EFFECTS OF IRREGULAR TOPOLOGY
IN
SPHERICAL SELF-ORGANIZING MAPS

A Thesis

Presented to the

Faculty of

San Diego State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Geography

---

by

Charles R. Schmidt

December 2008

SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Charles R. Schmidt:

Effects of Irregular Topology in Spherical Self-Organizing Maps

---

Sergio J. Rey, Co-Chair
Department of Geography

André Skupin, Co-Chair
Department of Geography

Robert P. Malouf
Department of Linguistics and Asian/Middle Eastern Languages

Approval Date

DEDICATION

Dedicated to my nieces and their future success.

ABSTRACT OF THE THESIS

Effects of Irregular Topology in Spherical Self-Organizing Maps
by
Charles R. Schmidt
Master of Science in Geography
San Diego State University, 2008

The traditional Self-Organizing Map (SOM) uses a rectangular or hexagonal network topology. These topologies create a well-known problem in SOM called the boundary or edge effect. Toroidal and spherical topologies have been widely suggested as a solution to the problem. However, there exist only five arrangements on the sphere which are completely regular; these are the five platonic solids (Ritter, 1999; Harris et al., 2000). Any other arrangement of neurons on the surface of the sphere will result in an irregular topology, as not all neurons will have the same number of neighbors. The classic method for minimizing this irregularity is to generate the spherical lattice by tessellating the sides of the icosahedron (Nishio et al., 2006). While this method will always result in a highly regular spherical topology, the main drawback is that the number of neurons in the network (the network size), N, grows exponentially as tessellations are applied. That results in only very coarse control over network size. We explore the effect of different topologies on properties of SOM. We suggest several diagnostics for measuring topology-induced errors in SOM and use these in a comparison of four different topologies. The results show that SOM is less sensitive to localized irregularities in the network structure than the literature may otherwise suggest. Further, the results support the use of spherical topologies as a solution to the boundary problem in traditional SOM.

TABLE OF CONTENTS

PAGE

LIST OF TABLES

PAGE

LIST OF FIGURES

PAGE

ACKNOWLEDGEMENTS

I would like to thank my committee for their patience and support through this process. I would also like to thank Philip Stephens and Boris Dev for their invaluable help and Maribel Elias for getting me out of the office on occasion. And the rest of the Regal team for friendship and support.

CHAPTER 1

INTRODUCTION

The Self-Organizing Map (SOM) is an unsupervised competitive learning process developed by Teuvo Kohonen as a technique to analyze and visualize high dimensional data sets. The applications of SOM are far reaching; Kohonen (2001) provides a thorough review of the SOM literature including applications of SOM. SOM has been used in applications ranging from speech recognition and image classification to breast cancer detection and gene expression clustering. Skupin and Agarwal (2008) outline the growing interest in SOM within GIScience, and propose that the relationship between SOM and GIScience should be bidirectional. The SOM offers a powerful method for exploring and visualizing geographic data and GIScience offers a wide array of tools and methods to enable the exploration of the SOM itself. The exploration of spatial relationships has always been of great interest to geographers, and as Ritter (1999) states, the goal of SOM is "to translate *data similarities* into *spatial relationships*" (Ritter, 1999, p. 1). This thesis leverages GeoVisualization and GeoComputation in order to explore some of the basic properties of the SOM.

The SOM is a type of artificial neural network in which neurons are "organized" in such a way as to project the high-dimensional relationships of a set of training data onto a low-dimensional network structure. The traditional SOM uses a rectangular or hexagonal network topology (Kohonen, 2001). These topologies create a well-known problem in SOM called the boundary or edge effect. Neurons on the boundary of the hexagonal and rectangular lattices have fewer neighbors, which reduces their ability to interact with other neurons during the self-organizing process. Using a spherical lattice has been widely suggested as a solution to the problem (Ritter, 1999; Boudjemai et al., 2003; Sangole and Knopf, 2003; Nishio et al., 2006; Wu and Takatsuka, 2006). The use of the spherical lattice, however, does not

completely overcome the problems caused by topology, and the choice of which spherical topology to use for the network can be difficult to make.

A regular network topology is one in which every node on the network has exactly the same number of adjacent nodes. Any topology involving an edge is irregular. Arranging our lattice on the surface of a sphere seems to be an obvious way to overcome the edge. However, there exist only five arrangements on the sphere which are completely regular; these are the five platonic solids (Ritter, 1999; Harris et al., 2000). Any other arrangement of neurons on the surface of the sphere will result in an irregular topology, as not all neurons will have the same number of neighbors. The classic method for minimizing this irregularity is to generate the spherical lattice by tessellating (subdividing) the sides of the icosahedron (Nishio et al., 2006). While this method will always result in a highly regular spherical topology, the main drawback is that the number of neurons in the network (the network size), N, grows exponentially as tessellations are applied. That results in only very coarse control over network size. Other methods for arranging neurons on the sphere allow for unlimited control over network size, but yield topologies with increased irregularity (Harris et al., 2000; Wu and Takatsuka, 2005; Nishio et al., 2006). To date the literature has largely ignored the more irregular methods in favor of the aforementioned tessellation-based methods. A topology which yields a more flexible network size may be desirable. However, in order to address this issue of network size, we must first determine the degree to which irregularity effects the SOM.

1.1 RESEARCH OBJECTIVES

The objective of this research is to determine the utility of certain irregular spherical topologies beyond offering greater control over network size. Toward that end, we develop and test new diagnostics to measure and visualize topology-induced errors in SOM. The following diagnostics are developed and implemented:

1. Compare the internal heterogeneity of observations captured by a given neuron to that neuron's first-order neighborhood size.

2. For different topologies, compare the internal heterogeneity of each neuron against a composite measure of topological regularity.

3. Develop a SOM-based visualization of the internal heterogeneity.

These diagnostics help facilitate the evaluation of both traditional and spherical SOMs. To satisfy the objective of this research, we apply these diagnostics to a series of comparable SOMs. Each SOM is trained using the same synthetic data and training parameters, but utilize different network topologies. By formally testing for difference of means and variance in the results of the diagnostics, the following questions are addressed:

1. Does the internal heterogeneity of a neuron decrease as its first-order neighborhood size, or degree, increases?

2. Is the average internal heterogeneity of a SOM higher when a more irregular topology is used?

3. Which insights, if any, can be gained from a SOM-based visualization of internal heterogeneity?

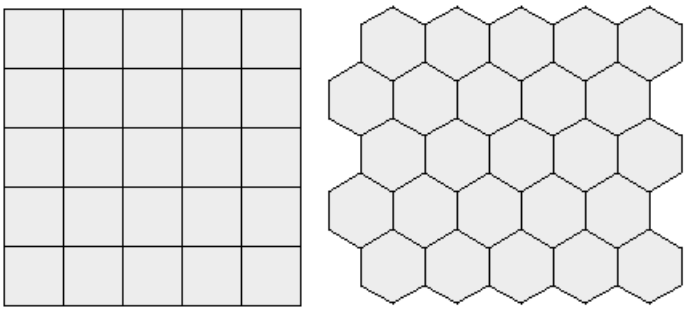
CHAPTER 2

BACKGROUND

This chapter is divided into six sections. Section 2.1 introduces the SOM and some of its basic properties. Next, section 2.2 discusses the training process which allows the SOM to represent our input data. In section 2.3 we explore the importance of topology in the SOM. The following two sections, 2.4 and 2.5, expand on this discussion by introducing some potential problems and possible solutions found with the traditional topologies used in SOM. Specifically, section 2.4 describes edge effects and section 2.5 describes the use of spherical topologies which attempt to overcome these edge effects. Finally, section 2.6 highlights the importance of the network size in SOM.

2.1 SELF-ORGANIZING MAPS

The SOM is a type of artificial neural network developed by Teuvo Kohonen. In the SOM an input-space is organized onto a set of neurons through an unsupervised competitive learning process (Kohonen, 2001). The neurons, arranged on a lattice, compete for input signals. The input signals are samples from an input-space. During training the winning neurons and those around them are updated to better model these signals. In each step of the training process the winning neuron is the one most similar to a given input signal. Traditionally, the neural lattice, or network, has either a rectangular or hexagonal topology. As shown in Figure 2.1 these topologies have different properties. In the rectangular topology, Figure 2.1(a), internal neurons are bounded by four neighbors. In the hexagonal topology, Figure 2.1(b), internal neurons are bounded by six neighbors. In both topologies, edge neurons have fewer neighbors. As the learning progresses, *data similarities* in the high-dimensional input-space are translated into *spatial relationships* in the low-dimensional neural network (Ritter, 1999).



(a) Rectangular Topology (b) Hexagonal Topology

Figure 2.1. In traditional SOM either a rectangular or hexagonal topology is used.

The SOM has a number of applications, but is primarily used for data reduction and data visualization. The SOM is often compared with other data reduction techniques, such as principal components analysis or multi-dimensional scaling. Like SOM, these techniques reduce the dimensionality of the input-space (Kohonen, 2001; Skupin and Agarwal, 2008). However unlike SOM, these techniques do not directly perform clustering. The SOM on the other hand has the ability to do both simultaneously. That is the SOM can collapse a high dimensional input-space into two dimensions, *and* collapse observations from the input-space into groups or clusters. The degree to which clustering occurs is controlled by the size of the SOM. In smaller SOMs, as the neurons try to model the input-space, observations are "collected" by the neuron which models it most accurately. In larger SOMs, perhaps even where there are more neurons than observations, observations are still collected by their best model. However, other neurons map the intermediate areas between observations, providing a low-dimensional spatial layout of otherwise high-dimensional data.

Skupin and Agarwal (2008) demonstrate these properties when they use state level data from the U.S. Census Bureau to train two SOMs of different sizes. In the three-by-three (9) case the neurons act as containers clustering similar states, while in the twenty-by-twenty (400) case relationships are expressed with much finer granularity. In the second case the SOM proves to be a very useful visualization tool. The similarities and dissimilarities among the states are represented as spatial orientations and distances. Even in this larger case, where neurons outnumber observations, clustering may still occur. This happens because the SOM

tries to represent the entire input-space. The SOM may better represent the input-space by not assigning observations to every neuron. In this case, neurons that have not collected any observations still represent dissimilarities between observations from the input space.

A useful property of the SOM is that the network structure connecting the neurons allows us to create meaningful visualisations. Observations used in the training, as well as new observations from the input space, can be mapped onto the trained surface in order to show higher dimensional relationships in a familiar map-like form. The SOM's component planes capture the spatial layout of each dimension, these are often visualized in a series of maps. These maps can provided useful information about the relationships between the different attributes of your input-space. In terms of information visualization, the spatial layout of observations on the network can provide far more insight than traditional methods, such as ordered lists or scatter plots.

2.2 TRAINING

As with other artificial neural networks, the SOM has to be trained with samples from the input-space. These samples, or observations, are represented as input vectors. During the training process neurons compete for inputs; with each training step winning neurons are adjusted to better match the signals they receive. Feedback between the neurons allows the entire network to eventually converge to a final state. After training, each neuron in the SOM will represent a portion of the input space. To accomplish this representation each neuron is associated with a parametric reference vector, m_i, referred to as a model vector (Kohonen, 2001). The length of each model vector is equal to the length of the input vectors, such that each element within a model vector represents a dimension of the input-space. The initial values of the elements are most commonly randomized, such that a mapping of the input-space onto the initial SOM would have no meaning. Other initializations are possible and may reduce the time required for the map to converge (Kohonen, 2001).

Our implementation follows the "Original Incremental SOM Algorithm" as laid out by Kohonen (2001). In each step of the algorithm, a randomly selected observation (input vector

x) searches for its best model (reference vector m_i) among the neurons. The best model is defined as the m_i with the smallest distance to x. These distances are referred to as quantization errors (QErrors), and they measure the distance between two vectors in attribute space (Kohonen, 2001). Our implementation uses Euclidean distances, however, any reasonable distance measure can be used here. The "winning" neuron is termed the Best Matching Unit (BMU c). The neighborhood (N_c) around the BMU (c) is found and all m_i within N_c are updated. The size of the neighborhood and the magnitude of the updates are controlled by the neighborhood function. In our implementation, the width of the neighborhood decreases as the training progresses, and the magnitude of the updates decrease, with a Gaussian function, toward the edge of the neighborhood. A learning-rate factor is used to further reduce the magnitude of the updates as training progresses. Combined these create the neighborhood kernel function h_{ci} which defines a scaler used to adjust the magnitude of each update in a given training step. This function always evaluates to zero for neurons outside the neighborhood. The update is defined as,

$$m_i(t+1) = m_i(t) + h_{ci}(t)[x(t) - m_i(t)] \tag{2.1}$$

where, t is the current training step. The training process is repeated a predefined number of times, or ideally until the map converges.

2.3 TOPOLOGY

Wu and Takatsuka state that "it is desirable to have all neurons receive equal geometrical treatment" (Wu and Takatsuka, 2006, p. 900). To satisfy this constraint, two conditions must be met. First, each neuron should occupy the same amount of space on the given surface. Second, each neuron should be bordered by the same number of surrounding neurons, and we should maximize that number. The first condition is largely irrelevant in the training of the SOM. However, visualizations that do not have uniformly sized and spaced neurons could potentially mislead an untrained viewer, as larger neurons may appear to be

more significant. Of greater importance to training is the SOM's topology, as it describes how the neurons are connected within the network. In training the topology defines, N_c, the neighborhood around the winning neurons, and irregular topologies directly impact the size and shape of these neighborhoods.

