GIS Based Topological Modeling for Infrastructure Systems

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Sriharsha Vankadara
Bachelor of Engineering
Birla Institute of Technology and Science, 2005

Director: Michael J. Casey, Assistant Professor
Department of Civil, Environmental and Infrastructure Engineering

Spring Semester 2010
George Mason University
Fairfax, VA

DEDICATION

This thesis would be incomplete without a mention of the support given by my parents, Mr. Sreedhar and Mrs. Rama Devi and my brother, Mr. Sriram.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Michael Casey, my advisor for his constant support and guidance that made my stay at Mason memorable. I would like to thank my committee members Dr. Houck and Dr. Venigalla for their time. I would also like to thank my family and friends who were there in times of need.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

GIS BASED TOPOLOGICAL MODELING FOR INFRASTRUCTURE SYSTEMS

Sriharsha Vankadara, M.S.

George Mason University, 2010

Thesis Director: Dr. Michael J. Casey

Today's society greatly depends on the operations of complex infrastructure networks such as transportation, utilities and telecommunication. While traditional modeling tools have provided an insight into the theoretical behavior of infrastructure networks, they lacked the ability to elucidate the spatial organization between multiple networks. Accurate modeling of infrastructure systems depends on integration of dependencies, spatial reliance and properties of each system. The purpose of this research is to develop a toolkit to model the spatial relationships between randomly generated infrastructure facilities using a Geographical Information Systems (GIS) based topological approach.

Topology describes the geometric associations between infrastructure elements represented as point, line and polygon features. The objective is to generate realistic topological networks in the absence of spatially complete datasets that represent infrastructure networks. The toolkit is developed in Python making use of custom GIS

libraries and comprises random topology generators for star, trunk and mesh shaped networks. The efficiency of the tools is tested by their ability to generate a water distribution topology from minimal spatial data to compare the created network with that of a reference distribution network.

CHAPTER 1

INTRODUCTION

1.1 Background

Infrastructure generally refers to the underlying framework or features that form the basis of a system or organization. From a civil engineering point of view, infrastructure systems are physical facilities and services that are necessary for a society or economy to function. These facilities typically include buildings, roads, sewers, dams, telecommunications and other physical structures that are essential to the existence and development of a society.

A large number of infrastructure works have their functions dependent upon the existence and operations of other systems. For example, roads provide the accessibility required for the transportation of raw materials to factories as well as distribution of finished goods to markets. Besides functional dependency, there is also a spatial reliance between disparate infrastructure systems. The dependencies among facilities can be characterized as logical relationships or physical entities on the ground, capable of transporting resources between elements of an infrastructure system. Accurate modeling of infrastructure systems depends on integration of dependency, spatial reliance, and engineering properties of each system. This task is extraordinarily difficult in practice and consequently results in models that are inaccurate, incomplete, or unrealistic.

Methods are needed to provide a flexible and accurate means of modeling infrastructure, specifically geospatial distribution and dependency.

This thesis focuses on modeling the spatial relationships between these systems using a Geographic Information Systems (GIS) based approach. GIS is widely used for spatial modeling as it provides powerful tools for data management, visualization, presentation and analysis. In a GIS based approach, individual elements of a large network of infrastructure system are represented as features with spatial attributes defined in co-ordinate pairs. These elements can be discrete, taking shapes of points (nodes) linked together to form discrete line features (arcs), or to form closed boundaries enclosing an area (polygon). A spatial relationship is one that describes the association between shapes or locations of features.

A topological model best reflects the geography of the real world, by providing a mathematical approach to describe the spatial relationships between geographic data types based on the principles of order, connectivity and adjacency. It helps maintain data integrity and quality by enforcing rules that model ways in which points, lines and polygons share geometry. The objective is to generate a topology consistent with a defined set of rules to model the spatial interactions between similar and dissimilar facilities represented as standard geographical features. Facilities that are geometrically polygonal in a geographic context are assumed to be point features in the topological models. Three distinct models are implemented as part of the work. A random feature chosen to symbolize the hub of a network is used to generate a star topology. A second model illustrating a trunk topology with nodes connecting to a main edge feature is

described. The third topological network represents nodes and edges organized into a mesh structure.

1.2 Problem

A thorough understanding of spatial relationships between elements of infrastructure can be of great assistance in crucial tasks such as planning, design, operation and maintenance of individual facilities and the system as a whole. Modeling these relationships, helps gaining better insight into methods in which entities behave and perform relative to each other in a geospatial context, thus providing an opportunity to generate near to accurate models that represent actual infrastructure networks on ground.

The foundation of any research problem is data. No amount or depth of data analysis can substitute for the lack of sufficient data. Of utmost importance is data acquisition, upon which subsequent modeling and analysis techniques depend which alternatively governs the accuracy and reliability of results. However, data is not always available. This essentially depends on the type of data being sought such as sensitive information pertaining to critical infrastructure. In such situations, random data can be substituted to simulate actual conditions. This thesis addresses the problem of modeling spatial associations between arbitrarily generated infrastructure entities and generating network topologies which under certain given circumstances can be comparable and used alternatively in place of real network models representing actual infrastructure systems.

1.3 Objectives

The objective of the research is twofold. The primary objective is the development of a toolkit comprising of tools to model spatial associations between infrastructure facilities by generating topological models from spatially random data using GIS principles and components. The tools are to be built with the ability to generate nodes representing entities and edges connecting the nodes symbolizing connectivity between facilities.

The second objective is to apply the programming techniques implemented in the tools to generate topology for a water distribution network comprising of distribution mains, service lines and land parcels from inadequate data such as the lack of location information pertaining to service laterals and by imposing connectivity restrictions between elements of the network. The resulting model is to be compared with a reference water distribution network available for the same geographic location and one that corresponds to an actual network implemented on ground to assess the reliability and accuracy of the programmatically generated topology.

1.4 Approach

A programmatic approach has been employed to demonstrate spatial interaction between features by generating topologies modeled in the form of star, trunk and mesh networks. A toolkit comprising three distinct tools one for each of the networks is developed using a popular and widely adopted scripting language. The tools rely on spatially random data as input, from which nodes representing physical or virtual

infrastructure sources producing or consuming resources, and edges representing conduits for resource flow are generated as geographic features.

The first tool simulates situation in which facilities represented as nodes or point features connect to a hub or central facility. The second generator tool models how facilities associate themselves to a trunk or main line running through them. The tool ensures that certain topological rules pertaining to non-intersection of relations represented by edges and distance constraints are met in the process. The last tool ensures that connectivity exists between a node and all other nodes in the network.

The methodology implemented in creating the trunk topology generation tool is used to program another script tool specifically for modeling a water distribution network. The tool makes use of land parcel data as input to generate a network model conforming to topological rules with parcels represented as nodes and water distribution utilities symbolized as edge features. The model thus generated is compared with a reference model of the distribution system to understand the effects of connectivity restraints on topologies and test for accuracy, similarities and feasibility between the models.

The modified trunk topology tool was applied over land parcel and utility main data, to generate service laterals that connect the distribution mains to service endpoints within parcel boundaries by implementing rules that prevent laterals from crossing over into adjacent parcels. For real estate that exists along street corners and has two boundaries available for placement of service laterals, another rule was implemented to determine the nearest distribution main to which the respective lateral would connect.

The results expected were feature containers of nodes and links which together along with the input data constitute a topological model compliant with proximity and adjacency rules that were imposed programmatically and bears close resemblance to the model chosen as reference. The model as a whole is compared to the reference to check for organization of the layout, similarities between arrangement of service laterals and accuracy.

CHAPTER 2

PRELIMINARY STUDY

Infrastructure modeling is widely researched to understand the behavior and performance of individual facilities as well as interdependent systems under certain modeling conditions specific to a study. The following section gives an overview of a modeling technique to study the impact of one infrastructure facility on dependent infrastructure. The next section describes methods and approaches for topology generation followed by GIS tools that are available and often used in infrastructure modeling. Modeling softwares that implement GIS technology for transportation and hydrological applications are presented in the final section.

2.1 Infrastructure models

Critical infrastructure is one of the most researched topics in the field of infrastructure modeling. The ramifications that critical infrastructure failure can have on the societal and economic conditions of a nation make it a commonly modeled problem. Critical infrastructure as defined by the U.S. Patriot Act comprises

> "systems and assets, whether physical or virtual, so vital to the United States that the incapacity or destruction of such systems and assets would have a debilitating

impact on security, national economic security, national public health or safety, or any combination of those matters" [1].

Innovative modeling approaches are necessary to identify and understand vulnerabilities within individual infrastructure components, failure of which may have a devastating effect on connected infrastructures. The research under review identifies that the problem in modeling cross-infrastructure effects depends on integration and behavior of individual critical infrastructure elements.

Tolone et. al. [2] adopted an approach that involves utilizing an intelligent agent-based methodology that develops integration awareness external to the infrastructure components. Agent-based integration uses knowledge of the context represented in the form of facts and rules that govern integration between individual infrastructures. A software agent is an autonomous program, or program component, that is situated within, aware of, and acts upon its environment in pursuit of its own objectives so as to affect its future environment. The modeling and simulation environment is designed in ways that allow the end user to execute simulations within a GIS context. Simulations are initiated by disabling infrastructure features and consequently viewing the impacts on connected elements through GIS visualization. The inter-infrastructure simulations are managed by collection of software agents which observe changes within infrastructure using knowledge of interdependencies, communicate with one another and based upon mutual interpretation affect changes across the concerned infrastructures. The agents are capable of affecting two types of state changes. First, having observed a state change in infrastructure, agents are capable of discerning impacts using knowledge available and

consequently affect changes in state within and across infrastructures. Second, agents can utilize GIS network analysis to reason and affect state changes.

The results of the simulation are renderings of subsequent state changes across infrastructures due to impacts caused by disabling certain features within an infrastructure. The results are graphical solutions viewed in a GIS display.

2.2 Topology generation

Infrastructure facilities represented as networks provide a useful framework for the representation and modeling of many physical, biological and social systems. Gastner and Newman in their study of spatial networks focused on the effects that geography has on the efficiency of networks [3]. The networks chosen by the authors for study were specifically distributed such as gas pipelines, sewage systems, and rail or air routes. The authors assumed that these networks have a root node that acts as a source or sink of the commodity distributed, for example, a sewage treatment plant [3].

The distribution networks were considered to have two properties that impact efficiency. The first required the path between a vertex and the root to be relatively short. This meant that the sum of lengths of the edges along the shortest route between the vertex and root is not much greater than the Euclidean distance between the same two vertices. The second property required that the total length of all edges in the network is less so that the network is economical to build. To evaluate the efficiency of networks in terms of path lengths and total length of edges, two topological models that are each optimal respective to one of these criteria are used to compare the measurements. A star

model is representative of the shortest path to root node with every vertex connecting to the root by a straight edge. On the other hand, a minimum spanning tree (MST) is the optimal network representing a case of minimum total edge length. The comparison with star graph is achieved by computing the network's *route factor* which is the mean ratio of distance from a vertex to the root and the Euclidean straight line distance between the vertices computed over all non-root vertices. The route factor is given by the equation:

$$q = 1 / n * (\sum_{i=1}^{n} l_{i0} / d_{i0}) \quad (1)$$

where l_{i0} is the distance between a vertex and root and d_{i0} is the straight line distance. The star network is optimal for having the lowest possible router factor of 1.

network	number of vertices	route factor			edge length (km)		
		actual	MST	star	actual	MST	star
sewer system	23922	1.59	2.93	1.00	498	421	102998
gas (WA)	226	1.13	1.82	1.00	5578	4374	245034
gas (IL)	490	1.48	2.42	1.00	6547	4009	59595
rail	126	1.14	1.61	1.00	559	499	3272

Figure 2.1: Route factors and total edge lengths [3]

The route factors for four networks under study along with their calculated total edge lengths are shown in Figure 2.1. From the route factors, it is evident that the networks are efficient with values close to 1. The edge lengths measured for the MST are also found to be reasonably close in comparison with the values of the actual networks. The remaining columns indicate that although the MST is optimal in terms of total edge

length, its values for route factor deviate to a large extent from those of actual networks and the reverse is true for the star shaped model. Due to this significant variation, the authors conclude that neither of these actual networks can function as a viable solution to the problem of generating an efficient distribution network [3]. Real world networks are capable of adjusting themselves in such a way that they concurrently possess benefits of both star and the minimum spanning tree.

Wang and Provan classify topological model generators into two categories: *explanatory* models, which capture topology growth that is based on specifics of the domain of the resultant model and *descriptive* models which are exclusively concerned with random topology generation [4]. The authors propose two possible methods as part of the explanatory models. The first being *spatial preferential attachment* which combines spatial constraints to reduce cost of generating connections and preferential attachment in which network growth primarily occurs around existing networks [4,5]. The second method that the authors suggest is based on principles of optimization by minimizing a function that is cumulative of the number of legs or edges a resource has to be transmitted and the cost of constructing those edges [4]. The concept of random topology generation, upon which the tools were built for building networks, is referred by the authors as the *generalized random graph (GRG) model* that is independent of any specifics of the network being generated [4]. To better represent real world networks, the authors suggest extending this random model by including a degree sequence for nodes which enables selecting at uniform a random graph from all possible graphs for the same degree sequence [4, 6].

The topology generation techniques described primarily deal with the objectives of generating efficient topologies that represent real world networks and in the process make use of domain-specific data or tend to improve upon random generation methodology. The topology generation toolkit is the programmatic solution to developing distinct network topologies under conditions of lack of spatial data and subsequently capable of being modified as per the requirements of specific modeling purposes.

2.3 GIS tools

GIS can be viewed as an integrated package of hardware and software components, with powerful tools for data management, complex spatial analyses and visualization. A GIS based approach exposes powerful tools for managing data, visualization and analysis. GIS based modeling can be most valuable when the region of study covers a large geographic extent; the spatial and non-spatial attributes of data have a significant role in the model and when spatial analysis and its results play a key role in the modeling approach.

The number of tools and depth in support for spatial analyses differ significantly between several GIS software available in the market. While there has been a tremendous rise in use of open source GIS tools in recent times, the GIS industry widely relies on commercial, off-the-shelf (COTS) software such as ArcGIS. Almost all commercial GIS software offer complex analysis and modeling capabilities in areas such as vector and raster analysis, cartographic modeling, topological modeling, network analysis,

geostatistical estimation, hydrological modeling and visualization of two and three dimensional data.

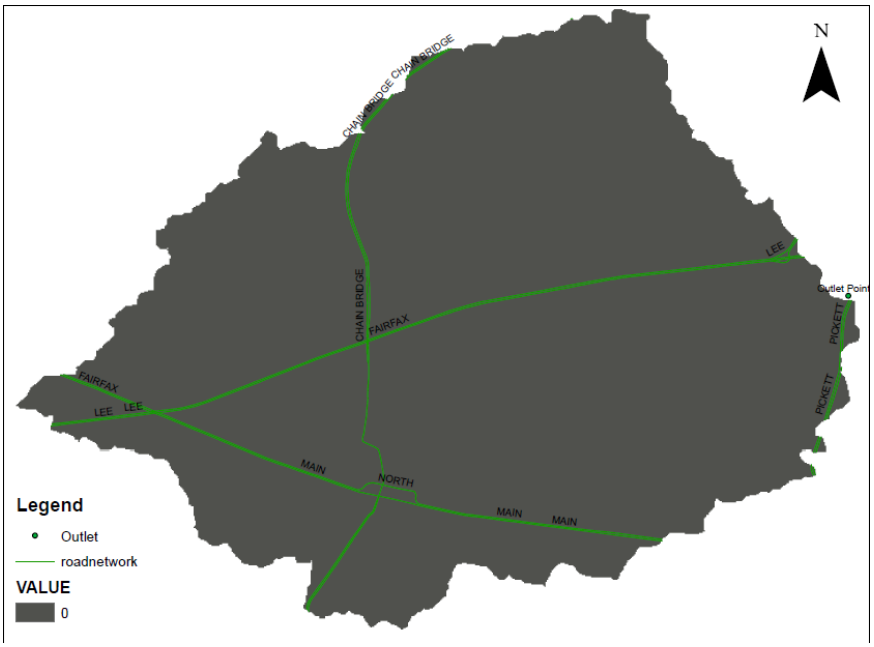

Figure 2.2: Hydrologic modeling of a watershed

The availability of such wide range of tools has resulted in the adoption of GIS technology across various industries such as science, government, academia, business for applications including real estate, public health, crime mapping, national defense, sustainable development, natural resources, archaeology, regional and community planning, transportation and logistics [7].

