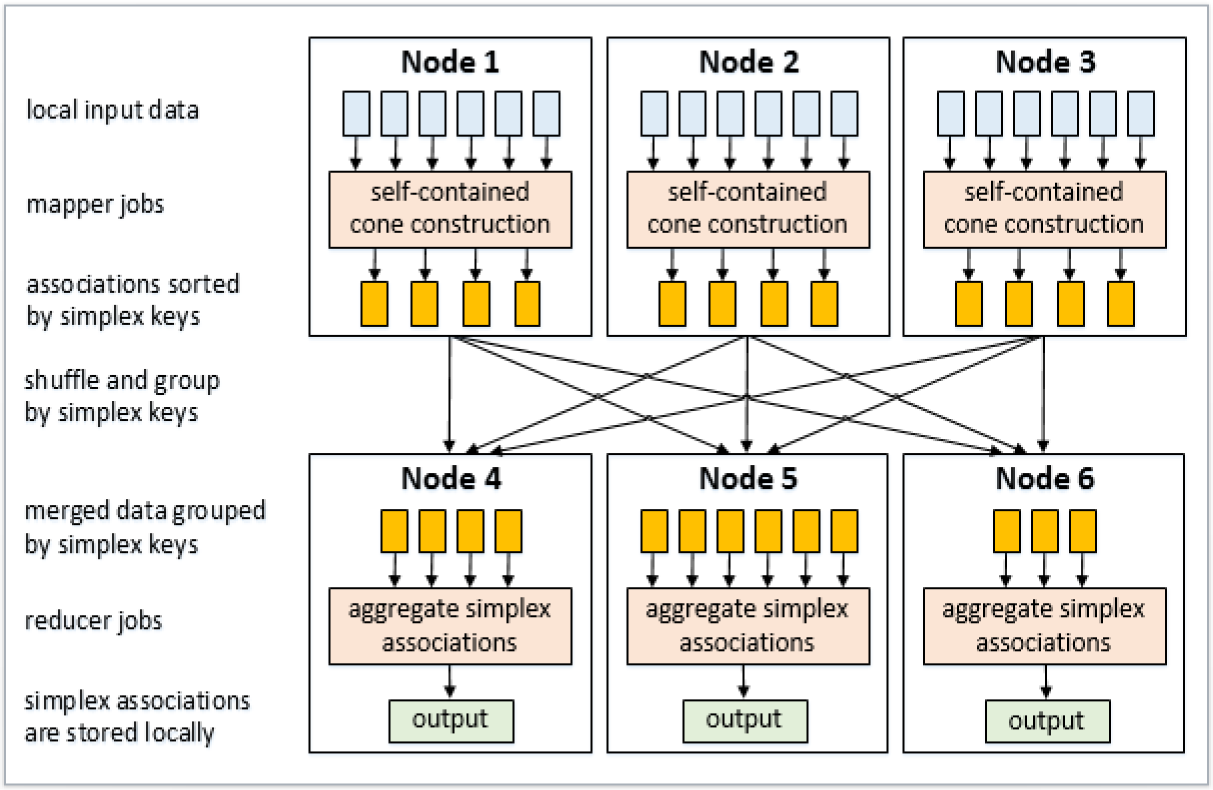

local input data
mapper jobs
associations sorted by simplex keys
shuffle and group by simplex keys
merged data grouped by simplex keys
reducer jobs
simplex associations are stored locally
Node 1
Node 2
Node 3
self-contained cone construction
self-contained cone construction
self-contained cone construction
Node 4
Node 5
Node 6
aggregate simplex associations
aggregate simplex associations
aggregate simplex associations
output
output
output

Figure 3.4.1: Running Simplicial Complex algorithm with MapReduce model on Hadoop cluster.

Note that the importance of locality of reference is being applied successfully in the proposed model above. The compute module for *Self-Contained Cone Construction* technique has been moved to the location of the partitioned data.

3.4.2 Horizontal Scaling

As shown previously, *Simplicial Complex* algorithm requires a lot of computational power when working with high-complexity, high-dimensional datasets. Scaling vertically, which is adding more CPUs and memory to a single computer, is simpler but not practical. A more ideal solution would to be to architect and design a distributed system that can vastly increase the computing power by adding more computers to an existing cluster. In short, what we want is horizontal scaling. We can achieve this goal by simply integrating our *Simplicial Complex* algorithm with Hadoop framework. As a result, we can build a *Simplicial Complex* framework that is reliable, scalable, and high-performance.