We believe the regularity of a given topology, as opposed to spacing and uniformity, is a better metric for evaluating different topologies for use in SOM. In graph theory a regular graph, is simply a graph in which every node has the same number of neighbors (Harris et al., 2000). A measure of regularity tells us how uniform neurons are in terms of their connections to other neurons in the network. In network theory, nodes with more connections are thought to be more central to the network and have a larger influence than nodes with fewer connections (Wasserman and Faust, 1994). A simple measure for capturing this is degree centrality. The degree (number of adjacent neurons) is measured for each neuron in the network. The variance in these measurements tells how regular the network is, a perfectly regular topology should have a variance equal to zero. Other methods, such as closeness centrality compare nodes based on their connectedness to every node in the network.

2.4 THE BOUNDARY EFFECT

Traditionally the SOM is laid out on a two-dimensional plane using either a rectangular or hexagonal topology. Both of these topologies are irregular, because neurons on the boundary of the network have fewer neighbors. Neurons with fewer neighbors have fewer chances of being updated (Wu and Takatsuka, 2006). As observed in Figure 2.2, neurons in the center of the map tend to better represent the mean of the input-space. In this SOM we used the same data as Skupin and Agarwal (2008) to map the fifty states and the District of Columbia onto a SOM trained with thirty-two population census variables. After training, we measured the distance between each neuron's model vector, m_i, and the mean of the input vectors, $\bar{x}$. Darker neurons have a relatively larger difference from the mean of the input-space, while lighter neurons are relatively closer. We also measured the difference of each observation to the mean $\bar{x}$, these distances are represented by the size of the point

symbols. Larger symbols are farther from the mean. The five observations furthest from the mean: Alaska, District of Columbia, Hawaii, Maine, and Utah are highlighted with bold labels. The five observations closest to the mean: Alabama, Illinois, Louisiana, Nebraska and North Carolina have underlined labels. Arguably the outliers are being pushed to the edges of the map, where they encounter fewer competing signals. Edge effects are also common in spatial analysis. For example, in point patterns the edge of a study unit may hide the true distribution of an observed pattern. In SOM, the edge of the neural lattice represents a true boundary, which effects its ability to represent *data similarities* as *spatial relationships*.

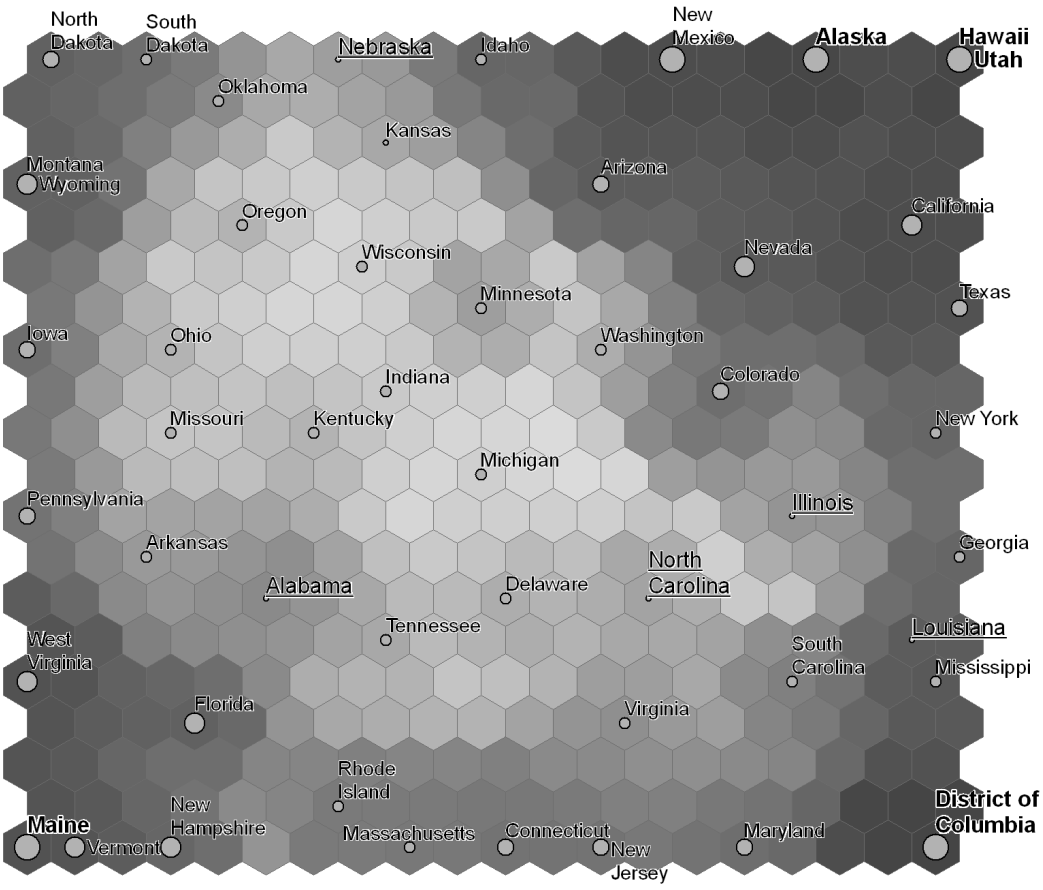


Figure 2.2. An example of the boundary effect in SOM.

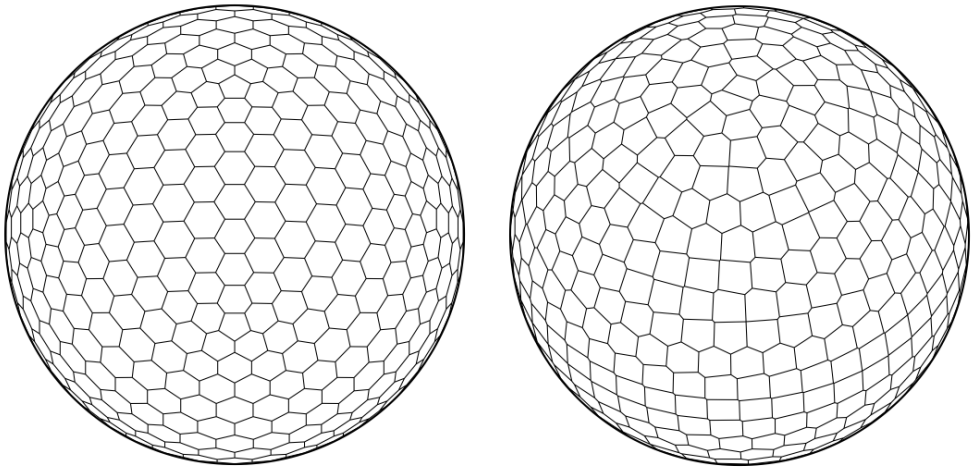
One way to eliminate the edge effect is to wrap the lattice around a three-dimensional object such as a sphere or torus, thereby removing the edge entirely. The toroidal SOM was

introduced by Li et al. (1993), however the torus is not effective for visualization, as maps generated from a torus are not very intuitive (Ito et al., 2000; Wu and Takatsuka, 2006). Ritter (1999) describes the torus as being topologically flat and suggests that a curved topology, such as that of a sphere, may better reflect directional data. A sphere also results in a more intuitive map, since we are accustomed to looking at geographic maps based on a sphere.

2.5 SPHERICAL SOM

Ritter (1999) first introduced the spherical SOM, and several enhancements have since been suggested (Boudjemai et al., 2003; Sangole and Knopf, 2003; Nishio et al., 2006; Wu and Takatsuka, 2006). A good comparison of these enhancements can be found in Wu and Takatsuka (2006). All of these methods derive their spherical structure through the tessellation of a polyhedron as originally proposed by Ritter (1999). Wu and Takatsuka (2006) point out the importance of a uniform distribution on the sphere, and that it is preferable for all neurons to have an equal number of neighbors and to be equally spaced. They find generally that the tessellation method best satisfies these conditions, and specifically that the icosahedron is the best starting point (Wu and Takatsuka, 2005). Tessellation of the icosahedron results in a network of neurons, each having exactly six neighbors, save the original twelve which each have five neighbors. This is very close to the ideal structure in which every neuron would have exactly six neighbors. Wu and Takatsuka (2006) prefer this structure, because it has very low variances in both neuron spacing and neighborhood size.

Based solely on measures of neuron spacing, Wu and Takatsuka (2005) dismissed the usefulness of a method proposed by Rakhmanov et al. (1994) for distributing points on a sphere. Figure 2.3 shows both the tessellated icosahedron (a) and a less uniform topology (b) derived from this method of distributing points. Similarly Nishio et al. (2006) use these variance measures to support their helix algorithm for distributing points on a sphere. Table 2.1 shows that these metrics can be misleading and comparison across topologies may not be consistent. The traditional rectangular and hexagonal topologies have no variance in neuron spacing, and the generally preferred hexagonal structure displays greater variance in



(a) Geodesic Topology (b) Spherical Topology

Figure 2.3. The geodesic topology is create by subdividing the sides of an icosahedron. The "spherical" topology is created by drawing Voronoi regions around points distributed on the surface of a sphere.

neighborhood size than the rectangular structure. The torus, by comparison, would have variance in neuron spacing, yet no variance in neighborhood size. The distance between two neurons is only considered during the formation of the network topology. At this stage the spacing is significant as it plays a part in constructing the network's topology by determining neuron adjacency. However, using this measure to evaluate potential topologies for use in SOM may be misleading.

Table 2.1. Variances in Topologies

Topology	Grid Size	Neuron Spacing	Variance in Neighborhood Size
Rectangular	9x18	1	0.2716
Hexagonal	9x18	1	1.2138
Geodesic	162	0.25319 - 0.31287	0.0686
Rakhmanov	162	0.15779 - 0.30069	0.2908

As spherical (and other alternative) topologies become increasingly more common it is necessary to investigate how the choice of topology effects the SOM. In this thesis the effect of irregularity within topologies is studied as an attempt to investigate not only the edge

effect, but also to help facilitate the comparison of topologies. It is important to note that spherical topologies may not be appropriate for all applications. Removing the edge may reduce the SOM's ability to converge. As outliers are forced to interact they introduce more competition among the neurons. We would also expect outliers to occupy more space in the final map as their dissimilarity in attribute space should translate to more distant spatial relationships in the trained SOM. More research will be needed to help researchers determine the most appropriate topology for their data and research objectives.

2.6 NETWORK SIZE

The number of neurons used in the SOM is a decision the researcher must make and is both a function of the size of their dataset and the purpose for which the SOM is to be used. Generally speaking fewer neurons would be used in a clustering application while larger SOMs are commonly used for visualizing high-dimensional datasets. The literature offers little theoretical guidance on choosing an appropriate network size for a given dataset (Cho et al., 1996). Vesanto (2005) suggests that the network size should be "as big as possible," but also states that this becomes computationally impractical for larger problems. As a general rule-of-thumb, Vesanto suggests using a network size of $5\sqrt{n}$, where n is the number of observations. The application for which this network size would be most relevant is unclear.