2.4 GIS based modeling softwares

This section gives a brief overview of modeling softwares that integrate GIS principles and methodologies for topological modeling of infrastructure systems.

2.4.1 TransCAD

TransCAD is a transportation modeling software that leverages the capabilities of Geographic Information System (GIS) to store, manage, display and analyze transportation data. It supports all modes of transport and provides modules for routing, travel demand forecasting, public transit, logistics, site location, territory management, and decision support systems [8].

TransCAD makes use of a network data structure that aids in routing and network optimization problems. Modeling transportation networks require accurate data representation as network distances and travel times depend on the actual shape and connectivity between transportation features. This is ensured by topological tools available within the GIS framework. Networks in TransCAD are also capable of managing complex network attributes such as road blocks, one-way streets and intersection delays that impact network analysis. The software also supports dynamic segmentation and linear referencing for transportation network data [9].

The GIS based approach enhances visualization capabilities by providing graphical solutions which help the non-practitioner comprehend complex technical information.

2.4.2 MIKE SWMM

MIKE SWMM is an engineering software for the modeling and simulation of hydrology and hydraulics for urban storm water and waste water systems. It integrates the modeling capabilities of Storm Water Management Model (SWMM) with an improved user interface and cutting-edge simulation and visualization capabilities [10].

The United States Environmental Protection Agency (EPA) developed SWMM, a simulation model to estimate rainfall runoff quantity and quality from primarily urban areas for use in a single event or long-term simulation. The simulation model operates on a group of subcatchment areas that receive precipitation and generate runoff. The routing component of SWMM transports the runoff through pipes, channels and treatment devices. The model then tracks the quantity and quality of runoff generated within each subcatchment along with flow rates, flow depths and quality of water in each transport media during a simulation period [11].

MIKE SWMM encompasses GIS capabilities via an ArcView based model MOUSE (Model for Urban Sewers) GIS to provide spatial and visual representations of models which can have their data stored in GIS databases. MOUSE GIS is a collection of model simplification tools with the ability to import and convert sewer and drainage system data from a wide variety of formats [12].

CHAPTER 3

DESCRIPTION OF TOPOLOGICAL MODELING TOOLS

3.1 Overview

The first section in this chapter introduces the concept of GIS and features which are vital to the development of the modeling toolkit in this thesis. The second gives a brief overview of the programming language and spatial library used to accomplish certain geometric tasks within the tool framework. The following sections describe the principles and methodologies adopted to build star and trunk topologies.

3.2 GIS features

Most current open source GIS software systems support topological modeling, but Environmental Systems Research Institute's (ESRI) commercial mapping platform, ArcGIS has been chosen due to its wide use and extensive support for integration with programming languages. ArcGIS is a suite of geospatial products with ArcMap and ArcCatalog being the primary components used in this research. ArcCatalog serves as the data management and organizing tool, while ArcMap is primarily used for visualization, editing and analyzing geographic data. The following sections describe key features of the ArcGIS framework that play a significant role in the programming approach.

3.2.1 Feature class

A feature class is a homogeneous collection of features of type point, line or polygon and share a common set of attributes. Feature classes can be found in a feature dataset sharing the same coordinate system and organized into networks or exist independently in a geodatabase. The geographic data read and generated as part of the programmatic approach is stored in feature classes.

3.2.2 Geoprocessing

Geoprocessing in general is a GIS operation that manipulates GIS data. A typical geoprocessing function involves performing an operation on an input dataset and consequently producing a new dataset. The fundamental purpose of geoprocessing is to automate tasks involving repetition of work. These tasks can encompass a single tool or a series of tools combined into a sequence of operations known as workflows.

Figure 3.1: Data transformation in geoprocessing [13]

Geoprocessing in ArcGIS can be tools that run within the realm of the ArcGIS desktop interface or as standalone Python scripts that build on the capabilities of the 'geoprocessor' object. The geoprocessor is an object that manages all the geoprocessing

functions available within ArcGIS and exposes these methods for access in Python. The geoprocessor plays a key role in the development by making it possible to read and write geometries.

3.3 Python

Python is an interpreted, interactive, object-oriented programming language that offers strong support for integration with other languages and tools. It's easy to learn syntax emphasizes code readability, modularity and therefore makes it very attractive for rapid application development and reduced program maintenance. Python's extensive support for modules and packages encourages program modularity and code reuse.

The choice of Python as the preferred language can be attributed to the extensive support that ArcGIS provides to its geoprocessing framework and widespread use of Python in its user community. The following section describes a Python package that has been used to execute geometric operations essential to topology generation.

3.3.1 Shapely

Shapely is a Python package for analysis and manipulation of geospatial geometries [14]. The 'shapely' geometry module not only supports standard feature types – points, lines and polygons but also multi-point, multi-line, multi-polygon geometries and other complex geometries. It has the capability to produce new geometries such as buffer, boundary, centroid, convex hull and also test if shapes intersect, cross, contain or

touch each other. In this thesis, ‘shapely’ has been primarily used to determine scalar properties such as distance and check geometric association between geometries.

3.4 Infrastructure Topology Generator (ITG)

The Infrastructure Topology Generator (ITG) is the name given to a set of scripting tools developed in the Python programming language to create generic models of randomly generated infrastructure elements. The models provide insight into the spatial arrangement of elements by collectively organizing infrastructure representations into star, trunk and mesh network topologies. The scripts function by leveraging the capabilities of the ArcGIS geoprocessing framework for tasks such as reading and writing features, creating feature storage containers and managing workspaces. Equally significant to the tools are programming constructs, modules, functions and methods inherent to the Python language. The tools have been developed adhering to certain programming principles such as commenting code wherever necessary, replacing repetitive chunks of code with functions and deleting objects to reduce memory consumption.

The tools extensively make use of ‘lists’ which is one of the several types of sequences supported by Python. A list is the most flexible data type available in Python and is used as a container for items. Most lists implemented across the toolkit store coordinate pairs of point or line features. List objects support several operations including adding, indexing, slicing, multiplication, membership and finding largest and smallest elements. Another important aspect of Python programming is the use of ‘modules’.

Modules aid in the logical organization of related Python code components for easier understanding and use. A module is any other Python script capable of being imported into other scripts exposing its functional aspects in the form of classes, functions and variables. For example, the geoprocessing features of ArcGIS are imported into the tools via the module 'arcgisscripting'.

Code developed in scripting languages such as Python can be written in text files and saved with the required language extension (.py for Python files) before executing them with the Python interpreter. This being a viable alternative is not best suited when developing complex programs that require numerous lines of code. In such a scenario, a software application known as Integrated Development Environment (IDE) helps maximize productivity by providing a set of integrated tools within a single interface. An IDE normally consists of [15]:

- A source code editor
- A compiler or an interpreter
- Build automation tools and
- A debugger

The standard Python installation comes with a built-in IDE known as IDLE which supports features such as syntax highlighting, auto completion and smart indentation [16]. Due to a lack of adequate support for debugging (process of finding defects in a computer program) in IDLE, another standalone IDE known as PythonWin with rich

feature support and integrated debugging facility has been used in the development of the toolkit.

The following sections describe in detail the inner workings of the three tools developed for topology generation. The sections first provide an overview of the process in which topologies are created and go on to elucidate in detail the programming constructs.

3.5 Star topology tool

The star topology is a network of nodes connected via edges to a central node or hub. The topology generation first requires creation of a finite number of random nodes or abstract points which are consequently converted into features in a point feature class using the geoprocessing functionality of the ArcGIS framework. The conversion into ArcGIS point features is essential for visualizing the network.

The point features are looped through one at a time and appended to a Python list. This is followed by random selection of a central node from which a point object is created and consequently removed from the list. An empty line feature class to hold the edges connecting the hub and remaining points is generated. Looping through the nodes in the list, a line object for every link between the hub and corresponding point is created and inserted into the line feature class. The steps involved or the flow of program execution for generating a star topology is depicted in Figure 3.2.

A star topology is thus produced with nodes and edges stored in two discrete feature classes. The next section describes in detail, steps of the procedure from a programming perspective.

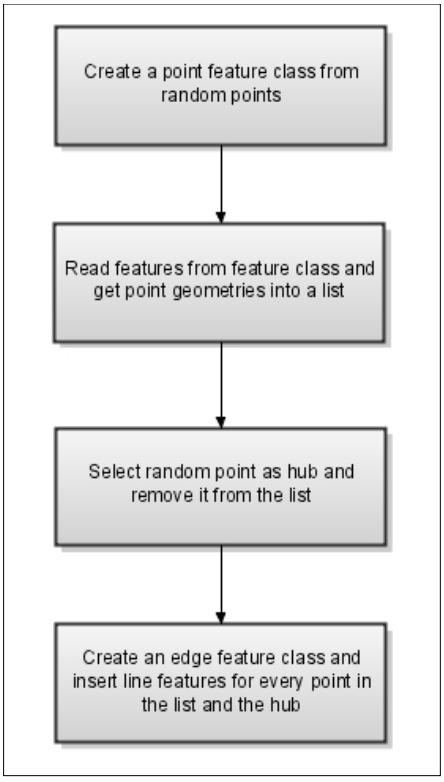

Figure 3.2: Script execution sequence for star topology

3.5.1 Tool description

Python being an interpreted language executes instructions in a sequence. The star topology generation tool comprises programming statements written in sequence. The objective and procedure involved in generating star topology being relatively simpler to that of the trunk tool, and with most operations being non-repetitive, the code has been

structured into sequential statements without the use of functions. This enhances code readability and makes debugging easier.

The program begins by importing required modules whose features are subsequently utilized in developing the tool. One of these modules is part of the toolkit and provides the star and trunk tools with the ability to generate random point locations. The module essentially is a Python file with a class definition and a method to generate random nodes. The class function utilizes methods exposed by a built-in module within the standard Python library named *random*. The module provides functions for generating random integers between a definite range of numbers or choosing a random item from a list of items. This custom class module implements one method *getRandomCoords* by taking as input the number of points to generate. A coordinate pair is then randomly generated within the defined bounds and is consequently appended to a list which is returned to the object calling the function. An information description also known as pseudocode of the random coordinate generating function is depicted in Figure 3.3.

```
Define function 'getRandomCoords' with argument <number of nodes to generate>:
Create a list object 'nodes' from number of nodes required
Create an empty list 'coords' to store coordinates of each node
For every node in the 'nodes' list:
        x-coordinate = get a random integer between 0 and 100
        y-coordinate = get a random integer between 0 and 100
        Append x and y coordinate values to the list 'coords'
Return 'coords' list to the object calling the function
```

Figure 3.3: Pseudocode for random node generation

Following the import of modules, the much required geoprocessor object is created and workspace for the tool environment is set. An empty feature class is created in the workspace to store all the point coordinates that are generated. When creating feature classes, the code implements a check to verify if a feature class with an identical name already exists in the workspace. In the event of an already existing file, the geoprocessor ensures that the feature object container is removed. Inserting features into a GIS feature class requires a *cursor* which is a data access object that iterates over rows in a table. Cursors are provided for searching, inserting and updating features. Each type of cursor is created by a corresponding geoprocessor method. With the creation of an insert cursor on the empty feature class, an object of the *random* class calls the function *getRandomCoords* with the required number of points and stores the returned coordinates in a list. For every point item in the list, a GIS point object is created and its 'X' and 'Y' attributes are assigned from coordinates of the respective item. The insert cursor now creates a new feature (row) for every point whose shape is set to the geometry of the point object. The cursor then inserts the feature into the container. The pseudocode for creating a feature class with randomly generated points is illustrated in Figure 3.4.

```
Create a geoprocessor object

Set the workspace for the tool

Set 'star_nodes' as the feature class name to hold coordinates

If feature class already exists delete it

Create the feature class in the workspace using the geoprocessor with the set name and feature
type which is 'point'

Create an insert cursor 'insCur'

Get random coordinates from the 'getRandomCoordinates' function into a list

Loop through each coordinate pair in the list:

        Create  a point object

        Set the point's x and y properties to the current coordinate

        Create a new feature using the insert cursor

        Set the point object to the shape of the feature

        Insert the new feature using the cursor
```

Figure 3.4: Pseudocode detailing feature class generation from random points

The feature class container now consisting of all point features is read using a search cursor by looping over features and gathering geometry of each point into a Python list. The next step involves choosing a random point from the list as the hub of the topology. The *random* module implements a method of the name *choice* for random selection of item in a list. A point is chosen from the list of geometries and stored in a variable to function as the central node or hub. The point selected is therefore excluded from the points list and a GIS point object is created from it. Having identified the hub and point features, the next step in the process is to create line (edge) geometries

connecting the hub to its nodes. An empty feature class capable of storing line features is created upon which an insert cursor is generated. Looping through the points in the list, an array object to store the two end points is created. The point being iterated over and the hub object are added to the array which is assigned as the shape of a new feature in the feature class. The insert cursor creates a new row for the feature and the container is thus updated with links. The feature class with line geometries connecting the hub to its nodes is showin in Figure 3.5.

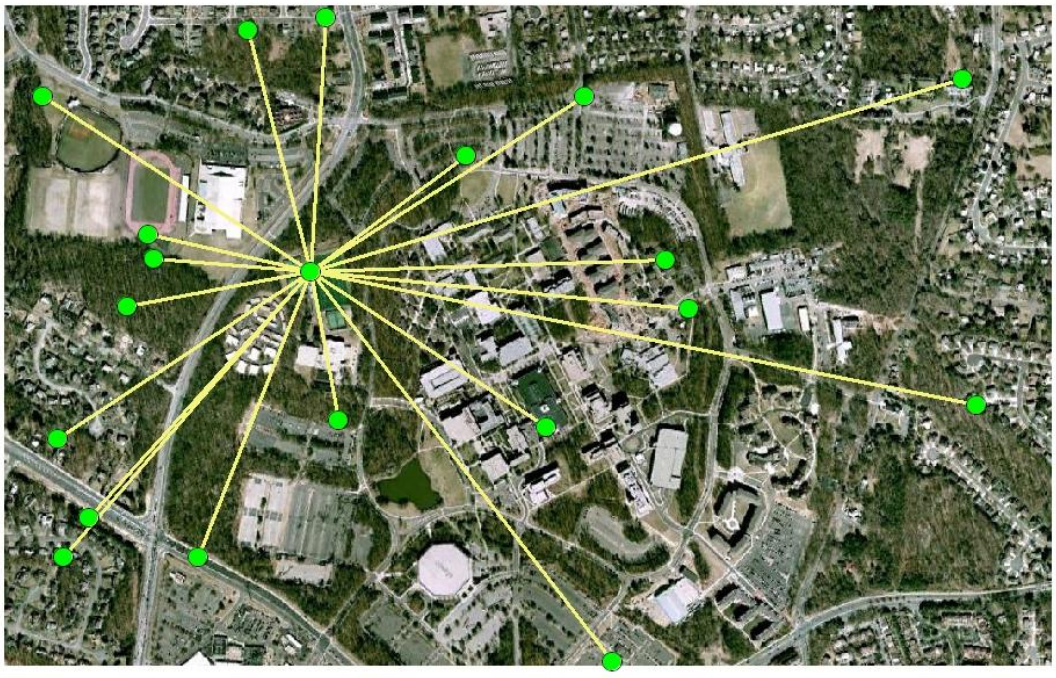

Figure 3.5: Star topology

3.5.2 Example

Star shaped networks also referred to as hub-and-spoke networks do not exist independently in the real world but are often part of larger networks. A pure implementation of star topology is possible in command and control environments where a central facility acts as a dispatch and monitoring center responsible for operations of connected facilities. A star topological network by itself can be non-reliable as failure of the hub instantly incapacitates the entire network.

