3-5-2007

Multi-Dimensional Range Querying using a Modification of the Skip Graph

Gregory J. Brault

MULTI-DIMENSIONAL RANGE QUERYING USING A MODIFICATION OF THE SKIP GRAPH

THESIS

Gregory J. Brault, Captain, USAF

AFIT/GE/ENG/07-04

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GE/ENG/07-04

MULTI-DIMENSIONAL RANGE QUERYING USING A MODIFICATION OF THE SKIP GRAPH

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Gregory J. Brault, B.S.E.E.

Captain, USAF

March 2007

AFIT/GE/ENG/07-04

MULTI-DIMENSIONAL RANGE QUERYING USING A MODIFICATION OF THE SKIP GRAPH

Gregory J. Brault, B.S.E.E.

Captain, USAF

Approved:

/ SIGNED /_____	5 Mar 07
Barry E. Mullins, Ph.D. (Chairman)	Date
/ SIGNED /_____	5 Mar 07
Christopher B. Mayer, Maj, USAF (Member)	Date
/ SIGNED /_____	5 Mar 07
Rusty O. Baldwin, Ph.D. (Member)	Date

Abstract

Skip graphs are an application layer-based distributed routing data structure that can be used in a sensor network to facilitate user queries of data collected by the sensor nodes. This research investigates the impact of a proposed modification to the skip graph proposed by Aspnes and Shah.

Nodes contained in a standard skip graph are sorted by their key value into successively smaller groups based on random membership vectors computed locally at each node. The proposed modification inverts the node key and membership vector roles, where group membership is computed deterministically and node keys are computed randomly.

Both skip graph types are modeled in Java. Range query and node mobility simulations are performed. The number of skip graph levels, total node count, and query precision are varied for query simulations; number of levels and total node count is varied for the mobility simulation. Query performance is measured by the number of skip graph messages used to execute the query while mobility performance is measured by the number of messages transmitted to maintain skip graph coherence.

When the number of levels is limited and query precision is low, or when query precision is matched by the number of levels in the skip graph and total network node counts are increased, the modified skip graph transmits fewer messages to execute the query. Furthermore, fewer update messages are needed to fix lost node references due to mobile nodes.

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Dr. Barry Mullins, for his guidance and support throughout the course of this thesis effort. I would also like to thank Capt Christopher Augeri, whose insight and guidance was invaluable throughout the research process.

Gregory J. Brault

Table of Contents

Page

List of Figures

List of Tables

Vita

Greg Brault was born in Warren, Michigan in 1980 and graduated from Henry Ford II High School in 1998. He was commissioned as a Second Lieutenant through the Air Force Reserve Officer Training Corps (AFROTC) at Michigan Technological University in May 2002. At Michigan Tech he received a Bachelor of Science degree in Electrical Engineering. His first assignment was with Air Force Research Laboratories, Space Vehicles Directorate at Kirtland AFB, Albuquerque, NM. There he performed basic research in satellite microelectronics and responsive space technologies, aiming to reduce the timeline for satellite deployment. Captain Brault was selected to attend the Air Force Institute of Technology (AFIT) and reported to Wright-Patterson AFB, Dayton, OH in March 2005. Captain Brault is assigned to the Air Force Information Warfare Center at Lackland AFB, San Antonio, TX upon graduation from AFIT.

MULTI-DIMENSIONAL RANGE QUERYING USING A MODIFICATION OF THE SKIP GRAPH

I. Introduction

Colonel Smith asks, “I need to know how many vehicles passed within a five square kilometer area centered at 33 20 N, 44 26 E within the last 24 hours.”

The lead Distributed Sensor Network (DSN) operator responds, “Right away, sir.” Five seconds pass after the operator inputs the query. “Looks like four tanks and nine trucks.”

This vignette is a glimpse into the future that illustrates the power and capabilities of DSNs. A DSN, in the context of this research, consists of tiny, battery-powered nodes equipped with a variety of sensors that periodically gather information about their surroundings, and either forward the data to surrounding nodes, or store it locally for future use.

The DSN operator in the above scenario issued a query to the network asking all nodes within a two-dimensional geographical bound to report any vehicle movement within the last 24 hours. In this example, the two dimensions being queried are latitude and longitude. Several issues must be addressed to efficiently execute multi-dimensional range queries such as this. Skip graphs can address some of those issues. This research investigates how a modification to a previously developed skip graph impacts multi-dimensional range query performance.

1.1 Motivation

An unyielding focus of the Department of Defense has been to increase our ability to gather as much situational information as fast as possible. Increased levels of information awareness equates to better decision making processes [PWW02]. Battlefield commanders privy to specific tactical information are better able to formulate battle plans. Satellites provide quite detailed information regarding troop movements and other in-theater information, given the proper orbit and sensors. If those conditions are not met, then there can be long lead times before a particular satellite can fulfill the requirements. DSNs provide a cheaper, faster alternative. DSNs can even provide more accurate information about the battlefield environment (audio, vibration) than a satellite can in certain situations.

There are a plethora of civilian applications for DSNs as well. Environmental and species monitoring, agriculture, production and delivery, and health care are just a few of the niches where this technology is deployable [RoM04].