Given this lack of theoretical development, researchers should be cautious when using methods that limit the control of network size. Having a high level of control over network size allows support for such very different SOM applications as clustering versus low-dimensional spatial layout. Figure 2.4 shows how the achievable network size varies between topologies. In rectangular and hexagonal SOMs it is undesirable to have one dimension drastically larger than another, as such there are practical limitations to the size of these networks. As an aside, the preferred ratio between these dimensions depends on the data being represented, and should generally not equal one (Kohonen et al., 1996; Vesanto, 2005). In Figure 2.4 we used a ratio of one only for comparison purposes with the other topologies. Ritter's tessellation method results in a network size that grows at a rate of $N = 2 + 10 * 4^f$,

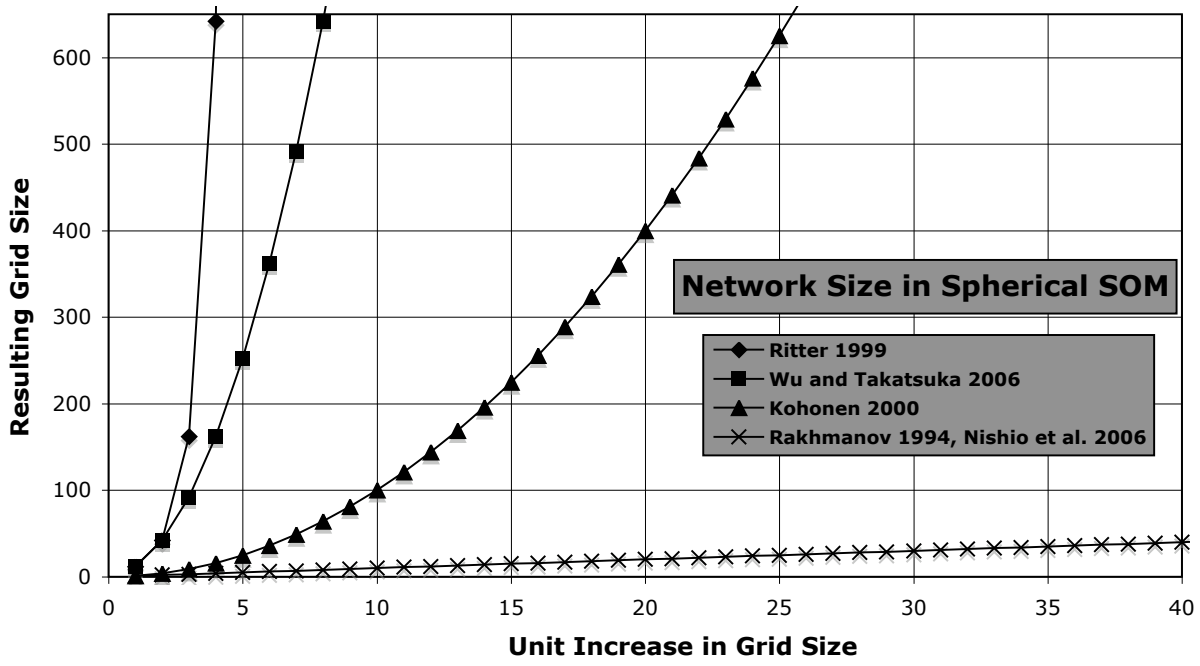


Figure 2.4. This figure demonstrates the achievable network size using various spherical topologies, in comparison with the traditional SOM. The Y-axis represents the achievable network size, while the X-axis represents the smallest increase in grid size for each topology.

where f is the frequency of tessellation. Wu and Takatsuka (2006) offer a slight improvement. Rather than recursively subdividing the faces, they redivide the original icosahedron with each step, resulting in $N = 2 + 10 * f^2$. Methods for arranging an arbitrary number of points on a sphere provide the highest degree of flexibility when choosing a network size. For example, the method proposed by Rakhmanov et al. (1994) can distribute any number of points onto the surface of a sphere. Strictly speaking, this is not a topology in itself, as no connections are defined between the points. In our implementation we create a spherical topology be applying Delaunay triangulation to these points (Ranka, 1997). We refer to this topology simply as "spherical". Using Ritter's method with a tessellation frequency of three, would result in 642 neurons, the next smallest size is 162 neurons. The geodesic topology offers three additional levels between 162 and 642 neurons (252, 362, 492). Using the spherical topology however, we are not limited to these network sizes.

Similarly Nishio et al. (2006) try to address the issue of network size granularity by departing from the tessellation method and suggesting the use of a partitioned helix to uniformly distribute any number of neurons on a sphere. The method proposed by Rakhmanov et al. (1994) was dismissed by Wu and Takatsuka (2005) for failing to satisfy the uniformity conditions described above. Nishio et al. (2006) suggest their method for distributing points satisfies these uniformity constraints, however they do not describe a network topology. It is unclear how they define neighbor relationships and without clarification we cannot implement their topology. Methods for distributing points on the sphere, which allow for fine-grained control over network size, produce slightly more irregular topologies. However, no substantive discussion of these irregularities or their effects on SOM training exists in the literature. Network size plays an import role in the SOM and given that limited theoretical guidance is available for choosing network size, researchers should be cautious when using topologies that limit control over this parameter. Particularly for larger SOMs, the desired network size may not be achievable via tessellation of the icosahedron.

CHAPTER 3

METHODOLOGY

This chapter is composed of four sections. Section 3.1 describes PySOM, our own graph based implementation of SOM. PySOM provides the necessary inputs to our three diagnostics as described in section 3.2. These diagnostics were developed to help understand how irregularities in topology effect the SOM. To use these diagnostics we must train several SOMs with comparable training data and parameters. The specifics of our training data are explained in section 3.3. Section 3.4 describes the training parameters used across all of our SOMs.

3.1 GRAPH BASED IMPLEMENTATION OF SOM

The most widely available implementation of SOM is Kohonen's own SOM_PAK (Kohonen et al., 1996). SOM_PAK implements both the traditional rectangular and hexagonal topologies. However, implementations of the geodesic and spherical SOMs were not readily available at the time this thesis was written. In order to test these topologies it was necessary to implement our own version of SOM, PySOM. Because the goal of this thesis was to study different topologies for use in SOM, we created PySOM to be independent from the topology. That is, our implementation is not explicitly aware of the topology. Rather, we represent the topology of the SOM as a graph. The graph provides the necessary information to determine neuron adjacency and construct neighborhoods. The nodes in the graph are pointers to the reference vectors (m_i).

To allow for rapid development and cross-platform support we choose to write PySOM in the Python programing language. As noted by Rey and Janikas (2006) Python is an object-oriented programing language that is becoming increasingly popular in scientific computing. PySOM is maintained as an open-source project in hopes of facilitating

collaboration and further research. In its present form, PySOM has no user interface, however it is written as a Python library which allows all of its functionality to be accessed programmatically. Python also provides an interactive interpreter which may be used to access PySOM in a command line environment.

By abstracting the topology, PySOM can train a SOM using any topology for which a graph structure can be created. We leverage an existing graph library, NetworkX, to represent our topologies (Hagberg et al., 2008). Our neighborhood functions are built on top of this library. Apart from our neighborhood search functions, our implementation follows the original incremental SOM algorithm that is described by Kohonen (2001). We store our trained SOM in a similar fashion as SOM_PAK. The reference vectors are represented in a simple text file. These "codebook" files contain a description of the topology and other training parameters in the first line of the file. Each subsequent line lists the values for the parametric reference vectors, m_i, of the SOM's neurons (Kohonen et al., 1996). In addition to this codebook file we also store the graph as a serialized Python object. Creating the graph structures for the spherical topologies is considerably more complex and requires more computation than the traditional topologies.

We provide utility functions to create the graph structures for both the rectangular and hexagonal topologies. As shown in figure 3.1, to create the graph structure for the geodesic topology we use the "dome" software package, which creates a "WRL" file containing the points coordinates for each node in the geodesic dome (Bono, 1996). These coordinates are fed into the STRIPACK software program. STRIPACK computes both the Voronoi cells and their complement, the Delaunay triangulation, on the surface of the sphere (Ranka, 1997). We provide a utility script, which converts the WRL file format into the XYZ file format that STRIPACK expects. The Delaunay triangulation provides the graph structure between the neurons of the geodesic SOM. A similar process is used for the spherical topology. In this case we wrote a Python implementation of the method for distributing points on a sphere that was introduced by Rakhmanov et al. (1994). Once again the coordinates are fed into

STRIPACK. An additional utility program reads the output from STRIPACK and creates the NetworkX graph structure. PySOM has no direct graphical output, however several utility functions are provided to assist in the creation of visualizations. These functions create text files that are compatible with popular GIS software packages, namely ESRI's ArcGIS. ArcGIS provides tools to create ShapeFiles from these text files.

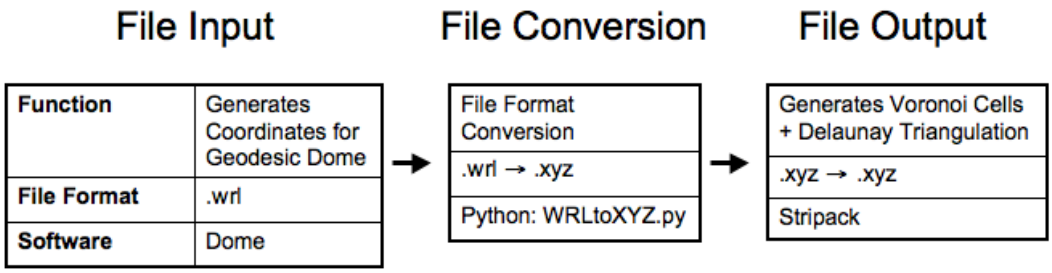


Figure 3.1. Steps for creating geodesic graph structures.

3.2 DIAGNOSTICS

In traditional SOMs, outlying observations are pushed to the edge of the map where they encounter fewer competing signals. A prime example of this is the "Utah-Hawaii" case shown in Figure 2.2. Relying only on the SOM, one would be left to believe that the two states are similar. Recalling that the QError measures the distance between two vectors in attribute space, we calculate that the QError from Utah to the neuron is 1.509, the QError from Hawaii to the neuron in 1.505, but the QError from Utah to Hawaii is 3.014. In this case only Utah and Hawaii were mapped to that neuron. In a case where multiple observations land on the same neuron, it is possible to measure the average pairwise QErrors between those observations. This gives us a notion of internal heterogeneity, H, for each neuron. We define the internal heterogeneity of neuron i as,

$$H_i = \frac{2}{{n_i}^2 - n_i} \sum_{j=1}^{n_i} \sum_{k=j+1}^{n_i} ||x_{ij} - x_{ik}|| \tag{3.1}$$

where, n_i is the number of observations mapped to i, and x_i are the input vectors mapped to i. The double bars represent the Euclidean distance between two observations which are references by j and k. We divide the sum of these distances by the number of pair-wise comparisons. For any neuron that captures more then one observation, this measure tells how dissimilar those observations are.

The edge effects in SOM make it clear that the compression of the input-space is not uniform throughout a trained map. While outliers being pushed to the edge of the SOM is not necessarily an undesirable outcome, it is important to understand when and where information is being compressed. This variable compression of the input-space is what allows the SOM to represent high-dimensional data, but it can also mislead the viewer. Observing the internal heterogeneity of neurons may shed light on the patterns of compression.

More specifically we have developed diagnostics that explore how irregularities in the topology of the SOM effect this internal heterogeneity. We compare the internal heterogeneity at the scale of the neuron and the overall map. Each diagnostic was designed to answer a specific research question. The first diagnostic addresses the research question regarding the internal heterogeneity and neighborhood size. The second diagnostic addresses the question concerning internal heterogeneity and topological irregularity. The third helps visualize the patterns between internal heterogeneity; the usefulness of which is examined in the next chapter.

3.2.1 Internal heterogeneity vs. first-order neighborhood size

This diagnostic compares the internal heterogeneity of each neuron against the neuron's first-order neighborhood size. It would be expected that in traditional SOMs neurons closer to the edge, those with fewer neighbors, will have higher internal heterogeneity. Neurons on the edge of the traditional (rectangular and hexagonal) topologies are considered more irregular, because they have fewer neighbors than neurons inside the edge. This can be extended to spherical SOMs by considering the degree of any given neuron. The degree of a

neuron $deg(m_i)$ measures the number of adjacent neurons. If the relationship between H_i and $deg(m_i)$ is consistent across topologies, we would expect neurons with lower degrees to display higher internal heterogeneity.

To implement this diagnostic we calculate the internal heterogeneity (H_i) and degree ($deg(m_i)$) of each neuron. The neurons are then separated into a small number of groups based on the degree. For most topologies the number of different degrees will be limited to three or four. The variance and mean is calculated for each of these groups. The expected result is that variances and means of the groups will decrease as the degree increases. This hypothesis is tested using random labeling as described by Rey (2004). In random labeling, we randomly assign our calculated H values to the neurons and recompute the mean and variance of the groups. We do this many times, 9999 in our case, in order to approximate the distribution under the null hypothesis that there is no relationship between the degree of a neuron and its internal heterogeneity. Finally we calculate pseudo p-values by comparing our observed mean and variance values with the simulated distributions. The results are also visualized using box-and-whisker diagrams. Box-and-whisker diagrams, or box plots, show the properties of a distribution. The diagram shows the mean, first and second standard deviations, and outliers that extend beyond the second deviation.

One problem that we face in this diagnostic is a small sample size when the neurons of a given SOM are grouped by their degree. For example, the four corners of the rectangular topology are the only neurons that have a degree of two. The rest of the neurons have three or four neighbors depending on whether or not they are on the edge. To address the problem of small sample size for topologies with relatively few neurons of a particular degree, we will increase the sample size by combining the results of many SOMs.

3.2.2 Internal heterogeneity vs. topological regularity

This diagnostic compares the internal heterogeneity of each neuron against a measure of regularity for its associated topology. As mentioned above the degree of each neuron

measures the number of adjacent neighbors. A completely regular network topology (i.e. the torus) has no variance between these measures. For irregular networks the variance between these degrees gives us a measure of irregularity. This particular measure is known as degree centrality. The degree of a node on a network is a measure of its centrality, or importance. Nodes with more connections are thought to be more central to the network and have a larger influence than nodes with fewer connections.

This holds with our understanding of the edge effect. Neurons on the edge are less central to the network and have less influence on training than other nodes. These edge nodes are also less influenced by the network, allowing outliers take root. During the training process, observations that are more average than others tend to be centralized. More extreme observations tend to be pushed to the edges. Referring back to figure 2.2, you'll notice that observations with smaller symbols are closer to the mean of the input-space and that these observations have been centralized in the network. Using the degree as a measure of centrality does not capture this picture well, as neurons near the edge can still have a large degree. A better way to capture this effect is to look at closeness centrality, which is the inverse of the average distance of a neuron to every other neuron on the network (Wasserman and Faust, 1994).