In a star topology, the central node has a much higher degree of connectivity than nodes to which a point-to-point connection exits from the hub. The remaining nodes in the network which only connect to the hub all have a single degree of connectivity. The number of links characteristic to a star topology is always one less than the number of nodes. A large scale implementation of the star model can commonly be found within the transport industry. The air transportation system in a country is a large network of connected hubs with each hub signifying center of a star network. Large cities often act as transportation hubs handling significant traffic volumes and connect to regions with relatively lesser traffic. The hubs in turn are connected to other hubs making up a larger network.

Barabasi describes how an air transport system resembles a scale-free network by comparing it with a highway system that is based on the principles of random networks. In a highways system, cities representing nodes are connected by highways and there are no cities that are served by hundreds of highways [17]. The degree distribution of such a network follows a bell shaped curve indicating that most nodes have equal number of

links and nodes with a large number of links do not exist signifying a uniform network. Air transport networks on the other hand follow a power law distribution in which there is no single node that is characteristic of all the other nodes, therefore lacking any scale. In such networks, most nodes only have a few links and are held together by few highly connected hubs. This is similar to the case in which a large number of smaller airports are connected to each other through a handful of hubs. Figure 3.6 illustrates random and scale-free networks.

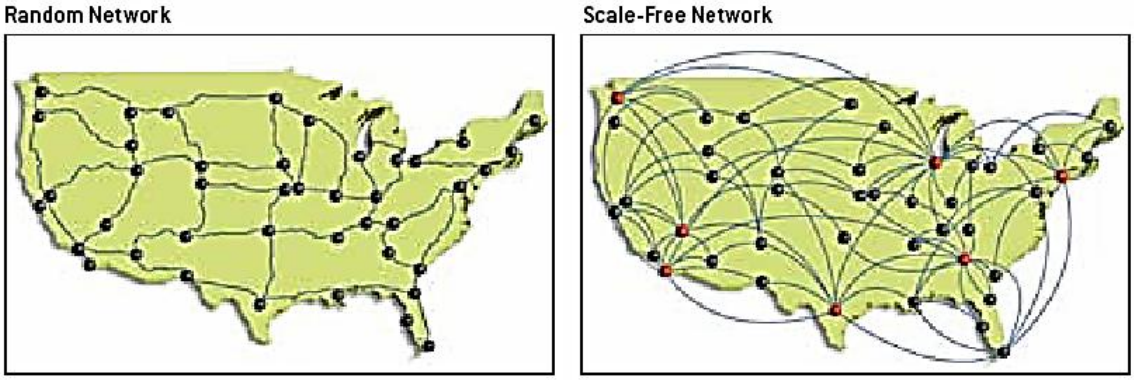

Figure 3.6: Random and scale-free networks [18]

3.6 Trunk topology tool

The trunk topology generation is a more complex process in the sense that certain topological rules governing the organization of nodes and edges have to be complied with. The development for this topology involves breaking down the process into numerous simple functions to avoid code repetition and reduce complexity. The random node generation and transformation into ArcGIS point features in a feature class is similar to that described in star topology construction. The trunk tool generates three

feature classes in the process with one containing all point features, one with a single polyline feature representing the trunk and one with edges connecting nodes to trunk.

The first step involves reading the point features into a Python list which is then passed to a function that results in a list of nodes participating in the trunk. The points of the trunk are randomly selected and subjected to a validation function that implements a procedure to check if the trunk intersects with itself. On successful verification, the points of the trunk are excluded from the entire points list. The Python list with points of the trunk is passed to a function that generates a discrete feature class for the trunk main.

The next step is to create a line feature class to store associations between remaining nodes and the trunk. For every node in the list, its straight line distance to each participating node in the trunk is computed. The trunk node that is nearest is determined as the one to which a link must be generated. With this identification of node pairs, the nodes are passed to a function that creates a line object and inserts it into the line feature class. Figure 3.7 illustrates the program execution in steps. A detailed description of the procedure adopted to create this tool is elucidated in the following section.

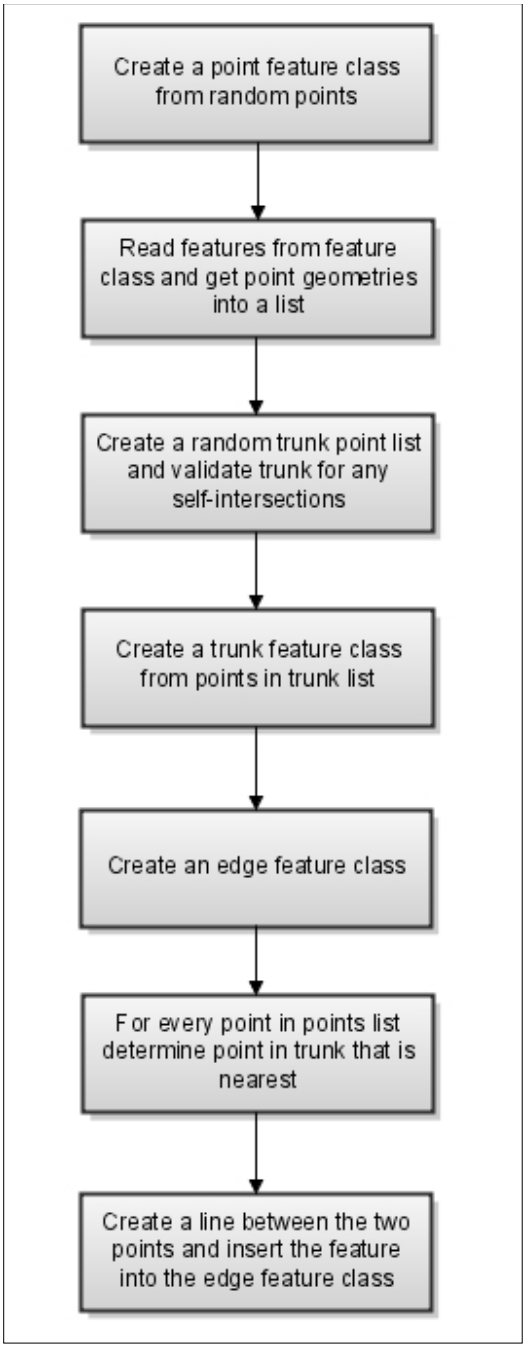

Figure 3.7: Script execution sequence for trunk topology

3.6.1 Tool description

Before delving into the specifics of tool development, it is essential to understand that complex and lengthy code when broken down into smaller chunks of related instructions with discrete functioning objectives, simplifies code manipulation, prevents code repetition and makes testing far less complex. Unlike the star topology tool which has been developed using sequential instructions the trunk tool relies heavily on the use of functions. The following tasks requisite for trunk network generation are implemented as functions:

- Reading point geometries from a feature class into a Python list
- Creating feature classes using the geoprocessor
- Identifying random points that make up the trunk
- Check to verify validity of trunk
- Creating a feature class to store the trunk main feature and
- Joining outstanding point features in the list to trunk

The Python tool for trunk topology generation utilizes the custom class similar to the star tool for generating point coordinates. The script implements a function for creating feature classes that takes as arguments, the path to the feature class and the type of features (points, lines or polygons). Upon successful execution the function returns an insert cursor. By providing a discrete function for creating feature classes, the need to

repeat instruction each time for this purpose is avoided thus reducing length and complexity of code.

An object of the custom class is created and the method for obtaining randomly generated point coordinates is invoked. The point locations returned are stored in a list. An empty point feature class is created by passing arguments required by the *createFeatureClass* function. The insert cursor returned by the function is used to insert points in the list after converting them into GIS point objects by looping over one another.

The points feature class updated with features from the random points list is then passed as parameter to another function *readTestPoints* that uses a search cursor to iterate over the features and retrieves point geometries appended to a list. The list with coordinates of each point in the feature class is returned and consequently passed onto another function *createTrunkList.* This function takes as parameters the node list and a numerical value that indicates the number of points that should make up the trunk. The objective of this function is to create a list of points that participate in the trunk. This is achieved by iterating over the entirety of points received as function argument and choosing a random point. A check is implemented to ensure that a point once chosen is not selected again. The point is then appended to an empty list container for points that create the trunk. Once the required number of non-repetitive random points is accumulated, the program breaks from the loop. Having identified points of the trunk, the next task is to validate the trunk for any self-intersections.

A topology rule that governs the spatial organization of points creating the trunk is implemented via a function *checkTrunkValidy* which requires as input the list of points forming the trunk. The function makes sure that the points chosen do not result in a trunk main that crosses itself. This test for geometric association is made possible by the Shapely library from which a *LineString* class is imported into the tool. The trunk is essentially a polyline made up of connected line features or line strings. The trunk points are iterated over and a line string object is generated by the class imported using a point and its adjacent in the list. The line string objects are therefore stored in a list object. With no possibility for two line objects sharing a common point to intersect, each line in tested with non-adjacent lines utilizing the *crosses* binary spatial operator. If the test determines that the trunk is invalid, the execution flow returns to the *createTrunkList* function to generate a new set of points. This process repeats itself until a valid set of points for the trunk are selected. The pseudocode describing the structure of the two functions and the control flow is depicted in Figures 3.8 and 3.9.

```
Function createTrunkList <points list, number of points required for trunk>

Set 'bool' = true

While bool is true:

    Create empty list 'trunk'

    For every point in points list:

        Trunk point = choose random point from list

        If there is no occurrence of trunk point in the trunk list append it to trunk list

        If the number of points required for trunk are obtained break out of loop

    Check if trunk is valid by passing trunk list as parameter to the function
    'checkTrunkValidity' and store the result in variable 'bool'

    If bool is true regenerate a new trunk

Return the trunk when valid trunk is generated
```

Figure 3.8: Random point selection for trunk

```
Function checkTrunkValidity <trunk list>:

Create an empty list 'lines'

For each point in trunk:

    Set line = create line using point and its following point

    Append the line to the list of lines

For every 'line1' in the list excluding the last two lines:

    For every 'line2' starting from 'line1' + 2nd line to the last line in the list:

        bool = does line1 cross line2

        If bool is true (trunk is invalid):

            Return bool
```

Figure 3.9: Checking trunk for self-intersection

The identification of a valid trunk is followed by creating a feature class with the lone trunk main feature. The trunk points are passed on to a *createTrunkFC* function which acquires itself an insert cursor from the *createFeatureClass* function. An array object with all the trunk point objects is assigned to the shape property of a new feature. The trunk feature is then inserted into the feature class. Figure 3.10 illustrates a trunk generated from randomly generated points.

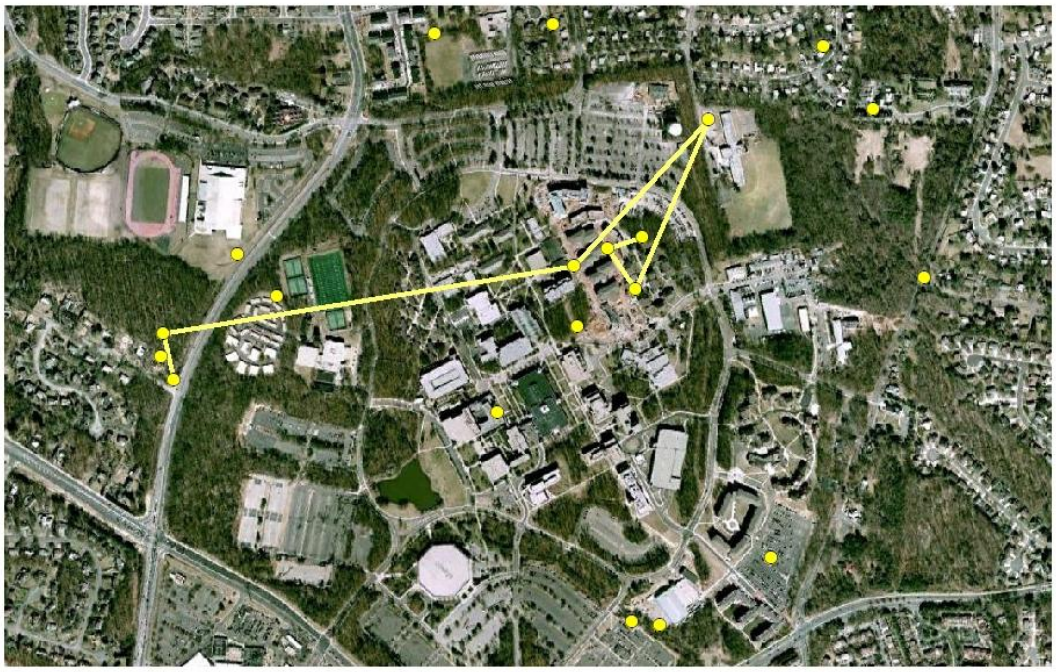

Figure 3.10: Trunk generated from random point features

The trunk having been identified, its points are excluded from the list containing all points generated for the topology. The last task in the process involves generating edges that establish spatial connections between the outstanding point features and the

trunk. An empty feature class to store edge features is created and an insert cursor is obtained. It is assumed that points attach to the trunk at nodes closest in distance. This requires calculating distance between every point feature and points of the trunk. The point features are iterated upon, and distance of each feature from every point in the trunk is calculated. The distances for a point feature are appended to a list and being numerical quantities, Python's minimum function is used to determine the least distance from the list. Utilizing the index of the shortest distance in the list, the trunk point which is closest to a point feature is deduced. This program flow for this process is described in Figure 3.11.

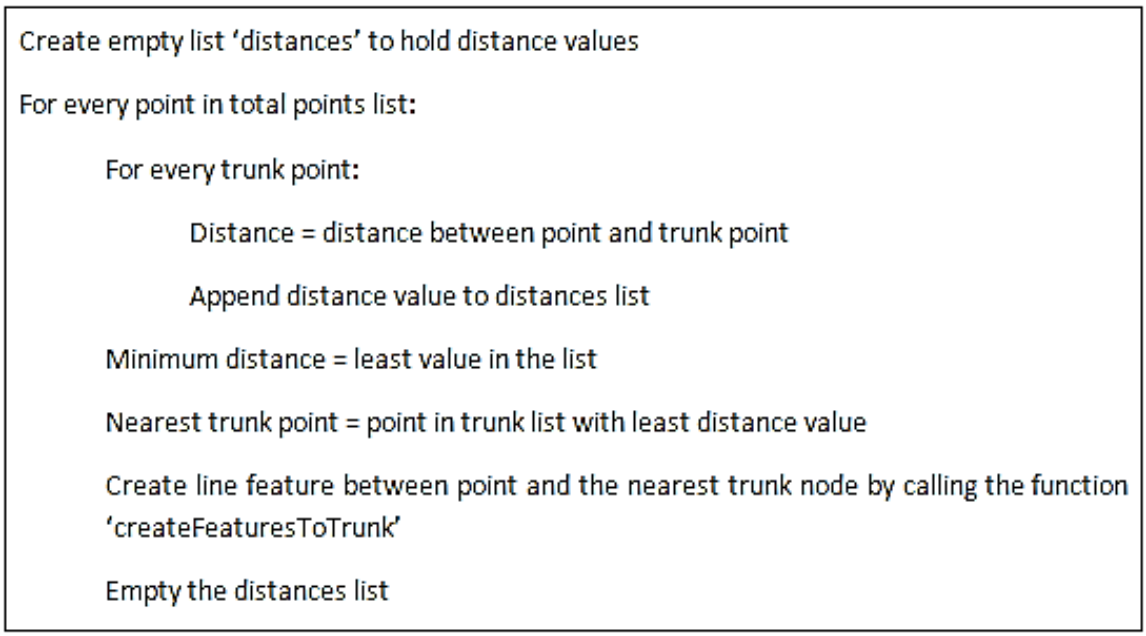

Figure 3.11: Determination of nearest trunk node

The point feature and closest trunk point along with the insert cursor are parameters to the function *createFeaturesToTrunk* which creates a line object with the points and inserts a new edge feature each time it is called. The function with its parameters and process of generating the output feature class is listed in Figure 3.12.

```
Function createFeaturesToTrunk < point, trunk point, insert cursor >:
Create a geoprocessor object
Create an array object using the geoprocessor
Create a point object and set its x and y coordinates to those of the point received as argument
Create another point object and set its coordinates to those of the trunk point
Add these two points to the array
Create a new feature in the feature class using the insert cursor
Set the array as the shape of the feature
Insert the feature using the cursor
Remove all points from the array
```

Figure 3.12: Creating edge features to trunk points

Figure 3.13 illustrates the generated topology with point, trunk and edge feature classes stacked upon one another forming a network of connected elements.