3.4.3 Input Split

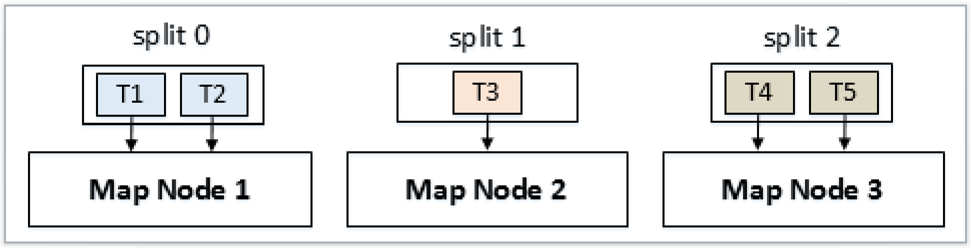
Hadoop is built on top of HDFS, which was setup to break down very large files into physical blocks and store multiple copies on different nodes in the cluster. A typical MapReduce task must process all records within a given block. Given that the size of a physical block is fixed, it is problematic when handling records that span the block boundaries. Hadoop solves this problem by introducing a logical representation of the stored data called *Input Splits*. In order to properly illustrate how this step works in our algorithm, we will refer back to our earlier example that was presented in Section 2.2.3.1, *Topological Cone Construction*, which consists of five tuples covering six qualified vertices. The six qualified vertices are: **A**, **B**, **D**, **E**, **F** and **H**.

split 0 split 1 split 2

	A	B	D	E	F	H
T1	1	1	1	1	0	0
T2	0	1	1	1	0	0
T3	1	0	0	1	1	1
T4	1	0	0	0	1	1
T5	0	0	0	1	0	1

Table 3.4.3: Reduced version of Table 2.2.1.1 with splits.

Let us further assume that the Hadoop framework somehow break our input Table 3.4.3 into three splits. Split 0 has two tuples {**T1**, **T2**}, split 1 has only one tuple {**T3**}, and finally split 2 has two tuples {**T4**, **T5**}. This will simplify the model and help us better explain the concept. In general, the number of splits should help determine the number of mapper nodes for an application. Each input split will get replicated several times to improve reliability and availability. Hadoop will then use the locality principle to assign each input split to a mapper node for processing. The diagram below shows a simple cluster with three mapping nodes that we will continue to use for the remainder of this section.



3.4.4 Map Function

The job of the map function in MapReduce model is to take a series of key-value pairs from input split, perform necessary operations on them, and then generate one more or key-value pairs as output. The data types for both inputs and outputs can be different from one another. The purpose of this map stage is to organize and prepare the data for the reduce stage. For example in *Simplicial Complex* example, the key is the tuple row, e.g. {**T1**, **T2**, **T3**, **T4**, or **T5**}, and the value is an array of data points that connect the edges to all available vertices. Another possible approach would be to assign the key to the split number and the value to a two-dimensional array where the rows are tuples within the split and the columns are vertices. Hadoop framework does have a size limit on each input split, e.g. 64 MB or 128 MB depending on the distribution. Because the restriction in split size is limited, our Self-Contained Cone Construction algorithm would fit nicely in this model.

To continue working with our simplified model that was described in Section 3.4.3, *Input Split*, we can now generate some output examples of key-value pairs for each of the three mapper jobs. See Table 3.4.4a to the right for more details. The sample outputs of key-value pairs that came out of this stage are sorted by the keys, which in this case are simplexes. We can view this sorting step in our MapReduce workflow as depicted earlier in Figure 3.3.2b. It was labelled as "sort". We need this step to accommodate the fact that *Simplicial Complex* algorithm traverses the topological graph in some suggested paths due to ranking of the vertices. That means the simplex-pair associations can be written out in an arbitrary order as long as the computations can be completed efficiently. Looking more closely at this output table, we can observe that there are 15 permutations calculated for 4 available vertices in split 1. This is identical to the sum of values in row 4 or 3-simplex of Pascal's triangle. We can also see that all permutations contain count of exactly 1. This is because split 1 has just one tuple {**T3**} that was assigned to map node 1 to process. Lastly, tuples {**T4**, **T5**} have 3 and 2 vertices respectively, and together they were able to calculate associations for 9 possible simplexes, thus leaving the remaining 6 slots as blanks, which were shown as grayed-out cells.

split 0		split 1		split 2	
Mapper 1		**Mapper 2**		**Mapper 3**	
key	**value**	**key**	**value**	**key**	**value**
A	1	A	1	A	1
AB	1	AE	1	AF	1
ABD	1	AEF	1	AFH	1
ABE	1	AEFH	1	AH	1
ABDE	1	AEH	1	E	1
AD	1	AF	1	EH	1
ADE	1	AFH	1	F	1
AE	1	AH	1	FH	1
B	2	E	1	H	2
BD	2	EF	1		
BDE	2	EFH	1		
BE	2	EH	1		
D	2	F	1		
DE	2	FH	1		
E	2	H	1		

Table 3.4.4a: Ouput key-value pairs from three mappers.