Closeness centrality provides a more complete measure of connectedness in a given topology than degree centrality. In this diagnostic we compare the internal heterogeneity of the neurons against the average closeness centrality of their respective topologies. This results in one group of internal heterogeneity measurements for each topology tested. We evaluated this diagnostic in much the same way as the last. We compare the variances and means of each group, testing for differences with random labeling. It is expected that the distribution of internal heterogeneity will be narrower for groups trained on more regular topologies. It is further hypothesized that the mean of internal heterogeneity will decrease when the network is more regular.

3.2.3 Visualize internal heterogeneity mapping

Visualizing the internal heterogeneity may yield insight into how irregular topology effects the SOM. Creating these visualization for many different topologies however, offers a number of challenges. While the rectangular and hexagonal topologies are rather straight forward to visualize, the spherical and geodesic topologies are significantly more involved. In order to leverage the utility of existing GIS software we represent our topologies in a form that these software packages understand. Toward that end we create polygon layers in which each polygon represents a neuron. Shared borders represent connections between neurons. Creating the polygons for these topologies required that we first compute the Voronoi diagram on the surface of the sphere. This is done using STRIPACK, a software program created by Ranka (1997). Despite the prevalence of spherical coordinates in GIS, modern GIS software packages have their roots in CAD software. As such they all assume Cartesian distances and thus can not handle polygons that cross the 180^{th} meridian. To accommodate this we split each polygon at the 180^{th} meridian and redraw it as two parts.

Once the GIS layers have been created and the internal heterogeneity of each neuron has been calculated, a number of visualizations become possible. We visualize the internal heterogeneity by shading the corresponding polygons. These visualizations allow for the exploration of patterns in internal heterogeneity with relation to the underlaying topology. Further, we can visualize the component planes, which show how the higher dimensions are represented in the various SOMs. We also map our synthetic data back onto the SOM in order to calculate cluster membership of the neurons. Visualizing cluster membership clearly shows how the various topologies perform clustering.

3.3 DATA

Our internal heterogeneity measure is sensitive to both the properties of the SOM and the properties of the training data. Therefore, a dataset with uniform properties is needed. We follow the method for generating uniform synthetic data used by Wu and Takatsuka (2006).

Their method creates seven clusters in three dimensions. The uniform clusters generated by this method allow us to systematically compare the diagnostics under several different topologies. Each cluster is normally distributed and has a standard deviation of one. The clusters are centered at the origin and ten units out in each directions on the x, y and z axes. To ensure that we can calculate an internal heterogeneity for as many neurons as possible, we create approximately $25,000$ observations in each dataset. As described in the next section, our SOMs have either 642 or 644 neurons (depending on the topology). Having a large number of observation relative to the number of neurons will force the SOM to perform clustering, increasing the number of neurons for which the internal heterogeneity can be computed.

As mentioned in section 3.2.1 we will need to combine the results of multiple SOMs in order to ensure a large enough sample size. To accomplish this we will create ten synthetic datasets as described above. Each of these datasets can be thought of as samples from the same input space. That is, the data generating process remains the same for each synthetic dataset created. After training ten SOMs for each topology we take the average internal heterogeneity of each trained SOM. We compare these results to ensure that trained SOMs behave similarly within each topology.

3.4 Training Parameters

Before we can go on to address the research questions we need to train a series of SOMs. We train SOMs using four different topologies: *rectangular, hexagonal, geodesic sphere* and *spherical*. The spherical topology is based on a method, developed by Rakhmanov et al. (1994), for distributing an arbitrary number of points on to the surface of a sphere. Delaunay triangulation is then applied to these points, producing a topological structure. To yield meaningful results these SOMs must be trained with comparable parameters. The literature provides many rules of thumb for training a SOM: each SOM is trained in two stages, the first of which uses a larger initial learning rate and neighborhood search radius with a small number of training steps; the second stage uses a lower initial learning rate and

neighborhood search radius, but extends the length of training. In PySOM we define the initial neighborhood search radius as a percentage of the maximum width of the graph.
First Stage Parameters:

- Initial neighborhood search radius of 50%, which decreases during training.
- Initial learning rate of 0.04 which decreases during training.
- 100,000 training steps.

Second Stage Parameters:

- Initial neighborhood search radius of 33%, which decreases during training.
- Initial learning rate of 0.03 which decreases during training.
- 1,000,000 training steps.

As shown in Figure 2.4, topologies differ in terms of achievable network size. For comparability, the network size of each SOM needs to be as close as possible. The achievable network size for the geodesic SOM is the most limiting of the topologies we test. We chose the eighth frequency geodesic sphere, which has 642 nodes, which is relatively close to the 644-node hexagonal and rectangular topologies achieved when the dimensions are set to $28x23$. Finally, the spherical topology was set to 642 nodes.

CHAPTER 4

RESULTS AND DISCUSSION

This chapter presents the results from our diagnostics. Section 4.1 reviews the training process. Section 4.2 discusses the application of the first diagnostic, which looks at how internal heterogeneity changes with a neuron's first order neighborhood size, or degree. In section 4.3 the second diagnostic, which compares the mean internal heterogeneity across topologies, is applied. Section 4.4 applies the third diagnostic, which visualizes the internal heterogeneity of each SOM. Finally, section 4.5 revisits the Utah-Hawaii example.

4.1 TRAINING

Before applying the diagnostics we must first train a series of SOMs. The training is accomplished using our graph based implementation of SOM, PySOM. Using the ten synthetic datasets we train ten SOMs for each topology, these trained SOMs represent forty simulations. The mean internal heterogeneity of these SOMs is summarized in Table 4.1. We find that the mean internal heterogeneity remains fairly stable, ranging from 0.0274 in the spherical topology to 0.0298 in the rectangular topology. This suggests that the results of each simulation can be combined within a given topology. For the rectangular topology, we now have forty neurons with a degree of two for which an internal heterogeneity can be calculated. It should be noted that we can only measure the internal heterogeneity when a neuron captures two or more observations from the training data. Therefore, it is still possible to have less than forty measurements.

We used several machines of varying configurations to train the forty different SOMS. On the fastest of those machines training one SOM took around forty-five (45) minutes. In contrast SOM_PAK takes only ninety (90) seconds to train the hexagonal SOM with the same training parameters. This significant difference is largely due to differences in the underlying

Table 4.1. Mean internal heterogeneity for each simulation, by topology.

Simulation	Geodesic	Spherical	Hexagonal	Rectangular
1	0.0277	0.0277	0.0285	0.0289
2	0.0281	0.0281	0.0291	0.0295
3	0.0278	0.0280	0.0286	0.0292
4	0.0280	0.0282	0.0286	0.0293
5	0.0279	0.0280	0.0289	0.0296
6	0.0278	0.0274	0.0285	0.0290
7	0.0286	0.0283	0.0294	0.0297
8	0.0284	0.0285	0.0294	0.0298
9	0.0283	0.0282	0.0293	0.0295
10	0.0285	0.0285	0.0293	0.0298
Combined	0.0281	0.0281	0.0290	0.0294

data structures. We traded speed for the ability to represent a SOM with any topology. Differences between the Python and C programing languages may also account for significant difference in runtime.

4.2 INTERNAL HETEROGENEITY VS. FIRST-ORDER NEIGHBORHOOD SIZE

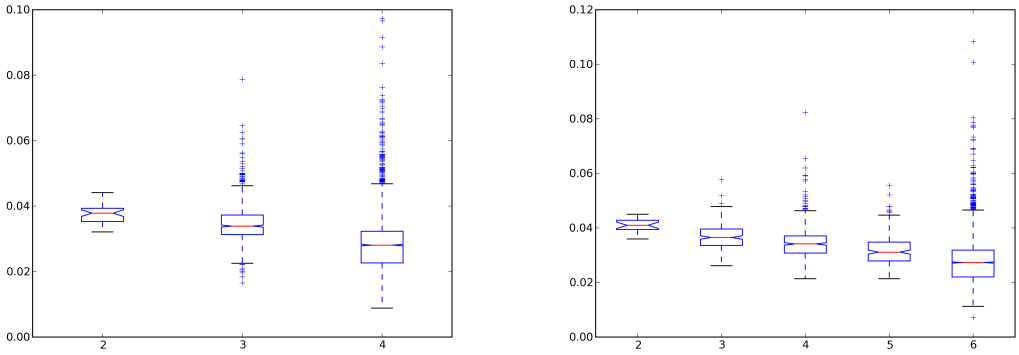
We hypothesized that outlying observations would migrate to less central neurons on the map, where they encounter less competition. We expected the internal heterogeneity as measured by equation 3.1 to increase at these locations, demonstrating that topological irregularity affects the placement of outliers in the SOM.

Table 4.2. Size, mean and variance of each sample

Degree	Geodesic		Spherical		Hexagonal		Rectangular	
	N	Mean (Var)	N	Mean (Var)	N	Mean (Var)	N	Mean (Var)
2					20	0.0409 (5.66E-06)	40	0.0378 (7.59E-06)
3					218	0.0371 (2.18E-05)	880	0.0348 (3.59E-05)
4					489	0.0347 (4.05E-05)	4926	0.0284 (6.28E-05)
5	113	0.0283 (5.28E-05)	526	0.0279 (6.43E-05)	206	0.0319 (3.50E-05)		
6	5598	0.0281 (6.05E-05)	4758	0.0282 (6.05E-05)	4954	0.0278 (6.09E-05)		
7			417	0.0273 (6.65E-05)				

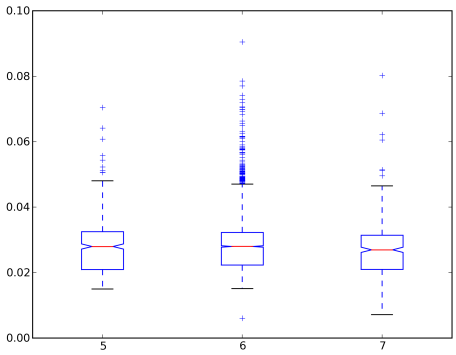
For each topology, we calculated the internal heterogeneity and degree of all the neurons. We then grouped the internal heterogeneity measures by degree. These groups are treated as representative samples from a larger distribution. Table 4.2 shows the details of

each sample, including its size, mean and variance. The size N is the number of observations for which we were able to calculate an internal heterogeneity. The mean and variance capture the centrality and dispersion of the samples. We create a box-and-whisker diagram (box plot) for each sample, as shown in figure 4.1, to visualize the samples summarized in table 4.2.

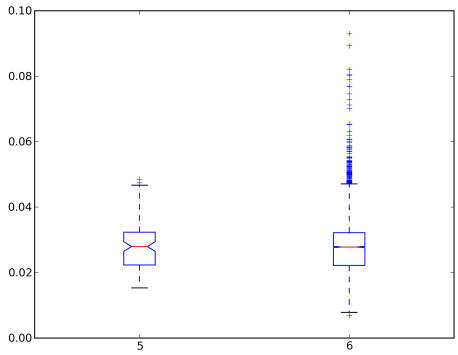


(a) Rectangular Topology

(b) Hexagonal Topology



(c) Spherical Topology



(d) Geodesic Sphere Topology

Figure 4.1. Box-and-whisker diagrams representing samples derived from forty trained SOMs. The samples within each topology were created by grouping the internal heterogeneity of each neuron of a given degree size. The diagrams how the centrality and dispersion of each sample.

We see that the means of the samples seem to respond as expected for the rectangular and hexagonal, or "flat," topologies, but not in the spherical topologies. While this is not what we anticipated, it may suggest that the spherical and geodesic topologies are effectively overcoming the edge problem. The variance of the samples, however, did not respond as

expected. In the flat topologies, the variance increased as the degree increased. This is possibly due to the large difference in sample size. To verify these conclusions we formally test for differences in means and variance between the samples.

Table 4.3. Results of Difference of Mean Testing Within Each Topology

(a) Rectangular Topology

Degree	2	3	4
2	(0.000008)	**0.002700**	**0.000100**
3	0.002969	(0.000036)	**0.000100**
4	0.009364	0.006396	(0.000063)

(b) Hexagonal Topology

Degree	2	3	4	5	6
2	(0.000006)	**0.000800**	**0.000200**	**0.000100**	**0.000100**
3	0.003810	(0.000022)	**0.000100**	**0.000100**	**0.000100**
4	0.006253	0.002443	(0.000040)	**0.000100**	**0.000100**
5	0.009008	0.005197	0.002755	(0.000035)	**0.000100**
6	0.013067	0.009257	0.006814	0.004059	(0.000061)

(c) Spherical Topology

Degree	5	6	7
5	(0.000064)	0.485500	0.197400
6	0.000250	(0.000060)	**0.020000**
7	0.000674	0.000924	(0.000067)

(d) Geodesic Topology

Degree	5	6
5	(0.000053)	0.783900
6	0.000203	(0.000060)

The results of the means test are presented in table 4.3. This table shows the difference in means below the diagonal, the p-value above the diagonal (with significant values in bold), and the variance of each sample along the diagonal. In the rectangular and hexagonal topologies, we observe that all sample means are significantly different. We also observed significant differences in the variance of the samples, except the case of degree size four and five in the hexagonal topology, where they were not significantly different. No variances in the spherical and geodesic topologies were significantly different. In those topologies, the

only significant difference in means was between the sample with degree size six and seven in the spherical topology.