1.2 Overview

Skip graphs are an application layer-based distributed routing protocol designed to process queries for a distributed network. The original skip graph [AsS03] searches for and queries a specific single-dimensional key or range of keys.

In a standard skip graph, each node in the network contains a key (e.g., geographical position) and is also associated with a membership vector (a random, arbitrary value). The membership vectors determine how nodes are grouped together which affects how fast a node can be found during a search.

An alternate method of organizing a skip graph is proposed in this research. Instead of grouping nodes based on randomized membership vectors, membership vectors are based on some multi-dimensional property. It is proposed that an alternative grouping will allow multi-dimensional queries such as a geographical range query to perform better.

This research implements both of these types of skip graphs in software, and investigates the impact the proposed change has on multi-dimensional queries. A corollary investigation is also performed on the impact the change has on mobility in a DSN.

1.3 Thesis Layout

This chapter introduces the topic of this research. Chapter 2 provides the background information, as well as related work in this area. Chapter 3 discusses the methodology used to configure and execute the simulations used to gather data. Chapter 4 analyzes and discusses the results of those simulations. Lastly, Chapter 5 presents the research conclusions and recommends avenues for further research.

II. Literature Review

This chapter presents relevant background information related to this research. Section 2.1 discusses DSNs and some of their applications. Section 2.2 presents skip lists and skip graphs, the distributed data structures in which the range queries execute. Section 2.3 discusses elements of distributed queries as they apply to skip graphs, including dimensional reduction techniques such as the *z*-order curve, and discusses how multi-dimensional queries are executed within the context of skip graphs. Section 2.4 discusses node mobility issues and how mobility impacts the integrity of a skip graph. Finally, Section 2.5 discusses related work forming the basis for skip graphs, and an application in which skip graphs are used.

2.1 Distributed Sensor Networks

In the context of this research, a DSN is a collection of independent, low-power devices distributed over some geographic area which have capabilities for inter-device communication. These nodes are populated with various sensors such as temperature, infrared/radio frequency, audio and vibration which collect information about the surrounding environment. These nodes record and store data for a period of time and then distribute that information to surrounding nodes at a later time. Data is transmitted within the network wirelessly using a routing protocol such as Ad-hoc On-demand Distance Vector (AODV) [PBD03].

Many challenges must be overcome to utilize the full potential of these networks. The challenge this research focuses on is how to index the massive amount of data collected by a large network so it may be efficiently queried.

These networks are assumed to be so large that it is too costly to collect and process each node's data at a central, external location. While an external location may be viable for low node count networks, it does not scale [Aug05]. It is unreasonable for a single node to collect and store all of this data considering these networks can be quite large, on the order of thousands of nodes. Therefore, efficient mechanisms must be in place to index and query this data in a timely manner.

2.2 Skip Lists and Skip Graphs

Skip lists and skip graphs are application layer-based distributed data structures which are searched for a key or a set of keys. A key at a node represents a piece of information associated with that node. The key can be scalar or multi-dimensional. An example of a scalar key is the output of a temperature sensor; the key value is directly mapped to the temperature data. An example of a multi-dimensional key is the geographical position of a node; the latitudinal and longitudinal coordinates comprise the two dimensions of the node's position.

A single dimensional range query might inquire about any nodes with temperature sensor readings between two values. A multi-dimensional range query might inquire about any nodes with geographical coordinates within a certain latitudinal-longitudinal area, as in the scenario in Chapter 1.

Skip lists and skip graphs are an extension of a more common data structure, the linked list. There are two types of linked lists, singly- and doubly-linked. A singly-linked list is a series of nodes linked to a single adjacent node via a reference to that node's location. In the doubly-linked list, nodes not only have a reference to a successor node,

but also to a predecessor node. Figure 1 is a graphical representation of a doubly-linked list. In contrast to a fixed-length data structure such as an array, linked lists can be dynamically resized by simply modifying affected nodes' references to other nodes. This method is used for both node insertion and deletion.

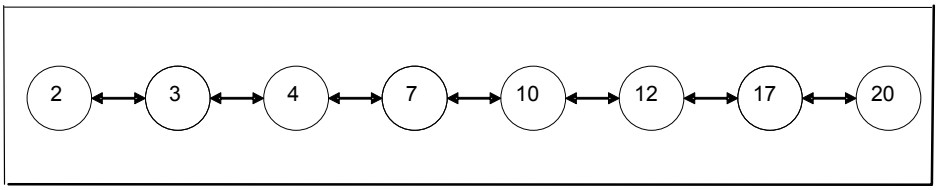


Figure 1. Doubly-Linked List

2.2.1 Skip Lists

A skip list is a balanced tree data structure organized as an aggregation of levels each containing an increasingly sparse ordered doubly-linked list [Pug90]. The base level L_0 contains just one ordered linked list complete with all nodes (as in Figure 1), and in each successive level *i*, each node in level *i*-1 appears in level *i* independently with some fixed probability *p*.