In MapReduce programming model, there is an intermediate stage that generally does not receive the same recognition as the Map and Reduce stages, but it is equally important. This step is often referred to as the *Shuffle and Sort* phase, or sometimes known as the handoff process. To help speed up the overall MapReduce system, data is immediately moved from mapper nodes to reducer nodes to avoid contentions for network traffic. The shuffle operation is responsible for transferring of data from mappers to reducers. The sorting component helps organize the data, and then distribute it to appropriate reducing tasks and trigger them to start. As a result, this can save a lot of time for the reducer phase. Keep in mind that *Shuffle and Sort* operations are being performed locally, by each reducer, for its own input data. Back to our simple MapReduce model, let's just say we want to designate each reducer job to process all key-value pairs that start with a unique alphabet letter. By following this design assumption, a total of 6 reducers would be recommended for handling this part of workload. Also with this suggested number of reducer nodes, a custom partitioner is required to distribute the results to appropriate reducers so the next phase can be completed efficiently.

3.4.5 Reduce Function

The job of a reducer is to obtain a sorted list of key-value pairs, aggregate or combine the value list into one result, and emit a single key-value pair for each input tuple. The value list should contain all values with the same key that was produced by the mappers. Figure 3.4.5 shows how the sorted lists of key-value pairs that get propagated from the *Shuffle and Sort* phase to the *Reduce* phase for merging and aggregating the association counts for all distinct simplexes. Note that the input blocks are broken into 3 different splits to show us what happens during *Shuffle and Sort* stage. For example, key-value pair of **{A, 1}** appears in all 3 splits but **Reducer 1** actually gets a key-value pair of **{A, [1, 1, 1]}**.

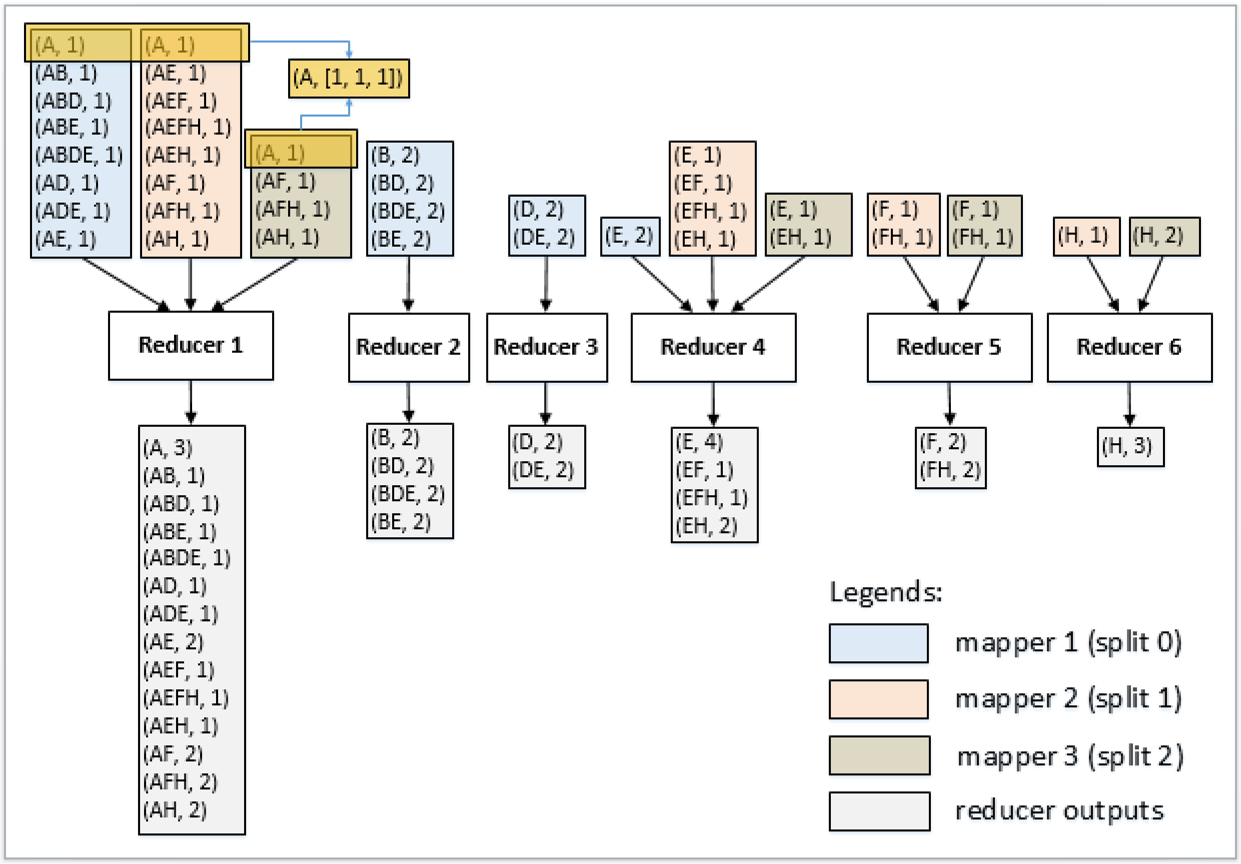


Figure 3.4.5: The sorted list of key-value pairs being assigned to 6 reducer nodes for aggregation.

For a complete list of the generated output by our *Simplicial Complex* program, please refer to Appendix (ii), *Simulation Runs*, (2) *Simplicial Complex Run Results*.