4.3 INTERNAL HETEROGENEITY VS. TOPOLOGICAL REGULARITY

Above we saw that changes in a neuron's degree was related to changes in internal heterogeneity in topologies with edges, but not in the two sphere-based topologies. The next step is to see if internal heterogeneity changes between topologies. To do this we will first order our topologies by a summary measure of the topological regularity. For this diagnostic we use the average closeness centrality for each topology as the summary measure. The summary measure and the sorting of our topologies is shown in table 4.4.

Table 4.4. Measure of Topological Regularity and Sample Mean and Variance

Topology	Closeness Centrality	Mean	Variance
Rectangular	0.0603	0.0294	0.0001
Hexagonal	0.0739	0.0289	0.0001
Geodesic	0.0890	0.0281	0.0001
Spherical	0.0906	0.0281	0.0001

In this diagnostic we once again group the neurons of our trained SOMs. This time we group them based on their topology, collapsing the neurons of the forty SOMs into four groups. This results in one sample for each of the four topologies. We test for a difference in mean and variance between each sample using the same method of random labeling that was applied in the previous diagnostic. No significant differences were found in the variances; the difference in means are presented in table 4.5. As in table 4.3, this table shows the difference in means below the diagonal, the p-value above the diagonal (with significant values in bold), and the variance of each sample along the diagonal. It was expected that the mean and variance of the samples would decrease for the more regular topologies. These results generally support this hypothesis. We see that the rectangular topology has the highest mean internal heterogeneity and is the least regular as measured by closeness centrality. The

geodesic and spherical topologies are the most regular and have the lowest internal heterogeneity. These two groups display very similar measures of closeness centrality and show no significant difference in mean internal heterogeneity. This suggests that even though the spherical topology is more irregular than the geodesic topology, similar levels of quality may be achieved.

Table 4.5. Results of Difference of Mean Testing Across Topologies

Topology	Rectangular	Hexagonal	Geodesic	Spherical
Rectangular	(0.000064)	**0.001000**	**0.000100**	**0.000100**
Hexagonal	-0.000479	(0.000064)	**0.000100**	**0.000100**
Geodesic	-0.001329	-0.000850	(0.000060)	0.505600
Spherical	-0.001328	-0.000849	0.000001	(0.000061)

4.4 VISUALIZATION OF INTERNAL HETEROGENEITY

In order to address our research questions we first created ten synthetic datasets. Each dataset was created by sampling from the same data generating process that was provided by Wu and Takatsuka (2006). We used these datasets to train ten SOMs for each of our topologies. The purpose of training ten SOMs was to produce large enough sample sizes for the difference of means and variance testing; this was necessary to formally evaluate the results of our first two research questions. In this section we verify that combining those simulations was appropriate by visualizing the similarities between them. In the remainder of this section we focus our efforts on comparing internal heterogeneity across topologies. We see that little variation exists between the ten simulations, as such we choose the first of those simulations for each topology and explore them in more depth.

As expected, figure 4.2 reveals little variation between simulations of a given topology. This homogeneity demonstrates that our ten synthetic datasets adequately increase our sample size without introducing bias. Because of the apparent homogeneity between the simulations *within* a given topology, we move on to compare the first simulation *across*

topologies. Our synthetic training data consisted of seven clusters located in three dimensional space. We examine how the SOM treats the original data through a series of visualizations. The component planes in figure 4.3 show how the SOM represents each dimension of the training data. The variations we see in the placement of high and low values, between topologies, is caused by the random initialization of the SOM. However, the relationships between the dimensions remain constant across topologies.

Figure 4.4 shows how the SOM detected the clustering of the input data. We compute the cluster each neuron represents by looking at the observations mapped to it. Knowing which cluster each observation belongs to allows us to see where the clusters are being mapped to on the trained SOM. For a given neuron we classify it by the cluster it captured most frequently.

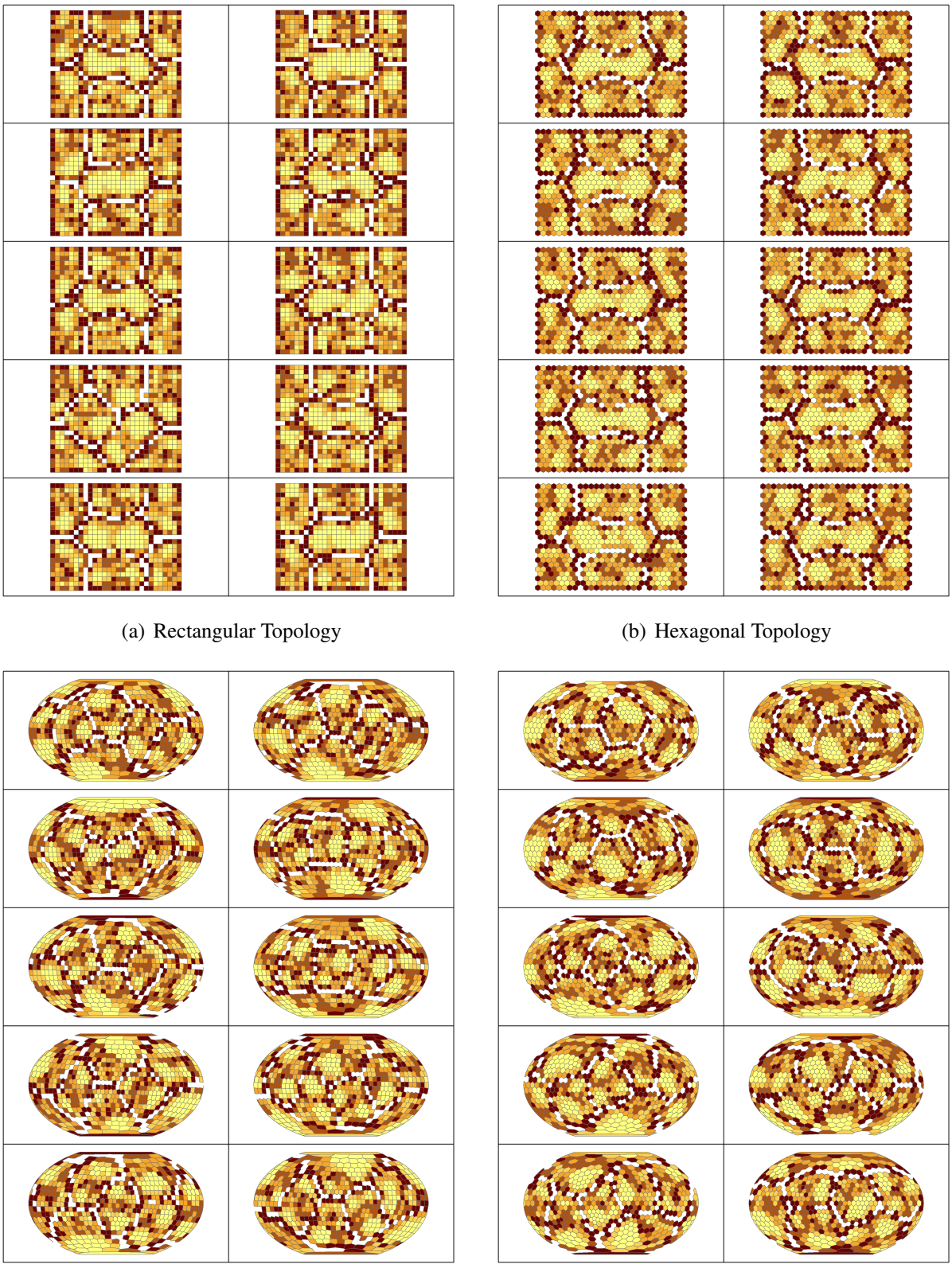
It is interesting to note that the edges of the cluster exhibit higher internal heterogeneity. This is somewhat intuitive, as our clusters are normally distributed; the majority of our observations will fall well within the regions observed in figure 4.4. Each cluster's outliers will be pushed toward the *edges* of these regions. An observation that is on the edge of a cluster in the original input-space is further away from the other observations in the input-space. Therefore, one would expect that observation to also be near the edge of a cluster in the SOM space. In order to represent a three dimensional cluster in two dimensions, the SOM must compress the edges of the clusters more than their centers. This explains the higher internal heterogeneity near the edges of the clusters.

4.5 THE UTAH-HAWAII PROBLEM

This research topic was initially inspired when an unwitting professor made the assumption that Utah and Hawaii were similar, because they both fell on the same neuron in a trained SOM. This raised the question of whether Utah and Hawaii were in fact similar, or if this relationship in the SOM was merely caused by an edge effect. It turns out the answer is slightly more complicated. We calculated the QErrors between every state and found that Utah and Hawaii are in fact each other's nearest neighbor, as defined by the census variables

used by Skupin and Agarwal (2008). However, as shown in Figure 2.2 and explained in section 2.4, both of these observations can be considered outliers. Neither fits the trained SOM well and both are far from the mean of all the states.

In Figure 4.5 we present the same data trained on a 400 neuron spherical SOM. The topology for this SOM is based on Rakhmanov et al. (1994). We use the same coloring scheme and labeling used in Figure 2.2. Darker neurons are further from the mean of the states, while lighter neurons are more similar to the mean. The size of the symbol represents the distance between each state and the mean of the states. The bold labels highlight the five observations that are furthest from the mean, while the underlined labels highlight the five states closest to the mean. We find, not surprisingly, that Utah and Hawaii are separated in this SOM. This implies that edge effects indeed played a large role in the placement of these two observations in the hexagonal SOM. As suggested in section 2.5, we also find these and other outlying observations occupy more space in the trained spherical SOM than in the hexagonal SOM. States closer to the mean such as Ohio, Missouri and Kentucky are distinctly more clustered in the spherical SOM than in the hexagonal SOM. It is interesting to note that the orientation between these three states remained largely unchanged between the two maps.



(a) Rectangular Topology

(b) Hexagonal Topology

(c) Spherical Topology

(d) Geodesic Sphere Topology

Figure 4.2. Internal heterogeneity mapping for each of the forty SOMs. Darker colors represent neurons that display larger internal heterogeneity. Neurons for which an internal heterogeneity could not be calculated are not shaded.

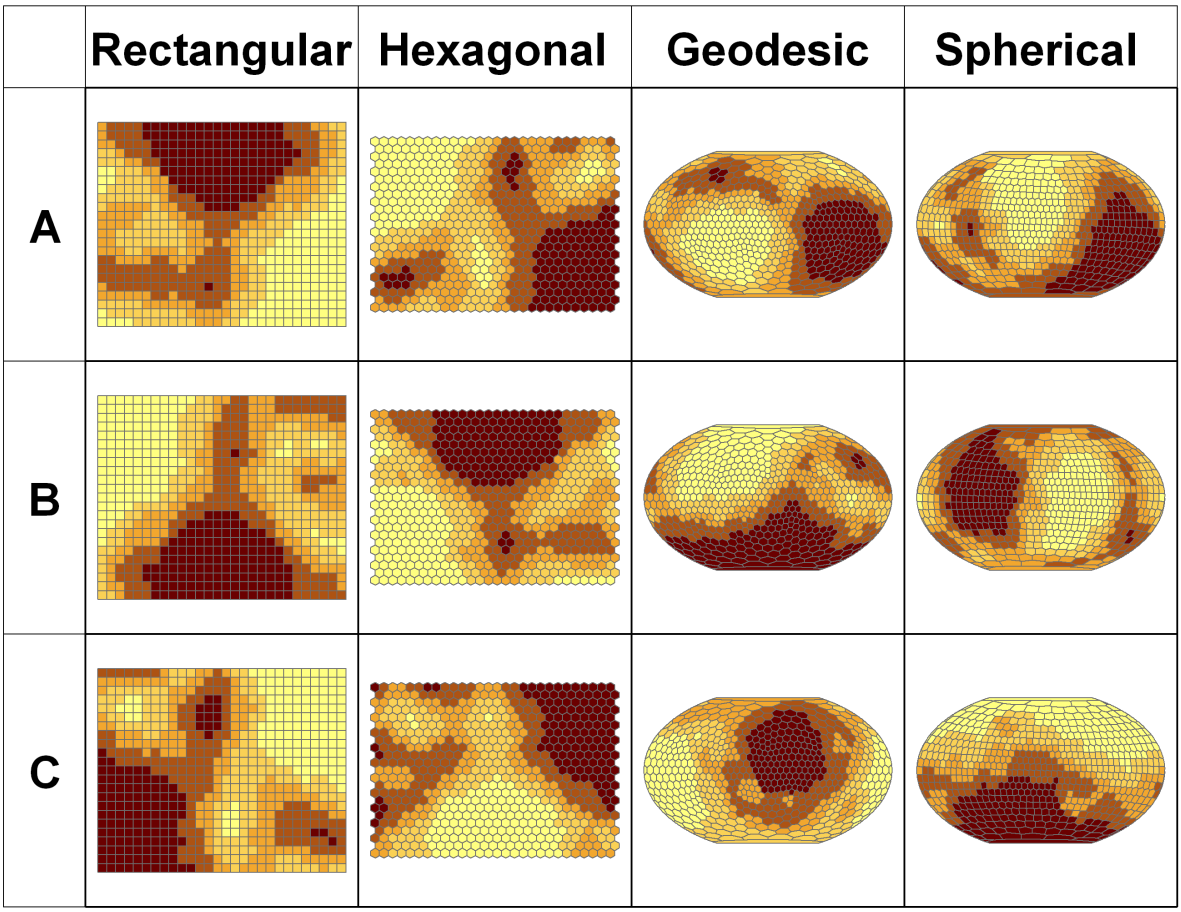
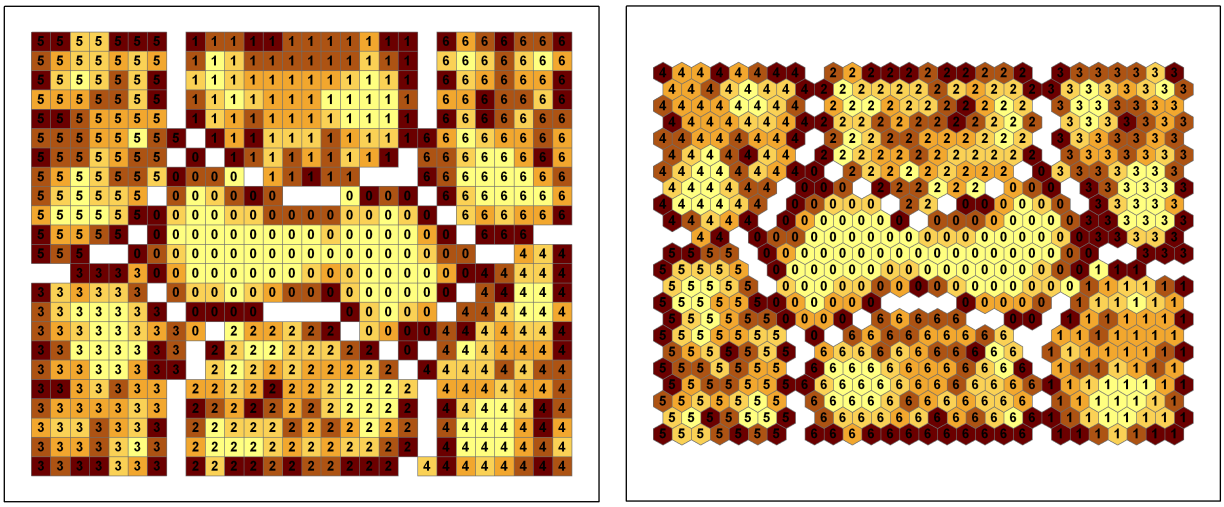