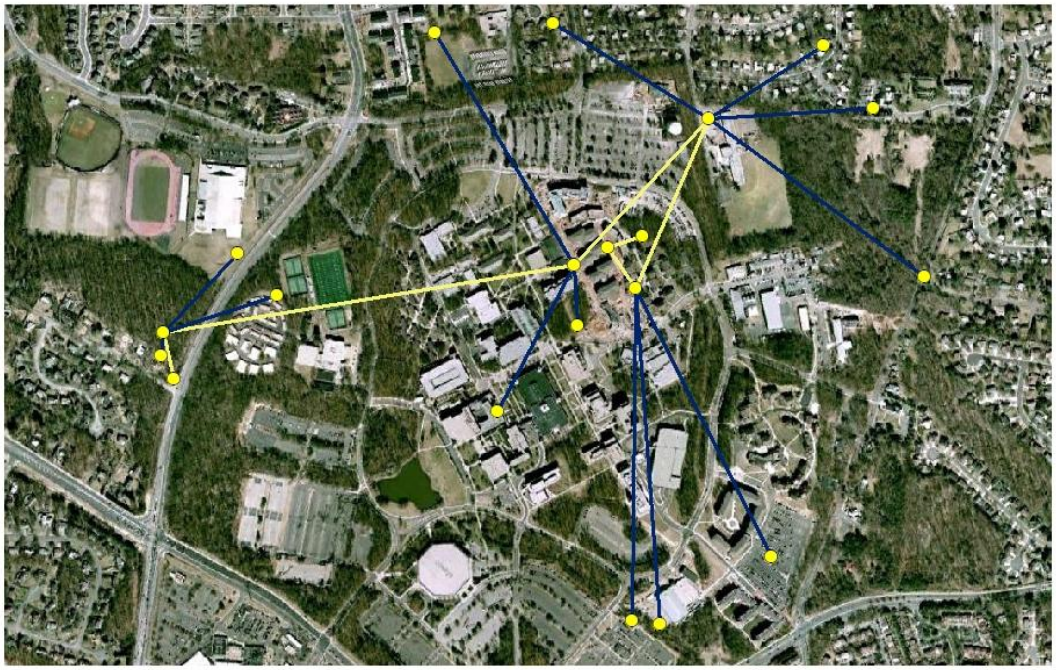

Figure 3.13: Trunk topology with point features connected to trunk

3.6.2 Example

A trunk topology is implemented for systems that require delivery of resources between sources and sinks by way of a trunk that runs through the network. The trunk can dispense resources to destinations connected at defined points of distribution that act as a hub or have sinks connected along its length via individual junctions. Figure 3.13 in the previous section depicts case of a trunk that connects to service nodes in the network at defined points along the length of the trunk.

A water distribution system best illustrates the application of a trunk based network. Pump stations represent the source from which water is distributed to customers (residential, commercial or industrial) using distribution mains that usually run alongside streets. Each customer is connected to the main via a lateral. The connections between service endpoints and distribution mains are governed by factors such as location of the customer relative to the main, purpose of the service and demand. A distribution network with mains and laterals is illustrated in Figure 3.14.

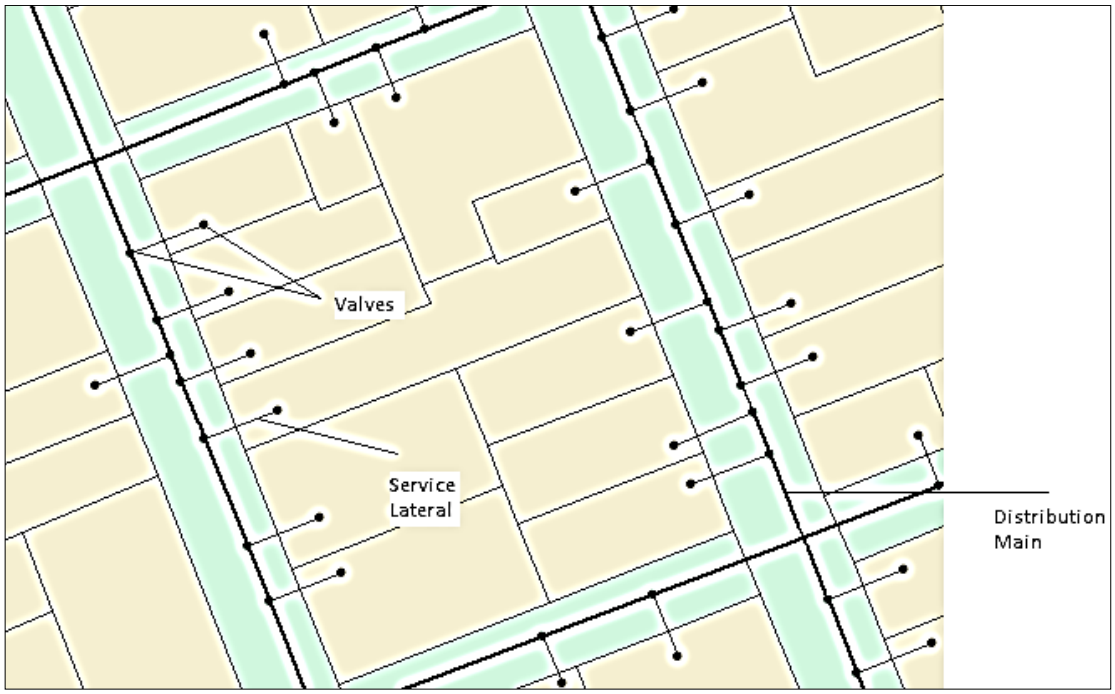

Figure 3.14: Water network depicting distribution mains and service laterals

For every service endpoint, there is just one lateral or link that connects to the distribution main, hence the degree of connectivity is one. Junctions at which the service laterals join the main have a constant connectivity degree of three with two links belonging to the main and one from the service line. Mains are pipes of equal or unequal diameters connected through junctions. These junctions vary in their nodal degree depending upon the number of number of mains that they connect to at an intersection. The degree of connectivity of these junctions can exist between the range of two and four. Segments of water distribution systems need to be taken out of service from time to time for maintenance and repairs. Those sections that need to be taken out of service are limited by the placement of shut-off valves. Higher the densities of these valves, fewer customers are affected with a reduced impact on the overall system operations. The number of these valves can vary depending upon design requirements. Since the objective of these valves is to obstruct flow in a pipe, the degree of connectivity is two. In the case of trunk topology generated by the tool, it can be observed that nodes of the trunk resemble hubs of a star network and have higher degree of connectivity than nodes that connect to them with individual links.

3.7 Mesh topology tool

A mesh topology can be described as a network of nodes and edges in which every node is attached to all other nodes. The mesh topology generation follows a more straightforward approach due to participating nodes having equal degree of connectivity. The Python based tool developed for mesh generation is built upon functions previously implemented in the trunk tool for creating node and edge feature classes. The coordinate pairs required for generating nodes are obtained from the custom class object built with the ability to create random Cartesian coordinates given the required number of points. The nodes are subsequently converted into features in a point feature class through geoprocessing techniques and are consequently read into a Python list object. Iterating over nodes in the list, an in-memory line or edge object is created between the current and each following node in the list. The edge features thus generated are inserted into a line feature class. The feature classes together form a mesh topology. The programmatic workflow involved in generating the network is illustrated in Figure 3.15.

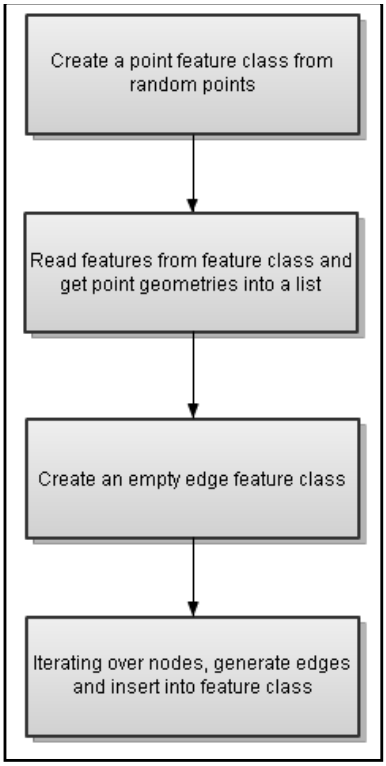

Figure 3.15: Process workflow for mesh topology generation

3.7.1 Tool description

The mesh topology generation involves an unambiguous approach due to the availability of methods for reading coordinate geometries into native Python objects and generating output features previously implemented for the trunk tool. The random nodes required for the network are generated in the manner similar to that demonstrated in the descriptions for star and trunk tools. An object of the random point generation class – *randnodes*, is used to generate coordinate pairs for the required number of mesh nodes. Figure 3.4 describes the process involved in generating the nodes. An empty feature class

for storing the nodes returned is created by the *createFeatureClass* function depending upon parameters such as type of feature, name and location on disk and returns an insert cursor. A geoprocessing point object is created for every node and its corresponding *x* and *y* properties are set before the cursor creates a new feature and the point is assigned to the feature's shape. The cursor consequently inserts the feature into the feature class. The pseudocode for this process is described in Figure 3.16.

```
Insert cursor = createFeatureClass (feature class name, type)
For every coordinate pair in random points:
        Create geprocessing point object
        Set point object's x and y coordinates
        Create a new row for feature in feature class
        Set feature's shape property to the point object
        Use the insert cursor to insert the feature in the row
```

Figure 3.16: Mesh node feature generation

The node feature class generation is followed by retrieving the coordinates of each point into a Python list using the *readTestPoints* function that makes use of a search cursor to iterate over each point feature. The list thus generated forms the basis for creating edge features that connect nodes. The feature class for storing edges generated in the process is created and the corresponding insert cursor is obtained. Each coordinate pair in the list is iterated upon, and a loop on points following the current point is applied

to create an array constituting start and end nodes of the respective edge being generated. The array thus produced is designated to the shape property of the new line feature created by the insert cursor and subsequently inserted into the edge feature class. The pseudocode detailing the edge creation procedure is shown in Figure 3.17.

```
Edge inserting cursor = createFeatureClass (edge feature class name and location, line feature type)
For every point in the list:
    For every point in the list following the above point to the last point:
        Create an array object
        Create point object with x and y coordinates set to point in the outer loop
        Create point object with x and y coordinates set to point in the inner loop
        Add the two points to array
        Create a row for the new edge feature
        Set feature's shape property to the array
        Using the cursor insert the feature in the row
```

Figure 3.17: Mesh edge feature generation

The node and edge feature classes together give rise to a topology arranged in the form of a mesh. A visual inspection of the feature layers as shown in Figure 3.18 corroborates the process employed in rendering a mesh topology.

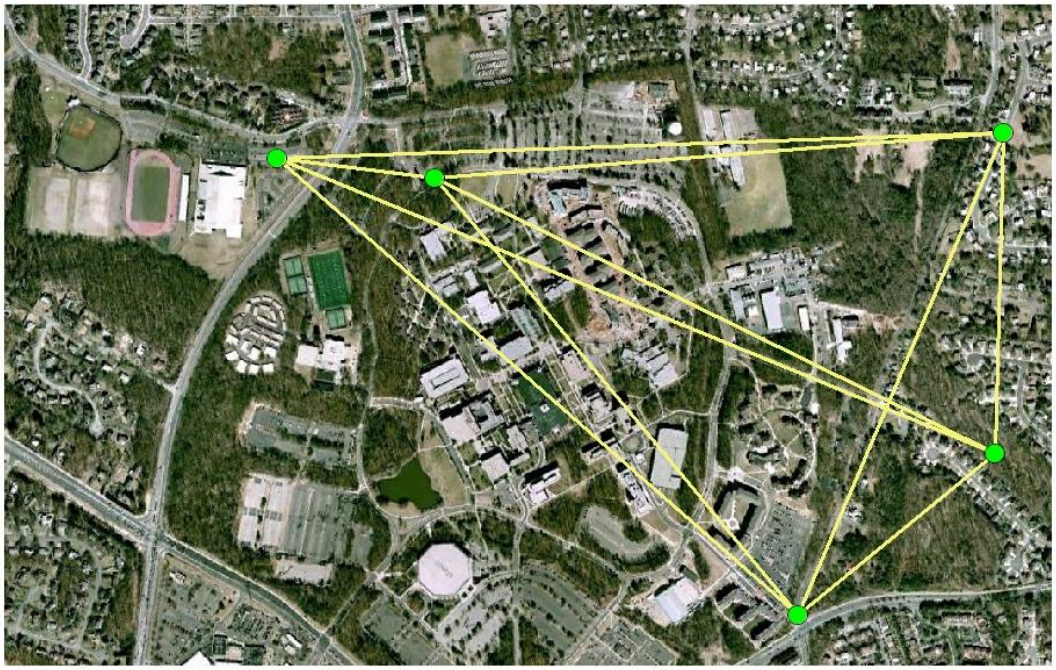

Figure 3.18: Mesh topology

3.7.2 Example

A mesh topology is a network in which all the nodes are connected to each other. Each node in a mesh network is independent of other nodes and maintains continuous flow between nodes. A mesh network is comprised of

$$n\,(n-1)\,/\,2 \qquad (2)$$

links that connect *n* nodes. Built upon the principle of redundancy, failure of a node does not disrupt network flow due to availability of alternate connection paths. Due to this

capability, mesh networks are known to be self-healing and reliable unlike star shaped networks.

An electrical distribution system is an example of an infrastructure system that implements a mesh based topology. It is one of the operations supported by an electrical network besides electricity generation and transmission. The components responsible for these operations are interconnected and form the electricity network or grid. The logical topology of an electrical grid varies depending upon constraints of the budget, system reliability requirements and the load and generation characteristics. The redundancy provided by mesh topologies is not deemed cost effective at the distribution level, but is considered to be a reliable alternative at the transmission level.

Every node in a mesh topology of n nodes has a $(n - 1)$ degree of connectivity. In the case of a mesh based electrical transmission network, failure or disruption of service at a generator does not interrupt service due to availability of alternate paths through which electricity can be transmitted. Though it may seem feasible and effective from a topological perspective, in a real network, the failure of a node and consequent re-routing of current to flow from other generators in the network over transmission lines of insufficient capacity can result in cascading failure and power outage across the entire network.

3.8 Discussion

The star, trunk and mesh topology generators provide a random topological model of connected nodes and edges which represent physical or virtual entities acting as sources or conduits for flow of resources or information and can vary significantly in their purpose depending upon the problem being addressed. These tools provide the necessary foundation for building spatially random networks for models that require an understanding of spatial organization and behavior of elements when data available at hand is limited or cannot be deemed accurate. The geographic bounds and number of nodes participating in the models can be controlled programmatically depending upon the spatial domain of the problem. Due to the random nature of topologies, the datasets produced are devoid of non-spatial information which is essential for most geospatial analysis procedures. The datasets can be edited to include attribute data as information becomes available in the course of the process. The spatial layout generated by the tools can be enhanced to reflect changes that may be induced by specific modeling requirements capable of improving the accuracy and in turn the results of any analysis that depend on the network.