4. Experiment

4.1 Results

Frequent Pattern (FP) Growth:

- Build FP-Tree structure
- Traverse and extract large item sets

Dataset	Time	Memory	Notes
5k rows	1.38 seconds	8 MB	4 dimensions, 0.097 threshold, 77 columns
65k rows	14.6 minutes	100 MB	4 dimensions, 0.042 threshold, 1257 columns
2 rows	44.7 hours	30 MB	28 dimensions, no threshold, 30 columns

Initial version 1 of *Simplicial Complex* algorithm:

- Navigate each column and from right-to-left order
- Connect each vertex with all constructed shapes
- Remember all constructed shapes, e.g. cone construction
- Good performance but very memory intensive

Dataset	Time	Memory	Notes
5k rows	7.75 seconds	233 MB	4 dimensions, 0.097 threshold, 77 columns
65k rows	31.4 seconds	3.5 GB	4 dimensions, 0.042 threshold, 1257 columns
2 rows	Failed	N/A	28 dimensions, no threshold, 30 columns

Improved version 2 of *Simplicial Complex* algorithm:

- Reduce size of graph based on threshold
- Visit and process each vertex only once
- Use recursion to connect points on graph to each vertex
- Shrink inverted table accordingly for each visited vertex

Dataset	Time	Memory	Notes
5k rows	1.21 seconds	4.7 MB	4 dimensions, 0.097 threshold, 77 columns
65k rows	4.73 seconds	43.3 MB	4 dimensions, 0.042 threshold, 1257 columns
2 rows	29.1 minutes	1.5 MB	28 dimensions, no threshold, 30 columns

Advanced version 3 of *Simplicial Complex* algorithm:

- Fix initial sorting of reduced graph
- Consolidate qualified columns and counts
- High performance, low memory usage
- Easy to scale horizontally using data partitioning

Dataset	Time	Memory	Notes
5k rows	0.01 sec	2.3 MB	4 dimensions, 0.097 threshold, 77 columns
65k rows	3.18 seconds	35.2 MB	4 dimensions, 0.042 threshold, 1257 columns
2 rows	5.25 minutes	0.8 MB	28 dimensions, no threshold, 30 columns

4.2 Performance

4.2.1 Comparisons

To compare the performance of new Simplicial Complex algorithm to traditional Frequent Pattern (FP) Growth method, we setup a stress testing scheme that allows us to gauge into the kind of improvements that we have made.

We attempted to run a broad range of features but maintain a consistent set of parameters across the two algorithms. For example, here are some parameters and their correspondent explanations:

- # of association rules – maximum allowable simplex for extracting relationships (**100**)
- Qualification threshold – consideration limit for dimensionality reduction (**0.0**)
- # of features in dataset – number of vertices on graph (**20**, **22**, **24**, **26**, **28**, **30**)
- # of data rows – how many rows of data for stress testing (**2**)

By observing the results that were depicted in Figure 4.2.1a, we can conclude that the performance of our new Simplicial Complex algorithm is much more efficient than that of traditional FP-Growth. By adding a few more dimensions can practically render FP-Growth useless. The main reason is that the construction of FP-tree is very time consuming and memory intensive when processing data with large number of dimensions.

	Time Elapsed (seconds)		
# of Features (n-1) dimensions	**Simplicial Complex**	**Frequent Pattern (FP)-Growth**	**Performance Speedup**
20	0.312	67.8	215
22	1.21	584.5	485
24	5.22	1920.2	370
26	20.6	11752.6	570
28	84.9	39342.5	465
30	315.15	161041.1	510

Figure 4.2.1a: Stress-testing runtime table for Simplicial vs. FP-Growth comparisons.

Input data for 20 features:
18 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Input data for 22 features:
20 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Input data for 24 features:
22 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

Input data for 26 features:
24 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Input data for 28 features:
26 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

Input data for 30 features:
28 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

Even though the stress tests designed above are quite simple, they allow us the opportunity to see (at the deepest level) how potent each algorithm really is. Since we were able to scale *Simplicial Complex* algorithm horizontally using Hadoop distributed computing, we only need to data rows to illustrate our point. In stress testing, we purposely make all vertices active except for last two in data row #1 and first two in data row #2. In reality, the data will be much more sparsely populated, especially for semi-structured and unstructured data. In either case, *Simplicial Complex* on Hadoop framework will handle them gracefully. Notice how Figure 4.2.1b shows a huge spike in slowness of FP-Growth as the number of features increases.