Figure 4.3. The first (A), second (B) and third (C) component planes are shown for the first simulation of each topology. These component planes show how the original dimensions are represented in the trained SOMs.



(a) Rectangular Topology

(b) Hexagonal Topology

(c) Spherical Topology

(d) Geodesic Sphere Topology

Figure 4.4. Detailed internal heterogeneity mapping for each topology. Darker colors represent neurons that display larger internal heterogeneity. Neurons for which an internal heterogeneity could not be calculated are not shaded. The numbers represent the primary cluster mapped at that neuron.

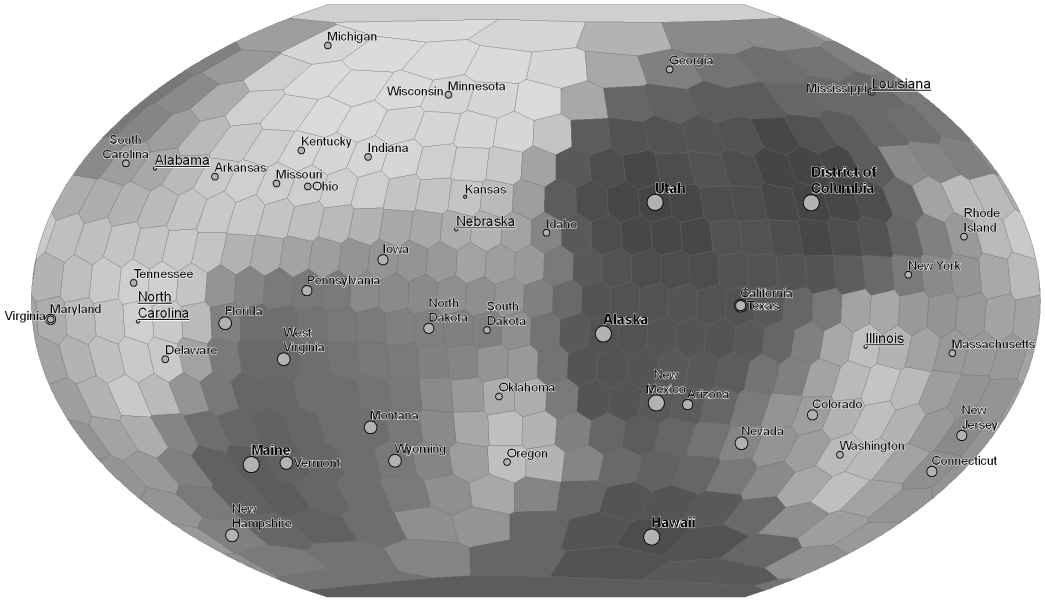


Figure 4.5. An example of how spherical SOM addresses the boundary effect found in traditional SOM.

CHAPTER 5

CONCLUSIONS

The spherical SOM has been widely suggested as a solution to the edge effect in traditional SOM. The edge of the traditional SOM is a problem because neurons on the edge of the topology are less central to the neural network than those in the center, resulting in differing levels of influence on the training process. We extended this problem to spherical SOM by asserting that irregular spherical topologies will have neurons that are less central than others. Existing research in spherical SOM has widely been focused on minimizing this irregularity and the commonly used tessellated icosahedron based topology offers the most regular topology. However, the main disadvantage of this topology type is that it offers very limited control over network size. Alternative methods for generating the spherical topology, which can create a network of any size, have been reviewed or suggested by Wu and Takatsuka (2005) and Nishio et al. (2006). These alternative methods have been largely dismissed because they produce network structures that are more irregular. This research has taken a closer look at the impact that irregularity has on the training process in an attempt to address the suitability of these less regular topologies for use in SOM.

The objective of this research was to determine the utility of certain irregular spherical topologies that offer greater control over network size. Toward that end, new diagnostic measures were developed that allow for SOM comparisons based on topology-induced errors. The diagnostics measure the internal heterogeneity of observations captured by a given neuron relative to that neuron's first-order neighborhood size, as well as relative to a composite measure of topological regularity.

In this study the new diagnostics were applied to the evaluation of both traditional and spherical SOMs. Each SOM was trained using the same synthetic data and training parameters, but utilizing different network topologies. By formally testing for difference of

means and variance in the results of the diagnostics, the following conclusions were made: 1) The internal heterogeneity of a neuron decreases as its first-order neighborhood size increases, but less so in spherical topologies. 2) The average internal heterogeneity of a SOM is higher when a more irregular topology is used. 3) Visualizing internal heterogeneity provides valuable insight into the underlying training process and how the SOM represents clusters. From these conclusions, we can make interpretations on the importance of topological irregularities in SOM.

In comparison to traditional rectangular and hexagonal topologies, the effects of irregular topology were minimized in both of the spherical topologies we tested. Specifically, we found no significant differences between the two different spherical topologies we test. As such, we believe that the importance of regularity within spherical SOM may be overstated. Relying only on highly regular spherical topologies may place unnecessary constraints on the users of spherical SOM. For example, using less regular spherical topologies we can overcome the resolution constraint and build SOMs with any number of neurons.

In pursuit of these findings we built an open source implementation of SOM, PySOM. This software is capable of training a SOM with an arbitrary topology. This allowed us to test the four different topologies presented in the previous chapters. Using PySOM, future researchers may train their SOMs on a variety of different topologies. We hope that this will enable an exploratory based approach to help determine the most appropriate topology for a given research question.

5.1 LIMITATIONS

A key limitation of this research is that we only address how topological regularity effects the SOM. We do not address how other attributes of the topology effect the SOM. Specifically, the benefits and/or costs of removing the edge are unclear. More research needs to be done in order to determine if spherical SOM is appropriate for all applications, or under which conditions it becomes inappropriate. The edge effect may in fact be useful in identifying outliers.

Our implementation, PySOM, is significantly slower than existing implementations of the SOM. As it is currently written it would not be of much use for extremely large SOMs due to the speed issues. PySOM currently does not implement the shortcut winner search described by Kohonen (2001). The shortcut winner search is a method in which an observation only searches the entire network for its best match once. On subsequent iterations that particular observation will only look in the neighborhood of its previous best match. If the new best match is on the edge of that neighborhood the search radius is expanded.

5.2 Future Directions

It would be beneficial to expand this research to investigate other aspects of topology and their effects on the SOM. The graph structure allows for great flexibility in testing changes to the topology. One could construct an experiment in which individual nodes are removed from the network topology to determine the impact of an individual neuron. It could also be beneficial to test other topologies using the framework laid out by this thesis.

Using spherical SOMs introduces many challenges in terms of visualization, particularly in projecting spherical SOMs into two-dimensional Euclidean space. The random initialization of the SOM results in an arbitrary rotation of the trained SOM. It would be helpful when visually comparing two SOMs if they were both rotated in a consistent fashion. One could accomplish this by identifying the two most prominent features of the trained SOM and rotating the SOM such that the most prominent feature is located at $0°$Latitude, $0°$Longitude and the second most prominent feature is located as far North as possible. A third feature could be required if the first two are antipodal. The challenge in this lies in identifying features in the SOM. A variety of metrics could be used to rank individual neurons. Alternately, one could cluster the neurons and use the resulting regions as features.

A number of improvements can be made to PySOM in order to make it of use to the broader scientific community. Implementing the shortcut winning search could dramatically reduce running time. Python allows for portions of the code to be written directly in the C programming language. The highly computational sections of the code, such as search for the

best matching unit, have the potential to run significantly faster in C. The neighborhood search routines can also be optimized. For example, caching the neighborhood around a given neuron would drastically reduce the number of neighborhood searches but increase the program's memory requirements. Perhaps the most useful addition to PySOM would be a graphical user interface (GUI). A well designed GUI would greatly increase the accessibility and usability of this code.

Traditionally the geographical sciences have been concerned with identifying and understanding relationships found in the context of spatial data. The existing toolbox for studying these relationships is sizable and steadily growing as geographers develop new ways to look at spatial information. While the SOM and spherical SOM are a welcome addition to that toolbox these tools provide much more than a new visualization method or clustering technique. Using SOM we can extract spatial representations from otherwise aspatial data, allowing us to leverage our existing set of tools on a whole new set of problems.

BIBLIOGRAPHY

Bono, R. J. (1996). Dome: Geodesic dome design. Version 4.6. Available from: `http://www.senecass.com/software/dome46.tar.gz` [cited October 26, 2008].

Boudjemai, F., Enberg, P. B., and Postaire, J. G. (2003). Surface modeling by using self organizing maps of Kohonen. In *Systems, Man and Cybernetics, 2003. IEEE International Conference on*, volume 3, pages 2418–2423 vol.3.

Cho, S., Jang, M., and Reggia, J. A. (1996). Effects of varying parameters on properties of self-organizing feature maps. *Neural Processing Letters*, V4(1):53–59. Available from: `http://dx.doi.org/10.1007/BF00454846`.

Hagberg, A., Schult, D., and Swart, P. (2008). NetworkX: A python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. Los Alamos National Laboratory.

Harris, J. M., Hirst, J. L., and Mossinghoff, M. J. (2000). *Combinatorics and graph theory*. Springer, New York.

Ito, M., Miyoshi, T., and Masuyama, H. (2000). The characteristics of the torus self organizing map. In *Proceedings of 6th International Conference ON Soft Computing (IIZUKA2000)*, volume A-7-2, pages pp.239–244, Iizuka, Fukuoka, Japan. Available from: `http://mylab.ike.tottori-u.ac.jp/~mijosxi/act1997-/#Oral`.

Kohonen, T. (2001). *Self-Organizing Maps*. Springer, 3rd edition.

Kohonen, T., Hynninen, J., Kangas, J., and Laaksonen, J. (1996). Som_pak: The self-organizing map program package. Technical Report A31, Helsinki University of Technology, Laboratory of Computer and Information Science, FIN-02150 Espoo, Finland.

Li, X., Gasteiger, J., and Zupan, J. (1993). On the topology distortion in self-organizing feature maps. *Biological Cybernetics*, V70(2):189–198. Available from: `http://dx.doi.org/10.1007/BF00200832`.

Nishio, H., Altaf-Ul-Amin, M., Kurokawa, K., and Kanaya, S. (2006). Spherical SOM and arrangement of neurons using helix on sphere. *IPSJ Digital Courier*, 2:133–137.

Rakhmanov, E. A., Saff, E. B., and Zhou, Y. M. (1994). Minimal discrete energy on the sphere. *Mathematical Research Letters*, 1:647–662.

Ranka, R. J. (1997). Algorithm 772. stripack: Delaunay triangulation and voronoi diagram on the surface of a sphere. *ACM Transactions on Mathematical Software*, 23(3):416–434.

Rey, S. J. (2004). Spatial analysis of regional economic growth, inequality and change. In Goodchild, M. F. and Janelle, D. G., editors, *Spatially Integrated Social Science*, pages 280–299. Oxford University Press, Oxford.