3.9 Summary

The process adopted in the development of scripting tools capable of producing random topological models with spatial layouts in the shape of star, trunk and mesh is emphasized in this chapter. GIS feature classes serve as containers that hold geometry objects generated by the tools. The geoprocessing framework of ArcGIS plays a pivotal

role in the development of these tools by providing necessary objects that support reading, writing and analyzing geometric data. The wide adoption of Python within the GIS domain along with the availability of extensive geometry analysis and manipulation libraries such as Shapely motivated the use of Python as the preferred language.

Random data generation which is a key requirement for the tools is implemented as a class module that requires number of nodes as input and consequently generates a list of random coordinate pair values. The star topology is a network of nodes connected via edges to a central hub. The steps involved in generating such a topology is illustrated via a pictorial as well as described in detail. Trunk topology is a network of nodes that connect to nodes of a trunk. The edge generation between nodes and trunk points is dependent upon computation of proximity values. The complexities involved in trunk generation are described at length with code presented in the Appendix. Mesh topology is a redundant network built upon the requirement that each node must connect to all other nodes.

The following chapter presents a modeling scenario with the objective of programmatically generating a water distribution topology model using insufficient data and comparing the results with a reference model.

CHAPTER 4

MODELING A WATER DISTRIBUTION NETWORK

4.1 Overview

As described in Chapter 1, unavailability of data is the driving force behind adopting a random modeling approach. With the likelihood of results generated from such an approach varying considerably each time a model is generated, it is beneficial to have access to a model that can serve as a reference to compare the output with. This chapter elucidates a modeling scenario in which a close to real water distribution system of a neighborhood is used as a reference to evaluate a topological model generated programmatically for the same neighborhood assuming the case of insufficient data.

4.1.1 Reference data

The reference data used in the test case modeling was created by Environmental Sciences Research Institute (ESRI) using a database structure similar to that of the city of Montgomery, Alabama. The data is stored in a geodatabase and is comprised of feature datasets for landbase and water data. The landbase dataset comprises feature class data for blocks, parcels, road centerlines and edges of pavements. The water dataset constitutes data representing distribution mains, service laterals to parcels, tanks, system valves, location of fire hydrants and water network junctions. For the current modeling

situation, the reference network data belongs to a small portion of the entire geographic extent extracted from the south west blocks comprising of land parcels, their distribution mains along with service laterals to each parcel. Figure 4.1 illustrates the distribution system with network connectivity used for comparing the results generated network topology.

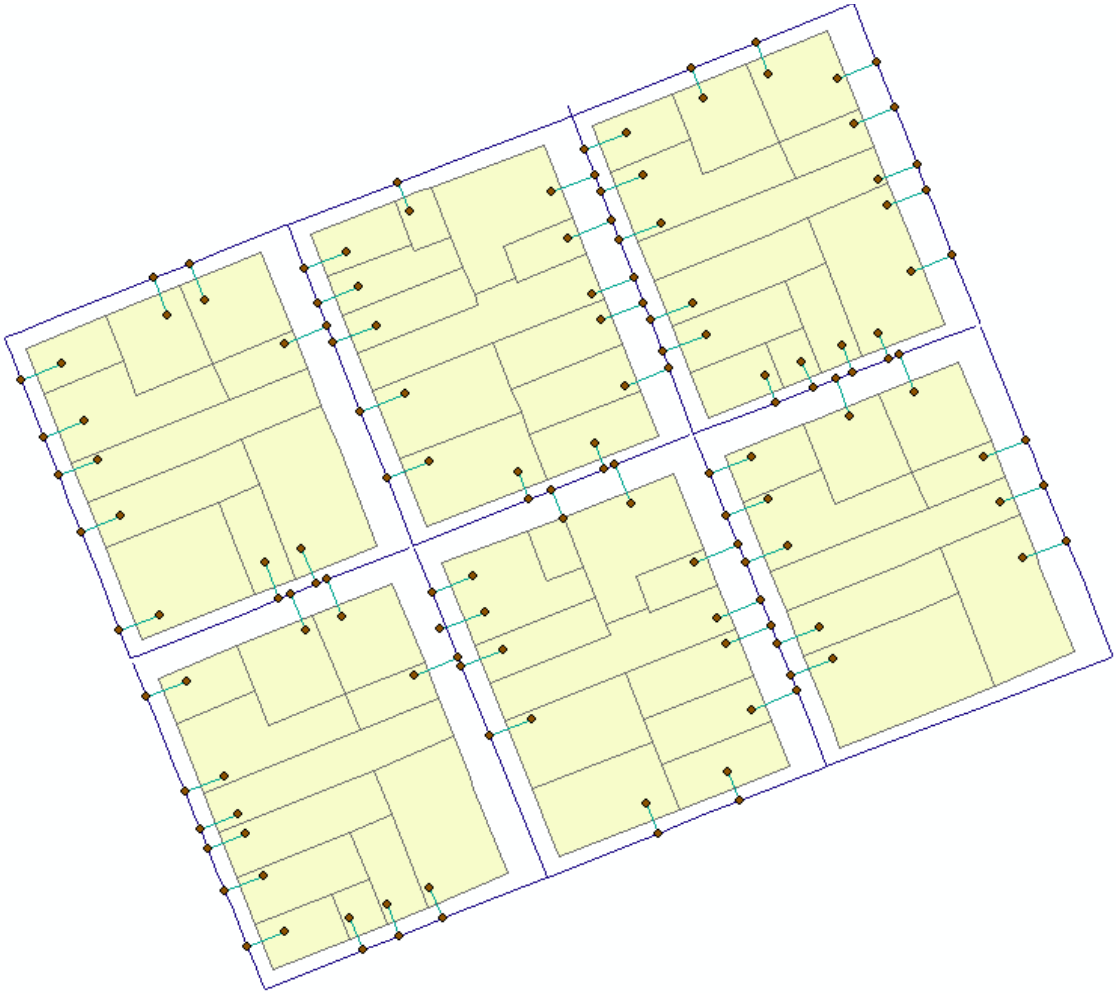

Figure 4.1: Reference water distribution network

4.1.2 Purpose

The purpose of this test case approach is to programmatically generate a topological model of network data from insufficient information such as land parcels, and utility mains, and consequently generating service lines from the mains to the parcels. The generated model is then compared with the actual reference model to test for similarities, inconsistencies, degree of accuracy and additional topological rules that can be administered for better results. The next section describes the procedure involved in the generation of service lines from mains to a network junction within a parcel.

4.2 Network generation

The modeling approach involves using land parcel data from the reference dataset as input and utility main features to generate network data comprising of service lines that connect point features within a parcel boundary to utility lines. The following sections describe the programmatic approach with illustrations of intermediate results and topological rules employed to arrive at the optimal and best network model.

The code for generating this topological model is developed in Python with extensive use of the ArcGIS geoprocessor object and Shapely library methods for creation of in-memory line and polygon geometry objects. The code developed is organized into functions with specific objectives and uses Python lists and dictionaries as intermediate storage objects.

4.2.1 Topological rules

The approach to modeling a water distribution system requires the network model to conform to certain topological rules by maintaining relationships of proximity and adjacency. One of the rules implemented ensures that service laterals do not cross adjacent parcels when connecting to their intended service nodes. The other rule applies only to parcels with more than one boundary available for service line connections. In such cases, the rule makes sure that service laterals connect to the nearest utility main. Figure 4.2 depicts violation of topology by service laterals connecting to utility main intersections.

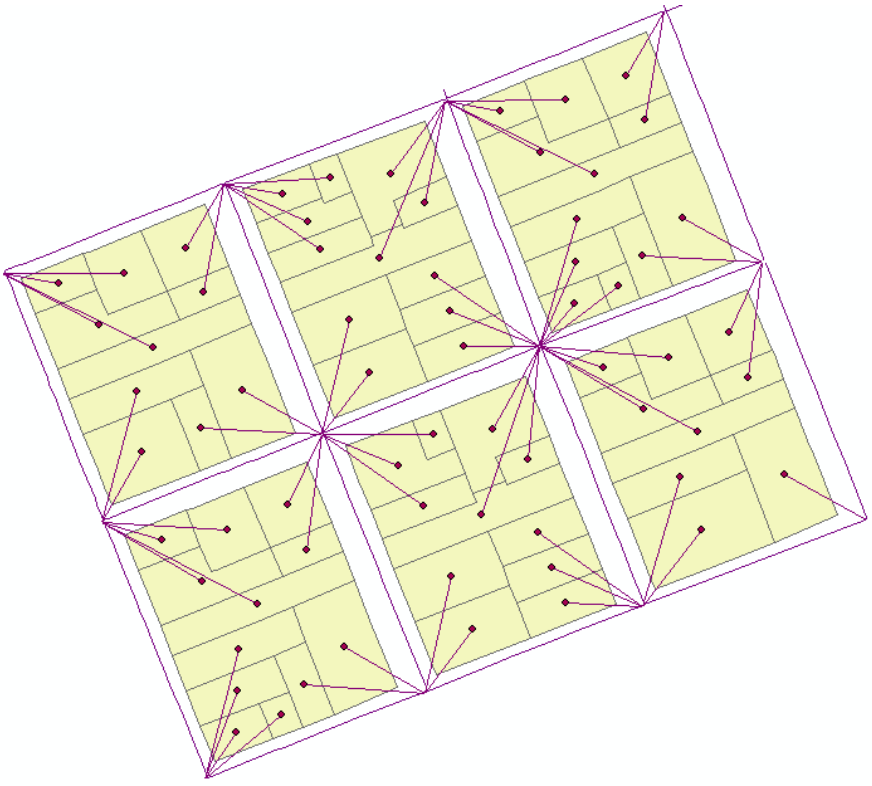

Figure 4.2: Violation of topological rule

4.2.2 The process

The process commences with reading parcel geometries to create point features that represent a point of connection within the extents of a parcel to the utility main through service laterals. This conversion of parcel polygons to point features is achieved by determining the centroids of each polygon feature and subsequently writing these point geometries into a feature class.

The next step involves determining the nearest utility main for each centroid in the parcel feature. This is achieved using one of the proximity analysis tools available within the ArcGIS framework. The tool enables calculation of distance from each point in the input feature class to the nearest point or line in another feature class of type point or polyline. The results are distance and location values along with the feature identification value appended to the attribute table of the input feature class. The input features in this scenario are the parcel centroids for which nearest distances to the utility mains are computed and corresponding point of intersection coordinates and identification values of nearest utility are updated as attributes.

OBJECTID *	Shape *	NEAR_FID	NEAR_DIST	NEAR_X	NEAR_Y
1	Point	11	71.691015	508476.648838	682739.986139
2	Point	11	72.632336	508380.742646	682701.4201
3	Point	11	50.3031	508285.857655	682663.264708

Figure 4.3: Updated feature attributes after nearest point analysis

Upon successful execution of the nearest point analysis tool, virtual in-memory service line objects are created for each parcel point feature using its geometry and coordinates of the nearest point on utility retrieved from the point's attributes. This is followed by

obtaining geometry of each parcel into a Python dictionary object in the form of key value pairs where the parcel object identification values serve as keys to the polygon geometry. This step leads to a similar process of acquiring geometries of features that make up the utility main. At this point of program execution, if the in-memory service line objects are converted into features in a feature class and visualized, it can be observed that laterals generated from the output of proximity analysis produce features that violate topology by crossing into adjacent polygons to connect to the closest main. This can be seen in Figure 4.4.

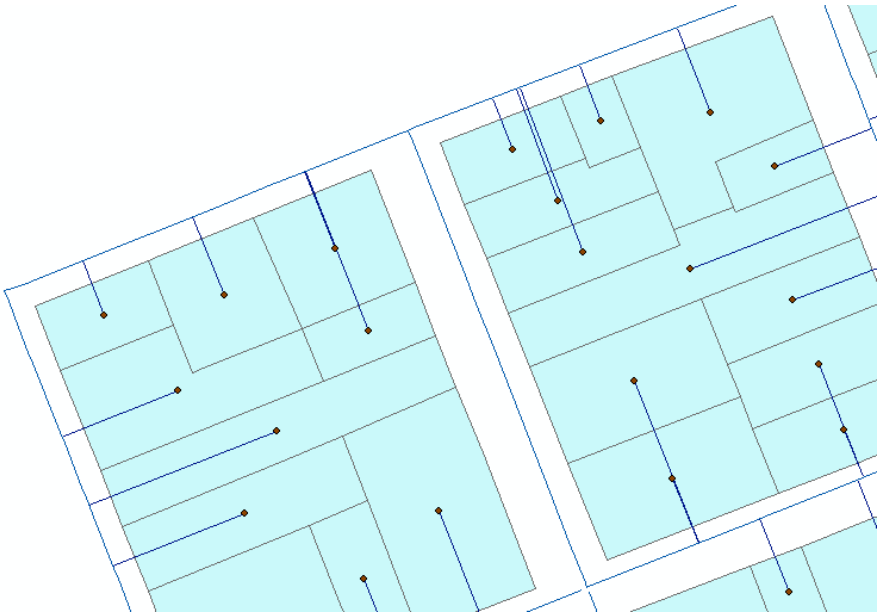

Figure 4.4: Service laterals crossing into adjacent parcels

This intermediate undesired output is avoided by storing service lines as objects in memory. An important observation made and utilized in addressing the crossing laterals situation is that a parcel polygon, the centroid feature representing it and the service line

connecting it to the nearest utility main, all share the same object identification values. The following step in the process involves looping through each service line and identifying those polygons through which lines pass but do not share the same identification values. For every such service line identified, the nearest utility found is discarded from the utility main geometries derived into a Python dictionary, and the distance from the corresponding parcel centroid is computed to every other utility geometry. The shortest of the distances is identified from which the required utility is obtained and the new point of intersection for the service line is computed which now conforms to the topological relationship. With the calculation of new closest points and their distances to the utility mains, the feature class with parcel centroids is updated with the new attributes. The service lateral features are now created by making use of the centroid geometry and the updated coordinate attributes for the points of intersection on utility main features. The pseudocode for service lateral generation procedure is illustrated in Figure 4.5. The code for the test case modeling is depicted in the Appendix.

```
For each service lateral:
    For each parcel polygon:
        If service lateral crosses polygon and object identification values do not match:
            Discard utility line to which lateral connects
            Get parcel centroid geometry
            For every utility main:
                Calculate distance from centroid to main
                Append distance values to a dictionary with utility id as key and
                distance as value
            Sort the dictionary with ascending distance values
            Get utility that is closest from the distance
            Calculate the point where service line would intersect the utility using the
            distance found
            Gather the points into a list
Update the centroid feature class with point geometries
Create feature class with new service lateral features
```

Figure 4.5: Pseudocode for service lateral generation

4.2.3 Results

The feature classes created as part of the rule based topological modeling process represent a series of connected nodes and edges in a water distribution network. A visual inspection of the network suggests that the model generated although not identical to the reference model, has a similar network layout with the orientation of most service lines coincident with those in the reference model. Figure 4.6 illustrates the distribution network when feature classes are stacked upon one another.