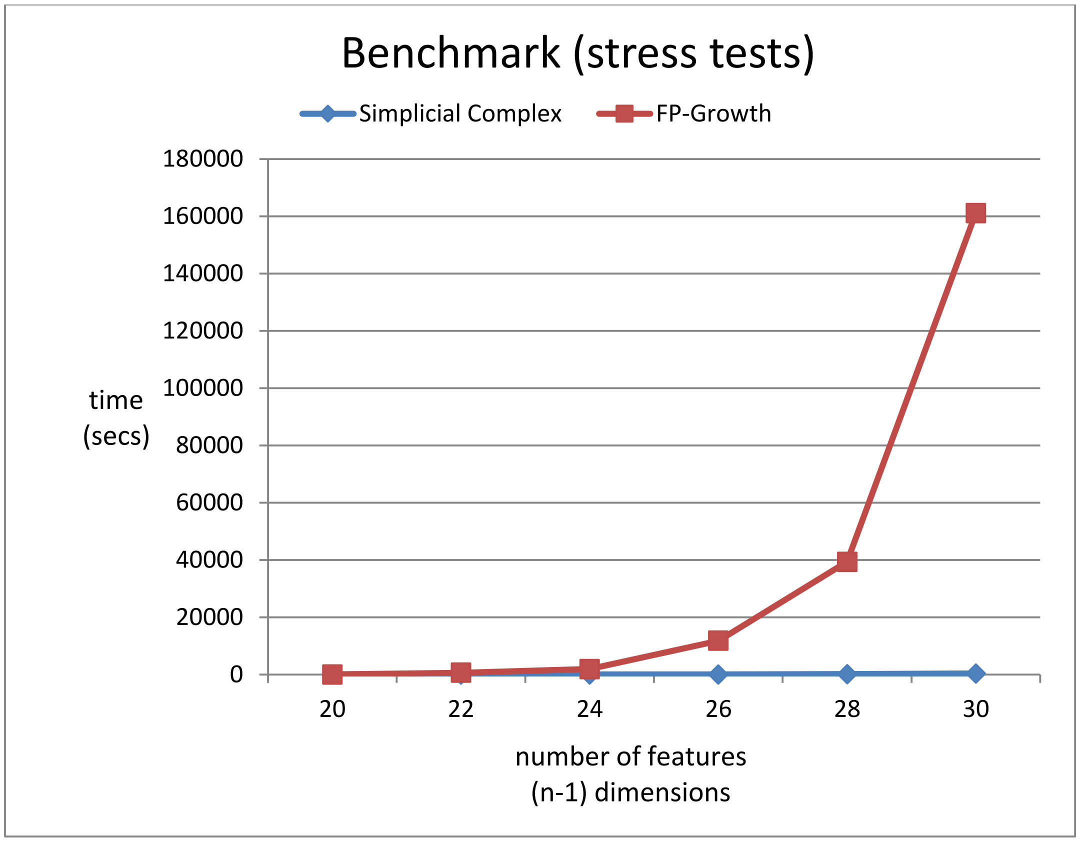


Figure 4.2.1b: Graph chart of performance benchmark comparisons for stress testing in Figure 4.2.1a.

4.2.2 Resource Utilization

This is one of the important aspects of Computer Science that many programmers often take for granted. This is due to that fact that nowadays many high-level languages try to take care of the memory management for them. This is what happened to our first implementation of *Simplicial Complex*. The algorithm was very fast but when we attempted to increase the complexity of our dataset from about 100 features and 1300 features, the application quickly jumped from 200 MB of memory usage to over 3.5 GB. It also took about 30 seconds to run on this controlled dataset. This caused us to reinvestigate our initial algorithm and the underlining data structures that it used. This work eventually led us to our development of *Self-Contained Cone Construction* technique. And the result is quite dramatic. For a 100-feature run, the application used only about 2 MB of memory and ran in merely a fraction of a second. As for 1300 features, it used approximately 35 MB of memory and completed the run in a little over 3 seconds. This is amazing. Not only it allocated a hundred times less memory, but it also executed ten times faster than the initial implementation of *Simplicial Complex*. What we did was we basically killed two birds with a one stone.

5. Conclusion

In this term paper, we have concluded that *Simplicial Complex* algorithm, when being used correctly, can yield a lot of benefits when dealing with large multidimensional datasets. It has also been shown that the *Self-Contained Cone Construction* technique within this algorithm is highly is adaptable to Hadoop ecosystem and MapReduce programming model. Not only that, there are so many Big Data problems that currently exist in Data Science that can leverage this *Simplicial Complex* framework to provide fast and reliable solutions. By publishing our research findings, we hope that the scientific community can make use of it and then add their own contributions.

5.1 Discovery

One of the important findings that we made in this paper is the idea of *Self-Contained Cone Construction* and how it utilizes the system's limited resources in an efficient manner. We discovered that it is not always correct to apply what works in theory, mathematics in our case, to practical applications without considering the designing of proper algorithms and data structures. In addition, we showed that the scalability bottleneck in traditional method is no longer a concern. We also illustrated how our proposed concept works and integrates with distributed computing. This allows us to scale *Simplicial Complex* like never before.