Rey, S. J. and Janikas, M. V. (2006). Stars: Space-time analysis of regional systems. *Geographical Analysis*, 38(1):67–86. Available from: `http://www.`

blackwell-synergy.com/doi/abs/10.1111/j.0016-7363.2005.00675.x.

Ritter, H. (1999). Self-organizing maps on non-euclidean spaces. In Oja, E. & Kaski, S., editor, *Kohonen Maps*, pages 97–110. Elsevier, Amsterdam. Available from: citeseer.ist.psu.edu/ritter99selforganizing.html.

Sangole, A. and Knopf, G. K. (2003). Visualization of randomly ordered numeric data sets using spherical self-organizing feature maps. *Computers & Graphics*, 27(6):963–976. Available from: http://www.sciencedirect.com/science/article/B6TYG-49S7595-3/2/0913af2271d2f83a833452eea7ebddf9.

Skupin, A. and Agarwal, P. (2008). Introduction: What is a self-organizing map? In Agarwal, P. and Skupin, A., editors, *Self-Organising Maps: Applications in Geographic Information Science*. Wiley.

Vesanto, J. (2005). Som toolbox: implementation of the algorithm. Available from: http://www.cis.hut.fi/projects/somtoolbox/documentation/somalg.shtml [cited November 20, 2006].

Wasserman, S. and Faust, K. (1994). *Social network analysis : methods and applications*. Cambridge University Press, Cambridge; New York.

Wu, Y. and Takatsuka, M. (2005). Geodesic self-organizing map. In Erbacher, R. F., Roberts, J. C., Grohn, M. T., and Borner, K., editors, *Proc. SPIE Vol. 5669*, volume 5669 of *Visualization and Data Analysis 2005*, pages 21–30. SPIE.

Wu, Y. and Takatsuka, M. (2006). Spherical self-organizing map using efficient indexed geodesic data structure. *Neural Networks*, 19(6-7):900–910. Available from: http://www.sciencedirect.com/science/article/B6T08-4K719S6-2/2/f6d26607f815781a5bd3cfe5df5d6214.

APPENDIX
PySOM Source Code

PySOM Source Code

A.1 README

```
================================================================
PySOM: Python Self-Organizing Maps, README
----------------------------------------------------------------
AUTHOR(S): Charles R. Schmidt cschmidt@rohan.sdsu.edu
----------------------------------------------------------------

PySOM is an implementation of the Self-Organizing Map training
algorithm written in the Python Programming Language.  What makes
PySOM unique compared to other implementations is that the
network topology is not hard wired. Instead, we represent the
topology as a graph.  The NetworkX python library is used to
manage the graph.  Using a graph structure allows us to train
SOMs on a variety of different topologies with relative ease.
This document is intended to provide an overview of PySOM and
help users get started with training their SOMs.  In its current
form some basic knowledge of the Python Programming language is
required to use PySOM. I hope to provide a more robust graphical
user interface in the future, but for now PySOM can only be
accessed programmatically.  In the following sections you will
learn how to organize your data, set up your topology, set your
training parameters, and train your SOM.

-----------------
| Prerequisites |
-----------------
PySOM requires Python version 2.4 or later, available from
http://python.org

In addition the following Python Libraries are required:
Numpy (1.0.4) available from http://numpy.org/
NetworkX (0.35.1) available from http://networkx.lanl.gov/
Version numbers used for development are provided, but future
version are expected to work.

----------------
| Installation |
----------------
Download the latest version of PySOM from
```

http://code.google.com/p/pysom-thesis/downloads/list

```
________
| Data |
________
```

The observation file should match the following format, where the first row indicates the number dimension in your input data set. Each subsequent row contains the values or a particular observation separated by spaces. Addition columns can exist in your observation file, e.g. for meta data. But PySOM will only read the number of columns specified on the first line. Your values should scale from 0 to 1. See the following extract from a real input file:

```
3
0.847 0.508 0.573 1
0.597 0.417 0.093 6
0.436 0.578 0.464 0
0.454 0.407 0.092 6
0.906 0.438 0.501 1
0.482 0.025 0.475 5
```

We create an instance of the observation file manager with the following code:

```
>>> from pysom import data
>>> f = data.ObsFile('/path/to/file.txt')
```

```
____________
| Topology |
____________
```

In the current version, PySOM will create both hexagonal and rectangular graph structures. Any valid undirected NetworkX can be used as the topology for your SOM. The nodes of the graph will be treated as neurons. To create a hexagon graph use the following python code:

```
>>> from pysom import topology
>>> rows = 3
>>> cols = 3
>>> G = topology.hexGraph(rows,cols)
```

For a rectangular graph change the last line to:

```
>>> G = topology.rookGraph(rows,cols)
```

```
______________________
| Training Parameters |
```

```
______________________
Before setting the training parameters you must create an
instance of the SOM.  This can be done with the following code:
>>> from pysom import som
>>> S = som.GraphyTopology(G)
where G is the graph we created above.

Once the SOM instance has been created we need to set the
following training parameters (comments marked by '#' explain
each parameter):
>>> #The number of dimensions in training data
>>> S.Dims = 3

>>> #The percent of the network to include in the first training step
>>> S.maxN = 0.5

>>> #The number training steps to perform
>>> S.tSteps = 10000

>>> #The initial learning rate
>>> S.alpha0 = 0.04


___________
| Training |
___________
With the training parameters set we can now randomly initialize
the SOM and start the training with the following code:
>>> S.randInit()
>>> # Save the initial State (optional)
>>> S.save('output/', 'initial_state')
>>> S.run(f)
where S was created above and f was the observation file manager
we also created above.

The SOM can be saved at any point with its '.save' method. This
method takes two parameters, the first is a path, the second is
the save as name. We'll save the trained SOM with the following
code:
>>> S.save('output/','phase1')

It is often desirable to train the SOM in two phases, to do this
simply adjust the training parameters, reset the file and restart
the training. As demonstrated with the following code:
>>> S.maxN = 0.333
```

```
>>> S.tSteps = 1000000
>>> S.tSteps = 1000
>>> S.alpha0 = 0.03
>>> f.reset()
>>> S.run()
>>> S.save('output/','phase2')

=================================================================
```

A.2 SOM.PY

```
"""
=================================================================
Python Self-Organizing Maps
-----------------------------------------------------------------
AUTHOR(S):      Charles R. Schmidt cschmidt@rohan.sdsu.edu
-----------------------------------------------------------------
* Copyright (c) 2006-2008, Charles R. Schmidt
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*     * Redistributions of source code must retain the above copyright
*       notice, this list of conditions and the following disclaimer.
*     * Redistributions in binary form must reproduce the above
*       copyright notice, this list of conditions and the following
*       disclaimer in the documentation and/or other materials
*       provided with the distribution.
*     * Neither the name of the San Diego State University nor the
*       names of its contributors may be used to endorse or promote
*       products derived from this software without specific prior
*       written permission.
*
* THIS SOFTWARE IS PROVIDED BY Charles R. Schmidt ''AS IS'' AND ANY
* EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL Charles R. Schmidt BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
* BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
* LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
```

```
* NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
* SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
============================================================================
"""
# Standard Libraries
import random,math,time,sys,os
from math import acos,sqrt,pi,degrees,sin,cos,asin
import pickle #provides object serialization

# External Libraries
# http://numpy.scipy.org/
from numpy import array,empty,take,put,zeros
import numpy as N
# https://networkx.lanl.gov/
import networkx as NX

# Local Libraries (part of pySom)
import networkFunctions as nf
from data import ObsFile

class som:
    ''' Base class for the Self-Organizing Map,
        Each topology will inherit from this class.
        A template is provided in 'Topology'
    '''
    def __init__(self):
        '''
        These initial training parameters should be
        set before training.
        '''
        # int, the number of dimension in the input-space
        self.Dims = 0
        # int, optionally the number neurons in the X dimension
        self.X = 0
        # int, optionally the number neurons in the Y dimension
        self.Y = 0
        # int, total number of neurons
        self.Size = 0
        # str, name of topology type
        self.Type = 'none'
        # int, total number of training steps.
        self.tSteps = 0
        # float, initial neighborhood radius, expressed as percentage.
        self.maxN = 0.0
        # float, initial learning rate
```

```
        self.alpha0 = 0.0

        # reference vectors are stored here.
        self.nodes = []
        # mapping of observations to neurons is stored here.
        self.daMap = {}

    def load(self,path='',name=None):
        ''' This function loads a saved SOM from disk into memory.
            pySom uses several files to represent the SOM,
            .cod is the Codebook file which represents the
                reference vectors
            .map represents the mapping of the observations
                onto the trained SOM

            If the path and name are omitted they will be guessed
            based on the SOM's parameters
        '''
        if not name:
            name = '%ds_%dd_%dr_%fa'%(self.Size,self.Dims,self.tSteps,...⇒
...self.alpha0)
        codname = path+name+'.cod'
        mapname = path+name+'.map'
        if os.path.exists(mapname):
            try:
                f = open(mapname,'r')
                self.daMap = pickle.load(f)
                f.close()
            except:
                self.daMap = {}
        dataf = open(codname,'r')
        header = dataf.next()
        Dims,Type,x,y,nType= header.split()
        self.Dims = int(Dims)
        self.Type = Type
        self.X, self.Y = int(x),int(y)
        self.Size = self.X*self.Y
        self.nodes = array([[0.0 for i in xrange(self.Dims)] for j in ...⇒
...xrange(self.Size)])
        for i in xrange(self.Size):
            data = dataf.next()
            data = data.split()
            data = map(float,data)
            self.nodes[i] = data
```

```
    def save(self,path='',name=None):
        ''' This function saves a SOM to disk.
            pySom uses several files to represent the SOM,
            .cod is the Codebook file which represents the
                reference vectors
            .map represents the mapping of the observations
                onto the trained SOM

            If the path and name are omitted they will be guessed
            based on the SOM's parameters
        '''
        if self.X == 0 or self.Y == 0:
            self.X = self.Size
            self.Y = 1
        if not name:
            name = '%ds_%dd_%dr_%fa'%(self.Size,self.Dims,self.tSteps,... ⇒
...self.alpha0)
        codname = path+name+'.cod'
        mapname = path+name+'.map'
        if self.daMap:
            mapfile = open(mapname,'w')
            pickle.dump(self.daMap,mapfile)
            mapfile.close()
        outf = open(codname,'w')
        outf.write("%d %s %d %d gaussian\n"%(self.Dims,self.Type,self.... ⇒
...X,self.Y))
        for i in xrange(self.Size):
            outf.write(' '.join(str(self.nodes[i].tolist())[1:-1].spli... ⇒
...t(', '))+'\n')
        outf.close()

    def randInit(self):
        ''' This function initializes the reference vectors with
            random values in the range of 0 to 1.  If your data has
            not standardized within this range, this function should
            be overwritten.

            * Ideally this function would analyze the input data and
              scale the randomization accordingly.  SOM_PAK does
              something along these lines, their code might help find
              a good solution for this.
        '''
        self.nodes = array([[random.random() for j in xrange(self.Dims... ⇒
...)] for i in xrange(self.Size)])
```

```
def findBMU(self,ind,v,ReturnDist = False):
    ''' This function returns the ID of the reference vector that
        is the closest (in Euclidean distance) to input vector v.
        Optionally this function will also return that distance
        between the two.  Overwrite this function if you need
        something other than Euclidean dist.

        the 'ind' parameter is unused and should be set to None.

        * As written this function does not handle missing values,
          a previous version did, but it was painfully slow. One
          option would be to remove the distance function and pick
          the appropriate dist function depending on the values
          present.
        * An external distance function could also be used here,
          however the extra function call will effect
          performance.
    '''
    d = ((self.nodes-v)**2).sum(1)
    minI = d.argmin()
    if ReturnDist:
        minD = d[minI]
        return minI,minD
    else:
        return minI

def alpha(self,t):
    ''' Returns the learning rate (alpha) as a function of time t.
    '''
    r = self.alpha0 * (1 - (t/float(self.tSteps)))
    if r < 0: r = 0
    return r
def hci(self, t, dist):
    ''' This kernel function adjust the magnitude with which the
        observation affects the reference vectors.
    '''
    sigma = self.kernelWidth(t)
    a = self.alpha(t)
    top = dist**2
    bottom = (2*(float(sigma)**2))
    return a * math.exp(-top/bottom)

def merge(self,t,ind,v):
    ''' This function adjusts the actual reference vectors at time
        't' based on observation vector 'v'.
```

```
        '''
        bmu = self.findBMU(None,v)
        sigma = self.kernelWidth(t)
        results = self.neighborhood( bmu , sigma )
        alteredNodes = [(results[i],self.hci(t,self.odist(i))) for i i...  ⇒
...n xrange(len(results))]
        for nodeID,hc in alteredNodes:
            if len(ind) == self.Dims:
                part = self.nodes[nodeID]
                delta = hc*(v-part)
                self.nodes[nodeID] = part+delta
            else:
                part = take(self.nodes[nodeID],ind)
                delta = hc*(v-part)
                put(self.nodes[nodeID],ind,part+delta)

    def run(self,obsf):
        ''' Call this function with your observation file are a
            parameter.

            obsf must be an instance ObsFile as provided by data.py

            All training parameters should be set before calling this
            function.
        '''
        self.neighborhoodCache = {}
        print "####################################"
        print "###        Configuration         ###"
        print "####################################"
        print "  N = %d, maxN = %f"%(self.Size,self.maxN)
        print "  Total runs: %d"%self.tSteps
        print "  initial learning rate = %f"%self.alpha0
        print "  Obs File:%s"%obsf.filename
        print "####################################"
        print "###          End Config          ###"
        print "####################################"
        print ""
        print "Running..."
        t1 = time.time()
        T = self.tSteps
        for t in xrange(self.tSteps):
            id,ind,v = obsf.stream()
            self.merge(t,ind,v)
            sys.stdout.write("\r%.2f%%"%(100*float(t)/T))
            sys.stdout.flush()
```

```
        print "\nRun completed in %f seconds"%(time.time()-t1)

    def map(self,obsf):
        ''' This function maps the observation back onto the trained
            SOM.  The QError is returned.
        '''
        qerror = 0
        counter = 0
        daMap = {} # keys are node ID's, values are lists of observati...  ⇒
...on ID's.
        for id,dimIds,obs in obsf:
            bm,err = self.findBMU(dimIds,obs,ReturnDist=True)
            qerror += err
            if not bm in daMap:
                daMap[bm] = []
            daMap[bm].append(id)
            sys.stdout.write(".%d,%f.\r"%(counter,err))
            sys.stdout.flush()
            counter += 1
        print ""
        qerror = qerror/counter
        self.daMap = daMap
        return qerror

class GraphTopology(som):
    ''' GraphTopology extends "som" and provides a functioning
        implementation of the Self-Organizing Map training algorithm.
        The topology is based on a graph structure with is provided
        using the NetworkX graph library.
    '''
    def __init__(self,G=None,Type='Graph'):
        ''' Special initialization for this topology,
            also calls som.__init__
        '''
        som.__init__(self)
        self.Type = Type
        if G:
            self.G = G

            # Number of nodes (neurons) in the network.
            self.Size = G.order()
            # if findWidth is not given a seed it will brute force the
            # total network width, this could take a long time. For
            # the spherical network, one of the polls should yield the
            # correct width. Or possibly the node with lowest degree.
```

```
            # The Width messures the largest distance between any two
            # nodes in the network.

            self.Width = nf.findWidth(G,G.nodes()[-1])

    def save(self,path,name):
        ''' in addition to saving this codebook, we also need to save
            the graph
        '''
        som.save(self,path,name)
        if not name:
            name = '%ds_%dd_%dr_%fa'%(self.Size,self.Dims,self.tSteps,... ⇒
...self.alpha0)
        graphFile = path+name+'.graph'
        f = open(graphFile,'wb')
        pickle.dump(self.G,f) #serialize the graph to a textfile.
        f.close()
    def load(self,path,name):
        ''' loads the graph and other information from disk '''
        som.load(self,path,name)
        if not name:
            name = '%ds_%dd_%dr_%fa'%(self.Size,self.Dims,self.tSteps,... ⇒
...self.alpha0)
        graphFile = path+name+'.graph'
        if os.path.exists(graphFile):
            f = open(graphFile,'rb')
            self.G = G = pickle.load(f)
            self.Size = G.order()
            self.Width = nf.findWidth(G,G.nodes()[-1])
            f.close()
    def kernelWidth(self,t):
        ''' kernelWidth returns the width of the neighborhood in terms
            of order.
        '''
        r = round((self.Width*self.maxN) * (1 - (t/float(self.tSteps)))... ⇒
...))
        r = int(r)
        if r == 0: r = 1
        return r
    def neighborhood(self,bmu,kernelWidth):
        '''
        This function returns a dictionary containing the neighbors of
        bmu as keys and their dist (as an order) as values.
        '''
        return nf.neighborhood(self.G,bmu,kernelWidth)
```

```
    def merge(self,t,ind,v):
        ''' This function adjusts the actual reference vectors at time
            't' based on observation vector 'v'.
        '''
        bmu = self.findBMU(None,v)
        a = self.alpha(t)
        sigma = self.kernelWidth(t)
        results = self.neighborhood( bmu , sigma )
        sigma = float(sigma)

        # hci has been internalized in this next line,
        # to speed up processing
        alteredNodes = [(node,(v-self.nodes[node])*(a*math.exp(-(odist...  ⇒
...*odist)/(2*sigma*sigma)))) for node,odist in results.iteritems()]
        for nodeID,node in alteredNodes:
            self.nodes[nodeID] += node

#class Topology(som):
#    """ Template class for topology.
#        Copy this class to create a new topology for som.
#    """
#    def __init__(self):
#        som.__init__(self)
#    def save(self,path,name):
#        som.save(self,path,name)
#    def load(self,path,name):
#        som.load(self,path,name)
#    def randInit(self):
#        som.randInit(self)
#    def kernelWidth(self,t):
#        ''' kernelWidth returns the width of the neighborhood in terms
#            of order.
#        '''
#        pass
#    def odist(n):
#        """
#        n is the nth neuron in the in neighborhood, returns order
#        example the 3rd neuron in the set is 1 order from the 0th.
#        """
#        pass
#    def neighborhood(self,bmu,kernelWidth):
#        '''
#        This function returns a dictionary containing the neighbors of
#        bmu as keys and their dist (as an order) as values.
#        '''
```

```
#       pass
```