Figure 4.6: Generated water network model

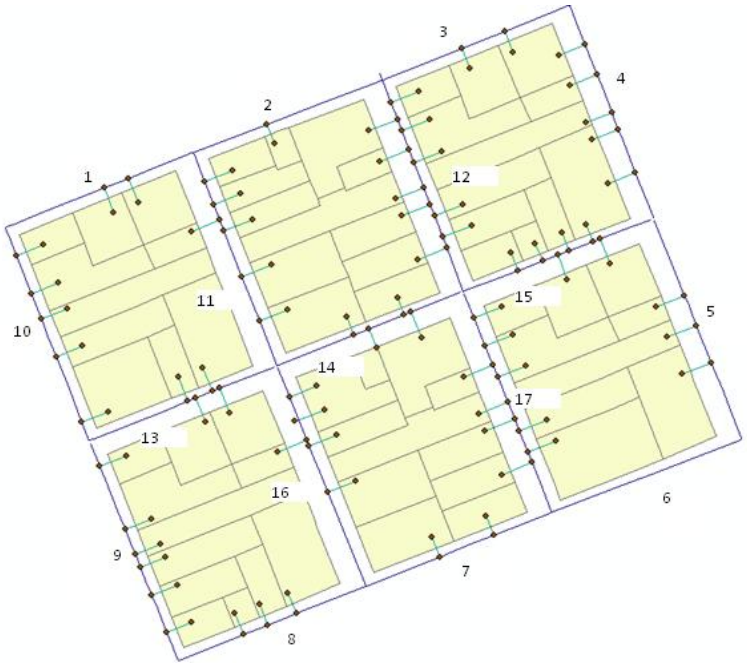

Figure 4.7: Reference water network model

The degree of connectivity for each service connection junction on the utility main is three and that of network junctions within individual parcels is one in both the generated topological network and reference model. Utility main intersections have varying degrees of connectivity depending upon the number of mains that intersect at the given location. In both models, it can be observed that the connectivity degree varies between two and four.

From the reference model in Figure 4.7, it can be observed that few parcels have more than one service line to the utility main such as the top rightmost block. The absence of this phenomenon in the generated model can be attributed to the lack of sufficient data. A comparison of generated and reference models to determine the variations in the number of service line connections to each distribution main results in the following graph illustrated in Figure 4.8.

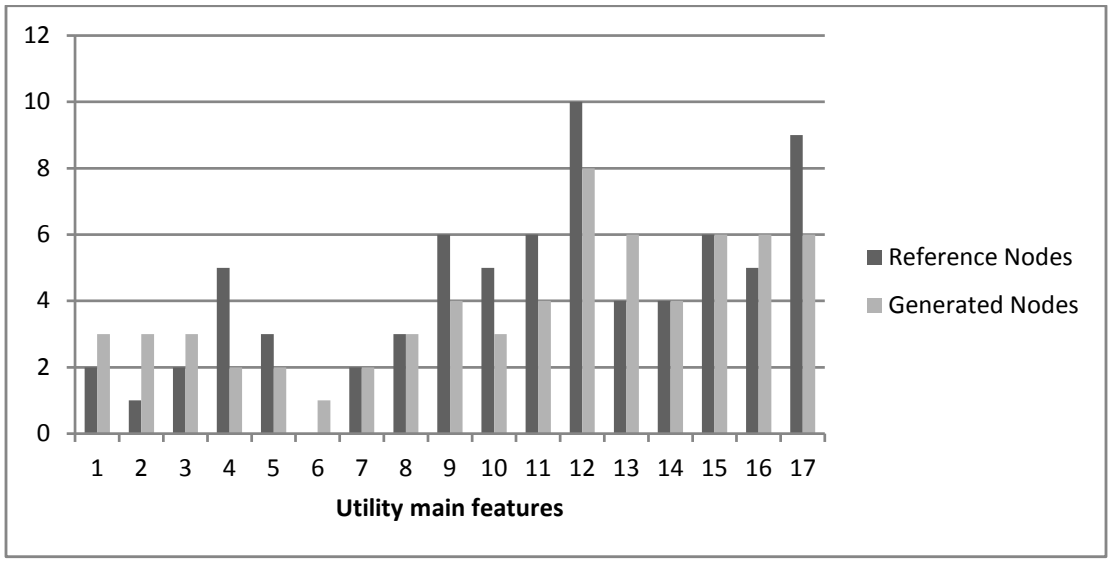

Figure 4.8: Service line connectivity to each utility main feature

From the graph, it can be observed that the maximum difference between service line allocations to a utility feature is three. The variation in distribution of service lines across the network for the generated model is more uniform, unlike that in the reference network where sudden rise in service connections is predominant. This dissimilarity can be attributed to the presence of multiple service connections to a parcel with a probable cause of high demand.

With the availability of additional data and information pertaining to the organization of features such as locations of service endpoints within parcels and demand requirements, a more accurate data model can be generated by plugging into the code, modeling requirements specific to the required network.

4.3 Generation based on engineering constraints

The topological models described in Chapter 3 are based on the assumption of inaccessibility to information and hence depend upon random data generation techniques. The models being generic and with topology predominantly related to the spatial orientation of geometric features, it is sufficient to generate data that is geographically referenced without the need for any non-spatial information pertaining to infrastructure objects in the model. Although it is possible to produce a model conforming to a standard set of topological constraints which primarily govern the spatial relationships of features, the model cannot be considered optimal relative to an actual infrastructure model, without knowledge of the engineering properties relevant to the scenario for which it is

being generated. These attributes play a key role in defining an objective and form the basis for a constraint based topological model.

The water distribution network model implements topological rules of proximity and adjacency between service laterals and land parcels but does not take into account the engineering properties of the network such as diameter, material, age of the pipes or even the types of soil around the study area. Figure 4.7 depicts attributes of pipes represented as line features.

FID *	Shape *	MATERIAL	DIAMETER
183	Polyline	Ductile iron	20"
184	Polyline	Ductile iron	20"
185	Polyline	Ductile iron	20"
186	Polyline	Ductile iron	20"
187	Polyline	Ductile iron	20"
190	Polyline	Cast iron	12"
192	Polyline	Cast iron	8"

Figure 4.9: Engineering properties of pipes

These non-spatial properties stored as attributes in the respective feature classes besides playing a key role in spatial analysis can have a significant impact on the topological organization of feature geometries by introducing constraints such as distribution mains represented as edges with different diameters or materials can only connect via junctions and fire hydrants can only connect to a hydrant lateral but not a service lateral. To ensure the flow of water between water mains and service laterals which are usually of different diameters requires a junction or reducer valve to maintain network connectivity and flow.

These constraints, also referred as network connectivity rules in the GIS domain, are not just limited to water distribution networks but can similarly be applied to other infrastructure models such as electrical lines, gas pipelines, telephone services and any network model that aids in the flow of resources.

Water distribution networks are often designed with a primary objective of meeting demand from consumers. The huge amount of information involved in creating such a network can be categorized into three important groups - customer information, data pertaining to infrastructure elements and geographical information of customers and infrastructure. Demand is a constantly varying parameter depending upon the type of land use – residential, commercial or industrial. These consumer types and their geographic locations significantly impact the placement and types of water infrastructure elements specifically pipes, pumps and valves. From a topological perspective, valves are the junctions that connect and regulate flow between pipes. Common pipe characteristics such as diameter, material and length are dependent upon consumption, flow requirements, pressure levels, durability, cost and their respective function within the pipe network. Pipes that serve as distribution mains often run along streets and are larger conduits of flow than service laterals which move water from the mains to consumers. Due to fluctuations in water demand between adjacent customers, service laterals may be of different sizes. The network connectivity between distribution mains and service laterals can be characterized by a trunk topology. Figure 4.8 illustrates connectivity and diameters of a distribution main and service laterals.

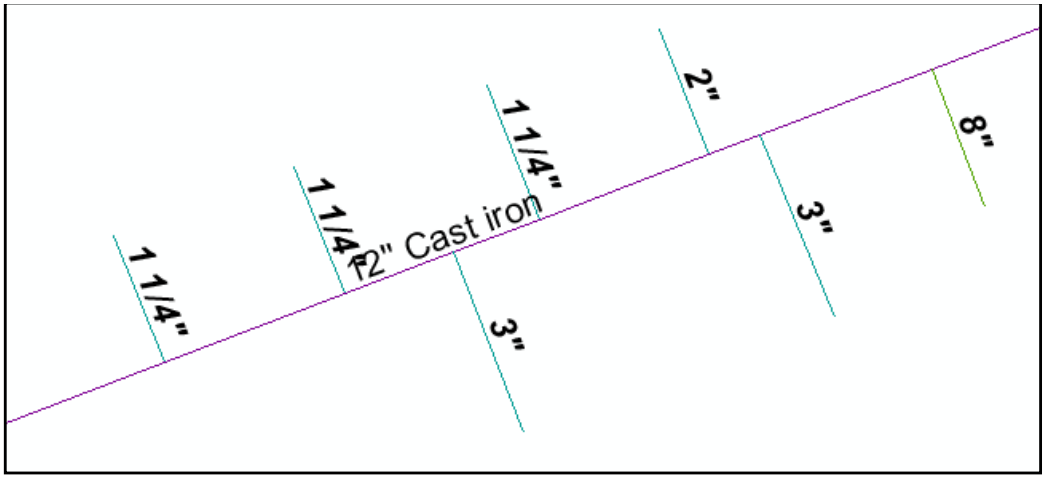

Figure 4.10: Diameters of distribution main and service laterals

Generating a topological model that is in close resemblance to an actual water network topology on the ground not only requires meeting water demand but must also take into consideration other governing factors such as cost and reliability. The generated water network can be considered as a topology comprised of a number of smaller topological networks that are dependent upon properties of individual infrastructure elements. At locations with equal demand from multiple consumers, use of laterals with equal sizes and materials may be considered to lower costs which can subsequently alter network layout. With proper evaluation and configuration of properties of pipes and valves, a topological model can be generated that represents an actual water distribution network.

Commercial GIS packages inherently handle connectivity relationships through rules which constrain the type of features that connect with one another or a number of features of one particular type that can be connected to a specific group of features of

another type. Two types of connectivity rules are implemented: edge – junction rules and edge – edge rules. An edge – junction rule dictates how an edge may connect to a junction, while an edge – edge rule establishes connectivity between two edges through junctions. Through the use of subtypes which are essentially a subset of features in a feature class that share similar attributes, connectivity rules can be applied between two feature classes or subtypes within the same edge feature class. In the water network example, ten inch and eight inch transmission mains represented as edges can be grouped into subtypes and connected via a subtype of reducer valves in the valves feature class.

The geoprocessing framework does not support creating connectivity rules for utility networks programmatically. In order to implement these constraints in Python, requires development of custom functions capable of simulating connectivity and accessing feature attribute information to determine the changes in geometric association between. Using existing methods such as retrieving geometry and accessing specific attribute data provided by the geoprocessing framework, custom code can be developed to enforce restrictions such as pipeline features of one specific diameter can only connect to another pipe smaller in diameter through a junction. Programmatically, this can be achieved by first retrieving all features of the required diameter and iterating through each pipe, a buffer is generated and a geometry intersection operation is executed to determine adjacent geometries. The pipe diameter of adjacent features is retrieved and checked if the pipe diameter is less than that required. The condition once met, the feature is identified and a junction represented by a point feature is inserted into the

network. Similarly, multiple constraints can be addressed making use of the engineering properties which subsequently results in the creation of close to real topological model.

4.4 Dynamic segmentation

Dynamic segmentation is the process of calculating map locations of point or line events stored in a table relative to a line feature using a linear referencing measurement system and displaying them on a map. It is built on the concept of avoiding splitting of line features into segments based on attribute values. Multiple sets of attributes can be associated with any portion of the linear feature irrespective of where it begins or ends. From the description of dynamic segmentation, it is evident that the integrity of underlying topology remains unaffected.

From a water distribution perspective, this process can be used to maintain attribute information describing characteristics of the pipeline segments such as quality, material and diameter without splitting the pipe network. Positions of service and hydrant laterals relative to a distribution main linear feature can be linearly referenced and their locations computed. Point event locations such as those of system valves and service laterals use only a single measure value to describe their location, whereas line events such as diameter and material of pipe make use of from- and to- measure values to describe the portion of the pipe they associate with. Dynamic segmentation plays a key role in large scale pipeline infrastructure systems to track operational conditions and hazard prone regions in the network.

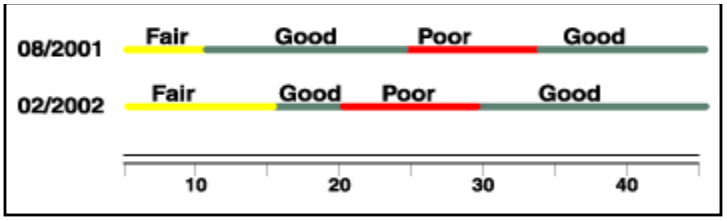

Figure 4.11: Operating conditions of a road or pipe recorded over time [19]

4.5 Summary

The objective of generating a topological model for a water distribution network using the principles and tools described in Chapter 3 is highlighted in this chapter. A reference model is selected to compare the output of the programmatically generated water network topology from land parcels and distribution mains. Service lines for each parcel from utility mains are generated to ensure water flow with the assumption that service junctions are located at the centroid of parcel polygons. Topological rules of proximity and adjacency ensure that service lines do not encroach into neighboring parcels and only connect to mains that are nearest. The topological violations which are intermediated results in the generation process are illustrated and the verification methods employed to rectify such situations are described.

The similarities between models and reasons for inconsistency in the number of service lines that connect to each main feature are described by using a graph. The availability of information such as demand values and location of service junctions within the parcel boundary are identified as factors that can result in the production of a more robust and accurate model. The following section describes the significance of engineering properties of network elements such as pipe diameters and lengths and

methods in which they can be programmed in the generation of a realistic model. The last section gives a brief overview of dynamic segmentation in the context of water networks.

CHAPTER 5

CONCLUSION AND FUTURE RESEARCH

5.1 Introduction

The problem of modeling spatial associations between arbitrarily generated infrastructure entities under conditions of non-availability of location specific data, and subsequently generating network topologies that are representations of the spatial organization between infrastructure facilities is addressed. Randomly generated spatial data is used as a substitute for lack of location information pertaining to infrastructure elements. The spatial dependence between elements or flow of resources between entities enables modeling the system as a network of connected elements. The objective is to create a programming toolkit capable of generating topologies that represent the spatial organization of infrastructure facilities into a network of connected elements. The tools are required to be used in the generation of a water distribution network model that resembles an actual network from insufficient data.

5.2 Assessment

Three possible spatial layouts in which infrastructure elements represented as point features complying with certain topological rules have been generated via a programmatic approach. The tools developed are capable of generating topological

models in the form of star, trunk and mesh networks from spatially random data. The process of generating a topology from random spatial data to generate network elements and the effects of topological constraints on the layout are described. The programming methodology involved use of GIS concepts and custom geometric libraries. Applications of these network topologies in the real world have been illustrated. Individual tools can be combined to produce more complex and realistic networks in which all three layouts coexist.

The tools are applied to a test case in which a water distribution network is generated from minimal data such as parcels and locations of distribution mains with the objective to connect service junctions to mains to ensure network flow is maintained. Certain rules that represent physical conditions on the ground are enforced to avoid erroneous topological layout. The resultant model is compared to a reference model to check for similarities and differences between the two models. Variations in connectivity at nodes and reasons for dissimilarities are addressed. Additional data that are required in the form of engineering properties to produce a more accurate and realistic model is described. Depending on the requirements and knowledge of modeling scenarios, topological constraints that govern and represent actual connectivity rules between facilities can be enforced to generate more accurate and genuine representations of networks.

5.3 Conclusion

Following are the conclusions that can be derived from this work on topological modeling.

- Infrastructure Topology Generator (ITG) is a toolkit comprising tools capable of generating realistic or probable topologies for infrastructure systems from random data in the absence of spatially complete datasets to model the geometric interactions and dependencies that exist between facilities.
- Unlike existing methods of topology generation as described in chapter 2 which only provide a theoretical approach and depend upon the availability of data, the programmatic approach presents a practical methodology of growing generalized network topologies from random spatial data which can be tailored to specific topological requirements.
- The implementation and programming of topological rules such as proximity and adjacency which impact the location of nodes relative to one another and connectivity through edges is elucidated.
- Engineering properties of network elements and topological constraints specific to a modeling scenario can be embedded into the code to modify the spatial layout of generated networks.
- GIS based commercial tools and custom spatial libraries provide a suitable framework for the creation, storage and visualization of individual and collective outputs of topological entities.