5.2 Future Work

Even though we made good progress for this research, there is still a lot of work to be done and improvements to be made. We developed the algorithm and setup the infrastructure so that one can easily take the existing work and improve upon it. Here is a list of possible future work:

- Performance improvements by applying graph partitioning techniques.
- Dynamically adjust size of input data blocks depending on number of selected features.
- Develop data clustering algorithms based on graph localization and translation.
- Feed results from *Simplicial Complex* framework into *Perceptron* model for deep learning.
- Designing of advanced analysis pipeline that incorporate and interact with this framework.

SAN JOSE STATE UNIVERSITY

MASTER'S WRITING PROJECT

Appendix

i. Algorithm Code

a. Data Parsing

```
function loadMatrix
  input: {$r_1$,$r_2$,...,$r_n$}
  input: $r_i$ = {$j_1$,$j_2$,...,$j_k$}
  $M_{i,j}$ ← 0
  for i in 1 to n
    for each j in $r_i$
      $M_{i,j}$ ← 1
  output: $M_{i,j}$
```

b. Dimensionality Reduction

```
function reduceSpace
  input: $M_{i,j}$
  for j in 1 to k
    if count($M_j$) >= threshold then
      Q ← add($v_j$)
  output: Q = {$v_1$,$v_2$,...,$v_k$}
```

c. Graph Traversal

```
function traverseGraph
  input: Q = {$v_1$,$v_2$,...,$v_k$}
  for k in 1 to m
    $\xi_k$(shape) ← Ø
    $\xi_k$(count) ← 0
  for j in k down to 1
    name ← string(j)
    $\Delta^0_{new}$ ← new($\Delta^0$)
    $\Delta^0_j$ ← buildSimplex($\Delta^0_{new}$,name,$v_j$,Q)
    $\xi_k$ ← connectSimplex($\Delta^0_j$)
  output: {$\xi_1$,$\xi_2$,...,$\xi_m$}
  output: $\xi_k$ = (shape, count)
```

d. Build "Self-Contained" Simplex

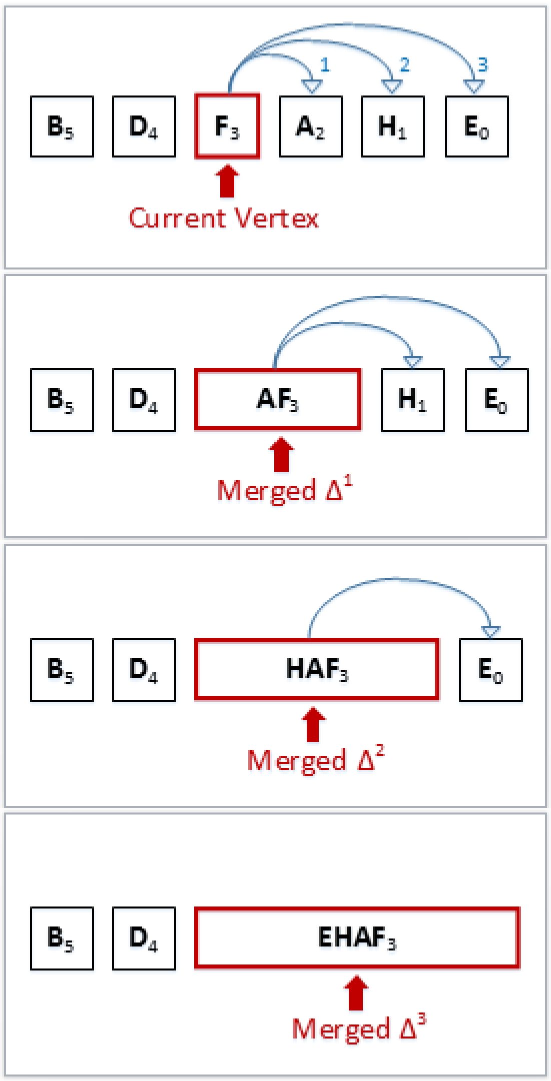
```
function buildSimplex
  input: $\Delta^i_{in}$(name,simplex,faces,links)
  input: $\sigma_j$ = {$\sigma_1$, $\sigma_2$,...,$\sigma_m$}
  input: {$\Delta_1$,$\Delta_2$,...,$\Delta_k$}
  $\Delta^i_{out}$ ← clone($\Delta^i_{in}$)
  $\Delta^i_{out}$(name) ← name
  $\Delta^i_{out}$(simplex) ← i
  $\Delta^i_{out}$(faces) ← {$\sigma_1$, $\sigma_2$,...,$\sigma_m$}
  for n in (j + 1) to k
    $\Delta^0_{new}$ ← new($\Delta^0$)
    $\Delta^0_{new}$(name) ← $\Delta_n$(name)
    $\Delta^0_{new}$(simplex) ← $\Delta_n$(simplex)
    for p in 1 to m
      if $\sigma_p$ in $\Delta_n$(faces) then
        $\Delta^0_{new}$(faces) ← add($\sigma_p$)
    if $\Delta^0_{new}$(faces) >= threshold then
      $\Delta^i_{out}$(links) ← add($\Delta^0_{new}$)
  output: $\Delta^i_{out}$
```