A.3 NETWORKFUNCTIONS.PY

```
"""
============================================================================
Python Self-Organizing Maps -- Network Functions
----------------------------------------------------------------------------
AUTHOR(S):      Charles R. Schmidt cschmidt@rohan.sdsu.edu
============================================================================
"""

import networkx as NX

def neighborhood(G,n,o):
    """ Returns the neighbors of node n in graph G out to order o,
        Returned as dictionary where the keys are node ids and the
        values are the order of that node,
    """
    base = G[n]
    neighbors = {}
    neighbors[n] = 0
    newNodes = set(neighbors.keys())
    for i in range(1,o+1):
        #for node in neighbors.keys():
        nodes = newNodes.copy()
        newNodes = set()
        for node in nodes:
            branch = G[node]
            for node in branch:
                if node not in neighbors:
                    newNodes.add(node)
                    neighbors[node]=i
    return neighbors

def findWidth(G,seed=None,returnWidths=False):
    """ Returns the maximum width of graph G
        If given a seed will only search from that node.
    """
    widths = []
    if not seed == None:
        nodes = [seed]
    else:
        nodes = G.nodes()
    for node in nodes:
```

```
            o = 0
            maxW = 0
            W = len(neighborhood(G,node,o))
            while 1:
                o += 1
                W2 = len(neighborhood(G,node,o))
                if W2 > W:
                    W = W2
                    maxW = o
                else:
                    break
            widths.append(maxW)
        if returnWidths: return widths
        else: return max(widths)

if __name__=='__main__':
    G = NX.Graph()

    G.add_edge(('1','2'))
    G.add_edge(('2','3'))
    G.add_edge(('3','4'))
    G.add_edge(('4','5'))
    G.add_edge(('5','6'))
    G.add_edge(('6','7'))
    G.add_edge(('7','8'))
    G.add_edge(('8','9'))
    G.add_edge(('9','1'))

    print findWidth(G)
    print neighborhood(G,'1',4)
```

A.4 DATA.PY

```
"""
=====================================================================
Python Self-Organizing Maps -- Data Tools
---------------------------------------------------------------------
AUTHOR(S):      Charles R. Schmidt cschmidt@rohan.sdsu.edu
=====================================================================
"""

from numpy import array,empty

class ObsFile:
    ''' ObsFile provides an interface to your Observation Data File.
```

```
        The File should be formated as follows:
        The first line should contain only the number of dimension in ...⇒
...the data set.
        Each subsequent line should contain one observation, each valu...⇒
...e should be separated
        a space.

        eg:
        3
        1.0 0.9 0.8
        0.6 0.0 0.2
        0.2 0.5 1.0
        0.0 1.0 0.0

    '''
    def __init__(self,filename,fileType = 'complete'):
        ''' Opens the file '''
        self.filename = filename
        self.fileObj = open(filename,'r')
        self.fileType = fileType
        self.reset()
    def __iter__(self):
        ''' Create an iterator '''
        return self
    def listolists(self,comments=False):
        ''' Returns the entire data file as a list of lists '''
        self.fileObj.seek(0)
        lines = self.fileObj.readlines()
        dims = lines.pop(0)
        dims = int(dims)
        lines = [line.split() for line in lines]
        if not comments:
            lines = [line[:dims] for line in lines]
            lines = [array(map(float,line),'float') for line in lines]
        else:
            lines = [line[dims:] for line in lines]
        return lines
    def Snext(self):
        ''' Returns the next line of a "sparse" data file.
            A sparse data file is one that contains missing values
        '''
        line = self.fileObj.next()
        id,line = line.split(':')
        line = line.split(',')
        Num = len(line)/2
```

```
        indices = empty(Num,'int16')
        values = empty(Num,'float')
        c = 0
        for n in xrange(0,Num*2,2):
            indices[c] = int(line[n])-1
            values[c] = float(line[n+1])
            c += 1
        return id,indices,values
    def Cnext(self):
        ''' Return the next line of a "complete" data file.
            A complete data file is one without any missing values.
        '''
        line = self.fileObj.next()
        line = line.split()
        id = self.nextLine
        for n in xrange(0,self.Dims):
            self.values[n] = float(line[n])
        self.nextLine+=1
        return id,self.indices,self.values
    def reset(self):
        ''' Resets the file such that the next line returned is the fi... ⇒
...rst line '''
        self.fileObj.seek(0)
        self.nextLine = 0 #Zero Base
        if self.fileType == 'complete':
            self.next = self.Cnext
            self.Dims = int(self.fileObj.next())
            self.indices = array(range(self.Dims),'int16')
            self.values = empty(self.Dims,'float')
        elif self.fileType == 'sparse':
            self.next = self.Snext
        else:
            raise "fileTypeError"
    def stream(self):
        ''' Loops of the records, always returns a line, if the end of... ⇒
... file is reached,
            the file pointer is reset back to that start.
        '''
        try:
            return self.next()
        except:
            self.reset()
            return self.next()
    def close(self):
        ''' Close the file '''
```

```
        self.fileObj.close()
```

A.5 TOPOLOGY.PY

```
"""
============================================================================
Python Self-Organizing Maps -- Topology
----------------------------------------------------------------------------
AUTHOR(S):      Charles R. Schmidt cschmidt@rohan.sdsu.edu
============================================================================
"""

import networkx
from utils.grid2rook import grid2Rook

def rookGraph(rows, cols):
    ''' Create a rectangular graph topology '''
    rook = grid2Rook(rows,cols,binary=1)
    G = networkx.Graph()
    for node in rook:
        for neighbor in rook[node][1]:
            G.add_edge((node,neighbor))
    return G
def hexGraph(rows, cols):
    ''' Create a hexagonal graph topology '''
    x = cols
    y = rows
    grid = []
    c = 0
    for i in range(y):
        row = []
        for j in range(x):
            row.append(c)
            c += 1
        grid.append(row)
    g = networkx.Graph()
    for i,row in enumerate(grid):
        # L * * *
        # R   * * *
        # L * * *
        for j,ptID in enumerate(row):
            if i%2: # R, can never be first row.
                g.add_edge(ptID,grid[i-1][j]) #my upper neighbor
                if j < len(row) - 1: #not last column
                    g.add_edge(ptID,grid[i][j+1]) #my right neighbor
```

```
                    g.add_edge(ptID,grid[i-1][j+1]) #my upper right ne... ⇒
...ighbor
                if i < len(grid) -1: #not last row
                    g.add_edge(ptID,grid[i+1][j]) #my lower neighbor
                    if j < len(row) - 1: #not last column
                        g.add_edge(ptID,grid[i+1][j+1]) #my lower righ... ⇒
...t neighbor
            else: # L
                if j < len(row)-1:
                    g.add_edge(ptID,grid[i][j+1]) #my right neighbor
    return g
```

ABSTRACT OF THE THESIS

Effects of Irregular Topology in Spherical Self-Organizing Maps

by
Charles R. Schmidt
Master of Science in Geography
San Diego State University, 2008

The traditional Self-Organizing Map (SOM) uses a rectangular or hexagonal network topology. These topologies create a well-known problem in SOM called the boundary or edge effect. Toroidal and spherical topologies have been widely suggested as a solution to the problem. However, there exist only five arrangements on the sphere which are completely regular; these are the five platonic solids (Ritter, 1999; Harris et al., 2000). Any other arrangement of neurons on the surface of the sphere will result in an irregular topology, as not all neurons will have the same number of neighbors. The classic method for minimizing this irregularity is to generate the spherical lattice by tessellating the sides of the icosahedron (Nishio et al., 2006). While this method will always result in a highly regular spherical topology, the main drawback is that the number of neurons in the network (the network size), N, grows exponentially as tessellations are applied. That results in only very coarse control over network size. We explore the effect of different topologies on properties of SOM. We suggest several diagnostics for measuring topology-induced errors in SOM and use these in a comparison of four different topologies. The results show that SOM is less sensitive to localized irregularities in the network structure than the literature may otherwise suggest. Further, the results support the use of spherical topologies as a solution to the boundary problem in traditional SOM.