5.4 Future work

The tools developed provide a foundation upon which advanced topological modeling capabilities can be implemented. In order to enhance the accuracy of results produced by the star and topology tools, the code can be modified to support additional topological rules that are characteristic of full scale GIS software applications. These supplementary features incorporated within the code enable modeling of complex spatial relationships between infrastructure elements. The tools can also be customized with the properties and functions specific to different types of infrastructure systems. This built in intelligence can prove to be of enormous assistance in generating topological models that are equivalent to actual infrastructure system networks from completely arbitrary or limited data.

The advantage of developing the topology tools in Python which is an open source language provides the opportunity to integrate with the numerous custom libraries that are available. Topological modeling can be further enhanced by providing users with the ability to interact with the tools via a graphical interface. This presents users with the facility to feed data into the tools as information becomes accessible and in addition facilitates regulation of requirements that control the modeling output. The proposed developments significantly enhance the methods in which infrastructure systems are modeled as topological networks.

APPENDIX

INFRASTRUCTURE TOPOLOGY GENERATOR (ITG) CODE

In order to use these scripts, Python (Python Software Foundation, 2009), Shapely and ArcGIS (Environmental Sciences Research Institute, 2009) are required.

topoclasses.py – class module that generates random point coordinates

```
# Import module 'random'
import random

# Class definition
class randnodes:
    # Constructor
    def __init__(self):
        pass

    # Function to generate random nodes takes number of nodes required as argument
    def getRandomCoords(self, numberofnodes):
        self.noofnodesinlist = numberofnodes
        nodes = range(self.noofnodesinlist)
```

```
        # Create an empty list to store random generated coordinates
        self.coords = []
        for node in nodes:
            # Get x and y coordinate values within the specified range
            xcoord = random.randrange(0,200)
            ycoord = random.randrange(0,200)
            # Append generated coordinates to the list
            self.coords.append(str(xcoord) + "," + str(ycoord))
        # Return the coordinates list through the calling object
        return self.coords

    # Test function to print generated coordinates
    def prnt(self):
        self.getRandomCoords()
        print self.coords
```

StarTopology.py – Python script to generate star topology

```
# Import arcgisscripting, os, random and topoclasses modules
import arcgisscripting, os, topoclasses, random
# Create the geoprocessor object
gp = arcgisscripting.create()
# Set the workspace
```

```
gp.workspace = r"D:\Topology\gentopo.mdb"
# Create feature class to store random generated points
inFc = "star_nodes"
# If feature class already exists in the geodatabase delete it
if gp.Exists(inFc):
    gp.Delete(inFc)


try:
    # Create feature class by using the workspace, name of the feature
    # class and feature type
    gp.CreateFeatureClass(gp.workspace, inFc, "Point")
    # Get the insert cursor
    insCur = gp.InsertCursor(inFc)
    # Create a random object of class itg
    rnd = itg()
    # Get the coordinates generated into a list
    rndcoords = rnd.getRandomCoords(20)
    # Loop through each coordinate pair in the list
    for coords in rndcoords:
        # Create an arcgis point object
        pntObj = gp.CreateObject("point")
        # Set the point object's X and Y properties from the coordinate pair values
```

```
        pntObj.X = int(coords.split(',')[0])
        pntObj.Y = int(coords.split(',')[1])
         # Create a new feature
        pfeat = insCur.NewRow()
        # Set the shape of feature to the point geometry
        pfeat.shape = pntObj
# Insert the feature into the feature class
        insCur.InsertRow(pfeat)
except:
    print gp.GetMessages(2)


# Read feature class created from random points identify the geometry field
desc = gp.Describe("star_nodes")
shapefieldname = desc.ShapeFieldName
# Create search cursor
rows = gp.SearchCursor("star_nodes")
row = rows.Next()
# Create empty list to hold point feature coordinates
pnts = []
# Enter while loop for each feature
while row:
    # Create the geometry object 'feat'
```

```
    feat = row.GetValue(shapefieldname)
    pnt = feat.GetPart()
     # Append point coordinates to the list
    pnts.append([pnt.x,pnt.y])
    row = rows.Next()


# Choose a random point for the hub of the network
pointHub = random.choice(pnts)
# Create a point object to store coordinates of the hub of the network
strpnt = gp.CreateObject("point")
strpnt.x, strpnt.y = pointHub[0], pointHub[1]
# Remove the hub point from the list of points
pnts.pop(pnts.index(pointHub))
# Create the output feature class that will store the edges connecting points to hub
outfc = r"D:\Topology\gentopo.mdb\star_edges"
# Check if output feature class already exists
if gp.Exists(outfc):
    gp.Delete(outfc)
# Create the edge feature class
try:
    gp.CreateFeatureClass(os.path.dirname(outfc), os.path.basename(outfc), "Polyline")
    cur = gp.InsertCursor(outfc)
```

```
    lineArray = gp.CreateObject("Array")
    # Loop through points in the list and create an edge object
    # between every point and the hub
    for point in pnts:
        lineArray = gp.CreateObject("Array")
        lineArray.add(strpnt)
        pnt = gp.CreateObject("point")
        pnt.x, pnt.y = point[0], point[1]
        lineArray.add(pnt)
        feat = cur.NewRow()
        feat.shape = lineArray
        cur.InsertRow(feat)
        lineArray.RemoveAll()
except:
    print gp.GetMessages(2)
# Delete the cursors and geoprocessing object
del rows, cur, insCur, gp
```

TrunkTopology.py – Python script to generate trunk topology

```
# Import the required modules
import arcgisscripting, os, random, math, topoclasses
# From Shapely module import LineString object
```

```
from shapely.geometry import LineString


def readTestPoints(featclass):
    ''' Read points from a point feature class into a list '''
     # Create geoprocessor object
    gp = arcgisscripting.create()
    desc = gp.Describe(featclass)
    shape = desc.ShapeFieldName
    # Get a search cursor to loop through features in the feature class
    testpointsrows = gp.SearchCursor(featclass)
    testpointrow = testpointsrows.Next()
    # Create an empty list to hold point coordinates
    testpntslist = []
    while testpointrow:
        testpointfeat = testpointrow.GetValue(shape)
        testpnt = testpointfeat.GetPart()
        # Append point coordinates to the testpntslist
        testpntslist.append([testpnt.x, testpnt.y])
        # Get the next row
        testpointrow = testpointsrows.Next()
    # delete the geoprocessor and search cursor
    del gp, testpointsrows
```

```
    # return the list with point coordinates to the caller
    return testpntslist


def createFeatureClass(featclass, feattype):
    ''' Generalized function to create a new feature class with parameters
    feature class name and type '''
    # Create a geoprocessor object
    gp = arcgisscripting.create()
    # Delete if feature class already exists
    if gp.Exists(featclass):
        gp.Delete(featclass)
    # Create a new empty feature class with function parameters
    gp.CreateFeatureClass(os.path.dirname(featclass),os.path.basename(featclass),
feattype)
    # Create an insert cursor to add new features to the feature class
    insertCursor = gp.InsertCursor(featclass)
    # Return the cursor to the calling object
    return insertCursor
    # Delete the geoprocessor object
    del gp
```

```
def createTrunkList(testpnts, noofpointsintrunk):
    ''' Create a list of points chosen randomly that make up the trunk '''
    # Random selection of points from the points list
    bool = True
    while bool:
        # Create an empty list
        trunk = []
        for n in range(len(testpnts)):
            # Get a random point
            trunk_pnt = random.choice(testpnts)
            print "Trunk Point: ", trunk_pnt
            # Check to see if the random point already exists in the list
            if trunk.count(trunk_pnt) == 0:
                trunk.append(trunk_pnt)
            # Break out of the loop when points in list match required
            # number of points in trunk
            if len(trunk) == noofpointsintrunk:
                break
        # Check trunk for any feature intersections
        bool = checkTrunkValidity(trunk)
        # If trunk intersects with itself loop back to get new set of points for the trunk
        if bool is True:
```

```
            continue
    # Return the trunk points as a list to the caller
    return trunk


def checkTrunkValidity(trunklist):
    ''' Function that implements a check on trunk intersecting itself '''
    lines = []
    # Create line geometries with points in the trunk list and append to lines[]
    for i in range(len(trunklist)-1):
        line = LineString((trunklist[i], trunklist[i+1]))
        lines.append(line)
    for i in range(len(lines)-2):
        for j in range(i+2, len(lines)):
            print i, j
            bool = lines[i].crosses(lines[j])
            print bool
            if bool is True:
                return bool


def createTrunkFC(trunklist):
    ''' Create a new feature class to hold the trunk main feature '''
```

```
    # Send the output feature class and its feature type as parameters to the
    #createFeatureClass function and get the insert cursor
    trunk_inscur = createFeatureClass(r"D:\Topology\gentopo.mdb\trunk_main",
"Polyline")
    # Create the geoprocessor object
    gp = arcgisscripting.create()
    # Create an array object that will hold points that make up the line feature
    trunkArray = gp.CreateObject("Array")
    # Loop through points in the trunk list
    for p in trunklist:
        # Create a point object
        pt = gp.CreateObject("point")
        pt.x, pt.y = p[0], p[1]
        # Add points to the array
        trunkArray.add(pt)
    # Create a new row in the line feature class
    trunkmain_feature = trunk_inscur.NewRow()
    # Set shape of the line feature to array created
    trunkmain_feature.shape = trunkArray
    # Insert the line
    trunk_inscur.InsertRow(trunkmain_feature)
    # Delete objects
```

```
    del gp, trunkArray, trunk_inscur


def createFeaturesToTrunk(fpoint, tpoint, topocur):
    ''' Join features participating in the trunk to other point features in the point fc '''
    # Create the geoprocessor object
    gp = arcgisscripting.create()
    # Create an array object
    lineArray = gp.CreateObject("Array")
    # Create point objects
    fpnt = gp.CreateObject("point")
    fpnt.x, fpnt.y = fpoint[0], fpoint[1]
    tpnt = gp.CreateObject("point")
    tpnt.x, tpnt.y = tpoint[0], tpoint[1]
    # Add from and to points to the array
    lineArray.add(fpnt)
    lineArray.add(tpnt)
    # Create a new row for line feature
    feat = topocur.NewRow()
    feat.shape = lineArray
    # Insert the feature using the insert cursor
    topocur.InsertRow(feat)
    lineArray.RemoveAll()
```

```
    # Delete objects
    del lineArray, gp, topocur


if __name__ == "__main__":
    # Create random object of the itg class
    rnd = itg()
    # Get random coordinates generated
    rndcoords = rnd.getRandomCoords(20)
    try:
        # Create the geoprocessor object
        gp = arcgisscripting.create()
        # Create feature class and get the returned insert cursor
        insCur = createFeatureClass(r"D:\Topology\gentopo.mdb\trunk_nodes", "Point")
        # Loop through each of the randomly generated point coordinate values
        for coords in rndcoords:
            # Create an arcgis point object and set its X and Y coordinate attributes
            pntObj = gp.CreateObject("point")
            pntObj.X = int(coords.split(',')[0])
            pntObj.Y = int(coords.split(',')[1])
            # Create a new feature and set its geometry
            pfeat = insCur.NewRow()
            pfeat.shape = pntObj
```

```
            # Insert the feature
            insCur.InsertRow(pfeat)
        # Delete objects
        del insCur, gp
    except:
        print "Error creating point objects from random coordinates"
        print gp.GetMessages(2)

# Retreive the point geometries of features generated from random points into a list
pointslist = readTestPoints(r"D:\Topology\gentopo.mdb\trunk_nodes")
# Get the list of points participating in the trunk
trunklist = createTrunkList(pointslist, 7)
# Loop through each of the trunk points and remove it from the total points list
for tstpnt in trunklist:
    pointslist.remove(tstpnt)

# Create trunk feature class by passing the trunk points list to function createTrunkFC
createTrunkFC(trunklist)
# Create the output feature class to contain edges generated and get the insert cursor
topocur = createFeatureClass(r"D:\Topology\gentopo.mdb\trunk_topo", "Polyline")
# Create an empty list to hold distance values
distances = []
```

```
# Loop through points list excluding trunk points
for point in pointslist:
    # For every point in trunk
    for trpoint in trunklist:
        # Calculate distance
        dist = math.sqrt(pow(point[0]-trpoint[0],2) + pow(point[1]-trpoint[1],2))
        # Append the distance values to distances list
        distances.append(dist)
    # Determine least distance
    mindist = min(distances)
    # Determine the trunk point that is nearest
    seltrnkpnt = trunklist[distances.index(mindist)]
    # Send the point, nearest trunk point and insert cursor to
    # the createFeaturesToTrunk function to create edge features
    createFeaturesToTrunk(point, seltrnkpnt, topocur)
    # Empty the distances list
    distances = []
```

MeshTopology.py – Python script to generate mesh topology

```
import win32com.client, topoclasses, os, arcgisscripting

def createFeatureClass(featclass, feattype):
    ''' Generalized function to create a new feature class with parameters
    feature class name and type '''
    # Create a geoprocessor object
    gp = arcgisscripting.create()
    # Delete if feature class already exists
    if gp.Exists(featclass):
        gp.Delete(featclass)
    # Create a new empty feature class with function parameters
    gp.CreateFeatureClass(os.path.dirname(featclass), os.path.basename(featclass),
feattype)
     # Create an insert cursor to add new features to the feature class
    insertCursor = gp.InsertCursor(featclass)
    # Return the cursor to the calling object
    return insertCursor
    # Delete the geoprocessor object
    del gp
```

```
def readTestPoints(featclass):
    ''' Read points from a point feature class into a list '''
    # Create geoprocessor object
    gp = arcgisscripting.create()
    desc = gp.Describe(featclass)
    shape = desc.ShapeFieldName
    # Get a search cursor to loop through features in the feature class
    testpointsrows = gp.SearchCursor(featclass)
    testpointrow = testpointsrows.Next()
    # Create an empty list to hold point coordinates
    testpntslist = []
    while testpointrow:
        testpointfeat = testpointrow.GetValue(shape)
        testpnt = testpointfeat.GetPart()
        # Append point coordinates to the testpntslist
        testpntslist.append([testpnt.x, testpnt.y])
        # Get the next row
        testpointrow = testpointsrows.Next()
    # delete the geoprocessor and search cursor
    del gp, testpointsrows
    # return the list with point coordinates to the caller
    return testpntslist
```

```
def main():
    rnd = randnodes()
    rndcoords = rnd.getRandomCoords(5)
    try:
        # Create the geoprocessor object
        gp = arcgisscripting.create()
        gp.overwriteoutput = 1
        # Create feature class and get the returned insert cursor
        insCur = createFeatureClass(r"D:\Topology\gentopo.mdb\mesh_nodes", "Point")
        # Loop through each of the randomly generated point coordinate values
        for coords in rndcoords:
            # Create an arcgis point object and set its X and Y coordinate attributes
            pntObj = gp.CreateObject("point")
            pntObj.X = int(coords.split(',')[0])
            pntObj.Y = int(coords.split(',')[1])
            # Create a new feature and set its geometry
            pfeat = insCur.NewRow()
            pfeat.shape = pntObj
            # Insert the feature
            insCur.InsertRow(pfeat)
        # Delete objects
        del insCur
```

```
    except:
        print "Error creating point objects from random coordinates"

    #Retrieve created random point geometries into a list
    pointslist = readTestPoints(r"D:\Topology\gentopo.mdb\mesh_nodes")

    # Get an insert cursor for creating edges in a polyline feature class
    edgeInsertCur = createFeatureClass(r"D:\ Topology\gentopo.mdb\mesh_edges",
"Polyline")
    # Create an edge for every node in the list to other nodes
    for i in range(len(pointslist)):
        for j in range(i+1,len(pointslist)):
            linArray = gp.CreateObject("Array")
            pnt1 = gp.CreateObject("point")
            pnt1.x, pnt1.y = pointslist[i][0], pointslist[i][1]
            linArray.add(pnt1)
            pnt2 = gp.CreateObject("point")
            pnt2.x, pnt2.y = pointslist[j][0], pointslist[j][1]
            linArray.add(pnt2)
            feat = edgeInsertCur.NewRow()
            feat.shape = linArray
            edgeInsertCur.InsertRow(feat)
```

```
            linArray.RemoveAll()
    del gp, edgeInsertCur

if __name__ == "__main__":
    main()
```