e. Cone Construction (connecting simplexes)

```
function connectSimplex
  input: $\Delta^i_{in}$(name,simplex,faces,links)
  $\xi_k$(shape) ← $\Delta^i_{in}$(simplex)
  $\xi_k$(count) ← increment by 1
  print($\Delta^i_{in}$(name))
  print($\Delta^i_{in}$(faces))
  if $\Delta^i$ < $\Delta^{max}$ then
    Q: {$\Delta_1$,$\Delta_2$,...,$\Delta_k$}
    Q ← $\Delta^i_{in}$(links)
    name_s ← $\Delta^i_{in}$(name)
    for n in 1 to k
      name_l ← $\Delta_n$(name)
      name ← name_s +" " + name_l
      $\Delta^{i+1}_{new}$ ← new($\Delta^{i+1}$)
      $\sigma_j$: {$\sigma_1$, $\sigma_2$,...,$\sigma_m$}
      $\sigma_j$ ← $\Delta_n$(faces)
      $\Delta^{i+1}_n$ ← buildSimplex($\Delta^{i+1}_{new}$,name,$\sigma_j$,Q)
      $\xi_k$ ← connectSimplex($\Delta^{i+1}_n$)
  output: $\xi_k$ = (shape, count)
```

ii. Simulation Runs

(1) Simulation Step #4 of *Self-Contained Cone Construction*:



Iteration #1:
Δ^1 **AF_3** = {**T3**, **T4**}
Δ^1 **HF_3** = {**T3**, **T4**}
Δ^1 **EF_3** = {**T3**}

Iteration #2:
Δ^2 **HAF_3** = {**T3**, **T4**}
Δ^2 **EAF_3** = {**T3**}

Iteration #3:
Δ^3 **$EHAF_3$** = {**T3**}

Iteration #4:
None left to merge (done)

Each of the iterations must operate within the four requirements that were explicitly stated in section 2.2.2.4. In addition, each step must also satisfy the three conditions listed in *Simulation Step #2* of that same section.

The key point to take away from these simulation steps is how each vertex is operating on its own and still able to leverage the concept of topological cone construction within the context of its ordered proximity. This is why we named this technique the "Self-Contained Cone Construction". And as soon as it moves out of its scope and into the next vertices, everything that was computed and stored by the previously visited vertices will be forfeited. This is absolutely necessary due to the exponential growth in number of permutations that are required to handle datasets with high number of selected features.

(2) Simplicial Algorithm Run Results:

Run command:

```
# <nrules> <threshold> <ncols> <nrows> <data-file>
SimplicialComplexNew.exe ^
4 0.0 10 5 ^
Inputs\dataSample-thesis-reduced.dat
```

Inputs\dataSample-thesis-reduced.dat

```
# <nvertices> <array-of-active-vertices>
4 0 1 3 4
3 1 3 4
4 0 4 5 7
3 0 5 7
2 4 7
```

Output results:

```
[E] 4
[H] 3
[H E] 2
[A] 3
[A H] 2
[A H E] 1
[A E] 2
[F] 2
[F A] 2
[F A H] 2
[F A H E] 1
[F A E] 1
[F H] 2
[F H E] 1
[F E] 1
[D] 2
[D A] 1
[D A E] 1
[D E] 2
[B] 2
[B D] 2
[B D A] 1
[B D A E] 1
[B D E] 2
[B A] 1
[B A E] 1
[B E] 2
```

iii. Standard Notations

Symbol	Meaning	Description
Δ^k	Delta-k (cap)	An *k*-simplex in *k*-dimensional space
Δ^0	Delta-0 (cap)	0-simplex or a vertex on topological space
Δ^1	Delta-1 (cap)	1-simplex or a line on topological space
Δ^2	Delta-2 (cap)	2-simplex or a triangle on topological space
Δ^3	Delta-3 (cap)	3-simplex or a tetrahedron on topological space
$\mathbb{R}^k$		A topology space with *k* dimensions
$\mathrm{M}_{\mathrm{i,j}}$		A matrix of i rows and j columns
S		A finite set of points in *k*-dimensional Euclidean space ($\mathbb{R}^k$)
C		A convex hull of a finite set of points in Euclidean space
K	Kappa	A finite set of simplexes on a graph
ϵ	Epsilon	A subset of a given set
ζ	Zeta	Zeta function associated with a finite graph
$\sum$	Summation	Summation of values from … to …
σ	Sigma	Any face of a simplex
τ	Tau	Subset of any face of a simplex
Ψ	Psi	A geometric flag of an *n*-polytope
O	Big O	Limiting behavior of a function
ω	Omega	Asymptotically dominant quantity in big O notation
Ω	Omega (cap)	An asymptotic lower bound in big O notation
α	Alpha	1st angle in a triangle
β	Beta	2nd angle in a triangle
γ	Gamma	3rd angle in a triangle
Γ	Gamma (cap)	Neighborhood of a vertex on graph
δ	Delta	The minimum degree of any vertex on graph
η	Eta	The partial regression coefficient in statistics
θ	Theta	The angle to the x-axis in xy-plane
λ	Lambda	A scalar coefficient in linear algebra
μ	Mu	A scalar coefficients Points in a topological space
ν	Nu	Dimension of null space in mathematics
ξ	Xi	A universal set of objects in geometric space
φ	Phi	A scalar field to every point in space
π	Pi	Projection in relational algebra for databases
$\subseteq$	Subset	A subset of another set with less or same # of elements
$\cap$	Intersect	Intersection between two sets of data
$\cup$	Union	Union between two sets of data
$\emptyset$		An empty set