testcase.py – Python script for water distribution network modeling

```
# Import all the required modules
import os, arcgisscripting
from shapely.geometry import LineString
from shapely.geometry import asPolygon
from shapely.geometry import asLineString
from operator import itemgetter
from shapely.geometry import Point

# Function definition for getParcelGeometry()
def getParcelGeometry():
    # Create arcgisscripting module
    gp = arcgisscripting.create()
    # Read the parcels feature class
    fc = r"D:\Topology\Landbase1.mdb\Export_Parcels"
```

```
    desc = gp.Describe(fc)
    shpfieldname = desc.ShapeFieldName
    # Get the search cursor
    rows = gp.SearchCursor(fc)
    row = rows.Next()
    # Create an empty dictionary for pacels
    parcels = {}
# Loop through each feature
    while row:
        # Create an empty coordinates lsit
        coords = []
        pfeature = row.shape
        partnum = 0
        partcount = pfeature.PartCount
        # If the parcel polygon is made up of multiple parts, get geometry for each part
        while partnum < partcount:
            part = pfeature.GetPart(partnum)
            pnt = part.Next()
            while pnt:
                # Append points of the part to the coords list
                coords.append([pnt.X, pnt.Y])
                pnt = part.Next()
```

```
            # Move to the next part
            partnum += 1
        # Create a shapely polygon object
        pa = asPolygon(coords)
        # Use the object id of the parcel polygon as key
        # to reference the shapely polygon geometry
        parcels[row.GetValue(desc.OIDFieldName)] = pa
        # Move to the next feature
        row = rows.Next()
 # Delete objects and return parcels to function caller
    del row, rows, pfeature, gp
    return parcels


# Function definition for createLineGeometry()
def createLineGeometry():
    # Create the geoprocessor object
    gp = arcgisscripting.create()
    # Absolute path of the location and name of centroids feature class
    fc = r"D:\Topology\Landbase1.mdb\parcelsCentroids"
    desc = gp.Describe(fc)
    shpfieldname = desc.ShapeFieldName
    # Get the search cursor on the feature class
```

```
    rows = gp.SearchCursor(fc)
    # Move to the first feature
    row = rows.Next()
    # Create empty dictionary to store line geometries
    lines = {}
    # Loop through each feature in the feature class
    while row:
        feat = row.GetValue(shpfieldname)
        pnt = feat.GetPart()
        # Create a Shapely LineString object from the centroid coordinate
        # and the x and y coordinate values of the nearest point
        line = LineString(((pnt.X, pnt.Y), (row.GetValue("NEAR_X"),
row.GetValue("NEAR_Y"))))
        # Use the object id of the parcel centroid to reference
        # the service line geometry created from the points
        lines[row.GetValue(desc.OIDFieldName)] = line
        # Move to next feature
        row = rows.Next()
    # Delete cursor, feature and geoprocessor objects
    del rows, row, feat, gp
    # Return the lines dictionary
    return lines
```

```
# Function definition for readTrunkUtilityGeometry
def readTrunkUtilityGeometry():
    # Function that retrieves utility main geometry
    gp = arcgisscripting.create()
    fc = r"D:\Topology\Landbase1.mdb\Export_Distribmains"
    desc = gp.Describe(fc)
    shpfieldname = desc.ShapeFieldName
    # Get search cursor
    rows = gp.SearchCursor(fc)
    row = rows.Next()
    # Create empty dictionary object to store trunk geometry
    trunk = {}

    # Loop through each feature
    while row:
        coords = []
        lfeat = row.GetValue(shpfieldname)
        partnum = 0
        partcount = lfeat.PartCount
        while partnum < partcount:
            part = lfeat.GetPart(partnum)
            pnt = part.Next()
```

```
            while pnt:
                coords.append([pnt.X, pnt.Y])
                pnt = part.Next()
            partnum += 1
        # Create a Shapely line object using utility feature coordinates
        li = asLineString(coords)
        trunk[row.GetValue(desc.OIDFieldName)] = li
        row = rows.Next()
    del row, rows, lfeat, gp
    return trunk


# Function definiton for getNear_Fid
# Function to get the object id of the nearest utility main feature
def getNear_Fid(k):
    gp = arcgisscripting.create()
    # The centroids feature class
    fc = r"D:\Topology\Landbase1.mdb\parcelsCentroids"
    desc = gp.Describe(fc)
    shpfieldname = desc.ShapeFieldName
    # Create a search cursor on the feature class
    rows = gp.SearchCursor(fc)
    row = rows.Next()
```

```
    lines = {}
    # Loop through centroid features
    while row:
        # If objectid equals function argument k break the loop
        if row.GetValue(desc.OIDFieldName) == k:
            return row.GetValue("NEAR_FID")
            break
        row = rows.Next()
    # Delete objects
    del rows, row, feat, gp


# Function definition for calculating point where
# service line intersects the utility main
def calculatePOI(k, pnt, street):
    cx = list(pnt.coords)[0][0]
    cy = list(pnt.coords)[0][1]
    ax = list(street.coords)[0][0]
    ay = list(street.coords)[0][1]
    bx = list(street.coords)[len(list(street.coords))-1][0]
    by = list(street.coords)[len(list(street.coords))-1][1]
    r_numerator = (cx-ax)*(bx-ax) + (cy-ay)*(by-ay)
    r_denominator = (bx-ax)*(bx-ax) + (by-ay)*(by-ay)
```

```
    r = r_numerator / r_denominator
    px = ax + r*(bx-ax);
    py = ay + r*(by-ay);
    p = [px,py]
    # Return the point
    return p


# Function definition for updating centroids feature class
# with new intersection coordinates
def updateFCCentroids(pois):
    # Create the arcgisscripting object
    gp = arcgisscripting.create()
    fc = r"D:\Topology\Landbase1.mdb\parcelsCentroids"
    desc = gp.Describe(fc)
    shpfieldname = desc.ShapeFieldName
    # Get the geoprocessor object
    rows = gp.UpdateCursor(fc)
    row = rows.Next()
   # Loop through features and update new near coordinates
    while row:
        for a, b in pois.iteritems():
            if (row.GetValue(desc.OIDFieldName) == a):
```

```
            row.near_x = b[0]
            row.near_y = b[1]
            rows.UpdateRow(row)
        row = rows.Next()
    # Delete objects
    del rows, row, gp

# Function definition for creating service laterals
def createLaterals(fcCentroids):
    # Create the geoprocessor object
    gp = arcgisscripting.create()
    desc = gp.Describe(fcCentroids)
    shapefieldname = desc.ShapeFieldName
    # Output feature class
    outSub = r"D:\Topology\Landbase1.mdb\parcelsLaterals"
    # If feature class already exists delete it
    if gp.Exists(outSub):
        gp.Delete(outSub)
 # Create the feature class using workspace, name and feature type parameters
    gp.CreateFeatureClass(os.path.dirname(outSub), os.path.basename(outSub),
"Polyline")
    # Get the insert cursor
```

```
insCur = gp.InsertCursor(outSub)
# Get the search cursor on the parcel centroids feature class
featCurCentroids = gp.SearchCursor(fcCentroids)
featCentroid = featCurCentroids.Next()
# Loop through the centroid features
while featCentroid:
    # Create a array object
    lineArray = gp.CreateObject("Array")
    pntfeat = featCentroid.GetValue(shapefieldname)
    parcelpnt = pntfeat.GetPart()
    # Add centroid geometry to the array
    lineArray.add(parcelpnt)

    # Create a new point object and set its geometry
    # from the centroids attribute values NEAR_X and NEAR_Y
    pnt = gp.CreateObject("point")
    pnt.x = featCentroid.GetValue("NEAR_X")
    pnt.y = featCentroid.GetValue("NEAR_Y")
    # Add the point to array
    lineArray.add(pnt)
    # Create a new line feature and set its geometry
    # Insert the feature in the feature class
```

```
        linefeat = insCur.NewRow()
        linefeat.shape = lineArray
        insCur.InsertRow(linefeat)
        lineArray.RemoveAll()
        # Move to next centroid feature
        featCentroid = featCurCentroids.Next()
    # Delete objects
    del insCur, featCentroid, featCurCentroids, gp

# Function definition to retrieve centroid geometries
def getCentroids(featclass):
    gp = arcgisscripting.create()
    rows = gp.SearchCursor(featclass)
    rows.Reset()
    row = rows.Next()
    centroids = []
    while row:
        # Use the feature's centroid property to get the value
        feat = row.shape
        # append the geometry to list
        centroids.append(feat.Centroid)
        row = rows.Next()
```

```
    # Delete objects and return the list
    del gp, rows, row
    return centroids

# Function definition for creating a feature class with centroids
def createCentroidsFc(centroids):
    gp = arcgisscripting.create()
    outfc = r"D:\Topology\Landbase1.mdb\parcelsCentroids"
    if gp.Exists(outfc):
        gp.Delete(outfc)
    gp.CreateFeatureClass(os.path.dirname(outfc), os.path.basename(outfc), "Point")
    insCur = gp.InsertCursor(outfc)
    for cnt in centroids:
        pnt = gp.CreateObject("point")
        pnt.x = cnt.split()[0]
        pnt.y = cnt.split()[1]
        pntfeat = insCur.NewRow()
        pntfeat.shape = pnt
        insCur.InsertRow(pntfeat)

    del gp, insCur, pnt
    return outfc
```

```
if __name__ == "__main__":
    # Set the parcels and main feature classes
    fc = r"D:\Topology\Landbase1.mdb\Parcels"
    fcUtility = r"D:\Topology\Landbase1.mdb\Distribmains"
    # Get centroids in a list
    lstCentroids = getCentroids(fc)
    # Create a feature class of centroid features by passing the list as parameter
    fcCentroids = createCentroidsFc(lstCentroids)
    # Proximity analysis tool near
    try:
        gp = arcgisscripting.create()
        gp.near(fcCentroids, fcUtility,"","LOCATION","")
        del gp
    except:
        print gp.GetMessages()


    #  Get the service line geometries
    lines = createLineGeometry()
    # Get geometry of parcels in a list
    polygons = getParcelGeometry()
    # Get the geometry of line features of trunk
    utility_main = readTrunkUtilityGeometry()
```

```
# Backup the utility main geometry list
utility_main_bkp = utility_main.copy()
# Create an empty dictionary to hold service line junction points on the trunk
pois = {}


# For every service line
for k, v in lines.iteritems():
    # For every parcel
    for i, j in polygons.iteritems():
        # If service line crosses parcel and their object ids do not match
        if v.crosses(j) and k != i:
            # Get the object id of the utility main the service line presently connects
            utilityToDiscard = getNear_Fid(k)
            # Get the parcel point
            pnt = Point(list(lines[k].coords)[0])
            # Delete the utility from list
            del utility_main[utilityToDiscard]
            # Empty dictionary of distances
            dictDist = {}
```

```
            # For every feature in the utility main feature class
            for s, t in utility_main.iteritems():
                # Get distance from centroid and include in the dictionary
                dist = pnt.distance(t)
                dictDist[s] = dist
            # Sort the dictionary and get the nearest utility
            listDict = sorted(dictDist.items(), key=itemgetter(1))
            shortestDistanceUtility = listDict[0][0]
            # Calculate the point where service line intersects by calling the
            # calculatePOI function and store it in a dictionary
            poi = calculatePOI(k, pnt, utility_main[shortestDistanceUtility])
            pois[k] = poi
            utility_main = utility_main_bkp.copy()
# Update the centroid features with new values
updateFCCentroids(pois)
# Create the output feature class with service laterals
createLaterals(r"D:\Topology\Landbase1.mdb\parcelsCentroids")
```

REFERENCES

REFERENCES

1. As cited in: The President's National Strategy for Homeland Security (2002)
2. Tolone et al. (2004). Critical Infrastructure Integration Modeling and Simulation. Retrieved November 25, 2009, from http://www.sis.uncc.edu/~anraja/PAPERS/ISI04.pdf
3. Gastner, M. T. (2006). Shape and efficiency in growing spatial distribution networks. Retrieved February 25, 2010, from http://www.cabdyn.ox.ac.uk/complexity_PDFs/ECCS06/Conference_Proceedings/PDF/p82.pdf
4. Wang, J. & Provan. G. (2009). A Comparative Analysis of Specific Spatial Network Topological Models. Retrieved April 5, 2010, from http://www.cs.ucc.ie/ccsl/GP-papers/2009/Wang_ICCS_2009_2.pdf
5. Wang, J. & Provan. G. (2008). Generating Application-Specific Benchmark Models for Complex Systems. Retrieved April 5, 2010, from http://www.aaai.org/Papers/AAAI/2008/AAAI08-090.pdf
6. Newman, M. E. J., Strogatz, S.H. & Watts, D. J. (2001). Random graphs with arbitrary degree distributions and their applications. *The American Physical Society, 64(026118).* doi: 10.1103/PhysRevE.64.026118

7. Geographic information system. (2009). *Wikipedia.* Retrieved December 2, 2009, from http://en.wikipedia.org/wiki/Geographic_information_system
8. TransCAD. (2009). Retrieved November 30, 2009, from http://www.caliper.com/tcovu.htm
9. TransCAD. (2009). *Wikipedia.* Retrieved November 30, 2009, from http://en.wikipedia.org/wiki/TransCAD
10. MIKE SWMM. (2009). Retrieved December 2, 2009, from http://www.dhi-italia.it/doc/home/Mike%20SWMM_colon_int-dhi.pdf?PHPSESSID=tvusa567rmc8kkrlbv43o8tga5
11. Storm Water Management Model (SWMM). (2009). *Urban Watershed Management Research*. Retrieved December 1, 2009, from http://www.epa.gov/ednnrmrl/models/swmm/index.htm
12. Mouse GIS. (2009). *GIS And Water Resource Modeling At DHI.* Retrieved December 1, 2009, from http://www.crwr.utexas.edu/gis/gishyd98/dhi/mouse/mousmain.htm
13. Geoprocessing [Flowchart]. (2007). *ArcGIS 9.2 Desktop Help.* Retrieved January 30, 2010, from http://webhelp.esri.com/arcgisdesktop/9.2/index.cfm?TopicName=What_is_geoprocessing?
14. Shapely. (2010). *GIS & Python & Invention*. Retrieved January 30, 2010, from http://trac.gispython.org/lab/wiki/Shapely
15. Integrated development environment. (2010). *Wikipedia.* Retrieved February 2, 2010 from http://en.wikipedia.org/wiki/Integrated_development_environment

16. IDLE (Python). (2010). *Wikipedia*. Retrieved February 4, 2010 from http://en.wikipedia.org/wiki/IDLE_(Python)

17. Barabasi, L. (2003). *Linked*. Cambridge, MA: Perseus Publishing

18. Holmes, B. J. & Scott, J. M. (2004). Transportation Network Topologies. Retrieved April 5, 2010, from http://www.airborneinternet.com/Docs/Holmes/ICNS_2004_Holmes_Scott.pdf

19. Multiple sets of attributes for road features [Graphic]. (2007). *ArcGIS 9.2 Desktop Help*. Retrieved January 30, 2010, from http://webhelp.esri.com/arcgisdesktop/9.2/index.cfm?TopicName=Some_linear_referencing_scenarios

CURRICULUM VITAE

Sriharsha Vankadara is a native of Hyderabad, India and obtained his B.E. (Hons) degree in Civil Engineering from Birla Institute of Technology and Science (BITS), Pilani in 2005. He worked as a GIS programmer and analyst for RMSI Pvt. Ltd., India on projects for Tele Atlas. While pursuing his M.S. degree, he worked as a Graduate Research Assistant on BIM and GIS integration and GIS applications in transportation modeling.