This table was generated based on mathematical symbols described in this Wikipedia page [1]: https://en.wikipedia.org/wiki/Greek_letters_used_in_mathematics,_science,_and_engineering#.CE.91.CE.B1_.28alpha.29

Bibliography

[1] Wikipedia, “The Free Encyclopedia”, Wikimedia Foundation, Inc., a non-profit organization, 2016.

[2] Allen Hatcher, "Algebraic Topology", 3rd Edition, Cornell University, 2002.

[3] James W. Vick, "Homology Theory: An Introduction to Algebraic Topology", 2nd Edition, 1994.

[4] T.Y. Lin, "Attribute (Feature) Completion - The Theory of Attributes from Data Mining Prospect", Data Mining, 2002 IEEE International Conference on Data Mining.

[5] T.Y. Lin, Albert Sutojo, Jean-David Hsu, "Concept Analysis and Web Clustering using Combinatorial Topology", SJSU Computer Science, IEEE International Conference on Data Mining.

[6] Jovan Popovic, Hugues Hoppe, "Progressive Simplicial Complexes (PSC)", Carnegie Mellon University, Microsoft Research, ACM SIGGRAPH 1997 Proceedings, 217-224.

[7] Rakesh Agrawal, Tomasz Imielinski, Arun Swami, "Mining Association Rules Between Sets of Items in Large Databases", IBM Research Center, 1993.

[8] Richard Hennigan, "Fast Simplicial Complex Construction for Computing the Persistent Homology of Very Large and High Dimensional Data Sets", UML Computer Science, 2015.

[9] Anton Dochtermann, Alexander Engstrom, "Algebraic Properties of Edge Ideals via Combinatorial Topology", Stanford University, 2008.

[10] P.Y. Lum, G. Singh, A. Lehman, T. Ishkanov, M. Vejdemo-Johansson, M. Alagappan, J. Carlsson & G. Carlsson, "Extracting Insights from the Shape of Complex Data using Topology", Nature Scientific Reports, Stanford University, 2013.

[11] MapR Technologies, "Introduction to Big Data", 2014.

[12] Dominique Attali, Andre Lieutier, David Salinas, "Efficient Data Structure for Representing and Simplifying Simplicial Complexes in High Dimensions", 27th Annual Symposium on Computational Geometry (SoCG 2011), Jun 2011, Paris, France, 2011. <HAL-00579902>

[13] Anna Gundert, "On the Complexity of Embeddable Simplicial Complexes", 2009.

[14] William Harvey, Yusu Wang, Rephael Wenger, "A Randomized O (m log m) Time Algorithm for Computing Reeb Graphs of Arbitrary Simplicial Complexes", Ohio State University, 2010.

[15] Laszlo Lovasz, "Topological Methods in Combinatorics", Ohio State University, 2013.

[16] Bradford Barber, David P Dobkin, Hannu Huhdanpaa, "The Quickhull Algorithm for Convex Hulls", University of Minnesota, Princeton University, Configured Energy Systems, Inc., 1996.

[17] Jeffrey Dean, Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004.

[18] Milka Doktorova, "Constructing Simplicial Complexes Over Topological Spaces", Dartmouth College, 2012.

[19] Tsau-Young (T. Y.) Lin, David Le, Jingjing Yang, "High Speed Association Mining – Data Analysis Using Topology", Department of Computer Science, San Jose State University, 2015.

(1) NetApp, IDC Worldwide Big Data Technology and Services, 2012-2015 Forecast, #233485, March 2012. http://siliconangle.com/blog/2012/05/21/when-will-the-world-reach-8-zetabytes-of-stored-data-infographic/

(2) High Scalability, “Building Bigger, Faster, More Reliable Websites”, September 2012. http://highscalability.com/blog/2012/9/11/how-big-is-a-petabyte-exabyte-zettabyte-or-a-yottabyte.html

(3) The Medium, “The History of Hadoop”, April 2015. https://medium.com/@markobonaci/the-history-of-hadoop-68984a11704

(4) Neev Technologies, “Hadoop Ecosystem at a Glance”, March 2013. http://www.neevtech.com/blog/2013/03/18/hadoop-ecosystem-at-a-glance/

(5) Netflix Life, Netflix image, September 2014. http://netflixlife.com/files/2014/09/d3302ac4b1333f5017b25a9cf8bc1198_large.jpeg