Figure 2 shows a skip list. Each circle represents a node, and the number shown inside each node represents both the node's identification and key value. Horizontal links between nodes represent application-layer links in the physical sensor network. When a link exists between two nodes, those two nodes are able to communicate with each other using the routing protocol specified for the sensor network's routing layer. For example, in L_1, node 7 maintains references to nodes 3 and 17, whereas in L_0, node 7 maintains references to nodes 4 and 10, respectively. For a particular node, maintaining several

links to different nodes at different skip list levels is beneficial when key searches are performed, as the following sections discuss.

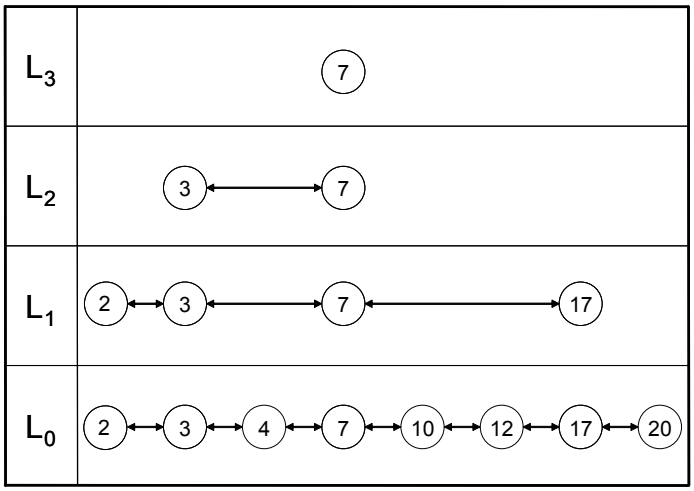


Figure 2. Skip List

Nodes do not maintain full copies of the skip list. As Figure 2 illustrates, node 20 does not have any knowledge about the link between nodes 2 and 3. If a search is received by node 20 for a key of 3, node 20 simply forwards it on to node 17, because 3 is less than 17. Thus, node 20 can be assured the search is advancing in the proper direction because the lists are ordered.

Skip lists are distributed data structures and so as a search is being executed, the search parameters are transferred from node to node. Each transfer of the search is meant to locate a node that contains the closest match to the search parameter.

Linked lists at higher levels offer “express lanes” to traverse the full list located at L_0 [AsS03]. A search is issued to a node in the highest level and continues to transfer within a linked list on the same level until one of three conditions is met. The following paragraphs explain each of these conditions.

The first condition arises if a search arrives at a node at a particular level and that node is at the end of the list. The node still needs to forward the search further in the appropriate direction, but it cannot because it does not have a reference to a node in the direction that the search needs to proceed. For example, suppose a search for a key of 4 issued at node 7 in the skip list of Figure 2. The initial level associated with the search is the top-most level, L_3. Node 7 has no neighbor node references at level 3, and so the search remains at node 7, but continues at the next lower level, L_2.

The second condition arises if a search is supposed to transfer to a neighbor node, but it is evident that the key associated with the neighbor node is beyond the requested search parameter. Consider the previous search for a key of 4. After the search drops down from L_3 to L_2, it still resides at node 7. At L_2, node 7 now has a node reference in the direction that the search should progress in, because the search parameter of 4 is less than the key at node 7. However, this neighbor node has a key of 3, which would advance the search too far. If this is not handled properly, the search will loop between nodes 7 and 3. To alleviate this, the search drops to the next lowest level, L_1.

The third and final condition is if the search finds a node with a key matching the search parameter. At this point, the search has completed. Continuing our search for key 4, we can see that the second condition is met again at L_1, and so the search is forced to drop down to the next and lowest level, L_0. In this level, node 7 has a new neighbor node in the direction that the search should progress in, and this node happens to have the key that is being searched for. Consider another search for a key of 5. The entire search process would have been identical up to this point. However, in this example, since the

search is in L_0 and the second condition has been met (node 4 is beyond the search parameter), then the search can determine that the slot between nodes 4 and 7 in the base list L_0 is where the search *should* end. This information is needed when nodes are added to a skip list; the inserted node needs to know where in the base list it is being added (i.e., who its neighbors are).

Consider another brief example of a search for a key of 2 in the skip list shown in Figure 3. This figure is the exact same skip list of Figure 2, but is annotated with arrows to clarify which level the search is being transferred to. The search begins at node 7 in the top-most level L_3, and since node 7 is the only node in that level, the search must drop down to L_2. In L_2, node 7 has a new reference to node 3. At this point, the search transfers over to node 3 because a key of 2 is being searched, and since the nodes are ordered in each list at each level, the search can determine that is the proper direction to progress. After the search has transferred to node 3, it must drop down to L_1 because node 3 is at the end of the list. At this point, node 3 now has a reference to node 2, and so the search transfers over. It can stop here, because node 2 contains the key that is being searched. On average, no more than $\frac{1}{1-p}$ nodes are searched on average per level, giving an average search time of [AsS03]

$$O\left(\log n \frac{1}{(1-p)\log\left(\frac{1}{p}\right)}\right) \tag{1}$$

where *p* is the probability of node in level *i* appearing in level *i*+1, and *n* is the total number of nodes in the skip list.

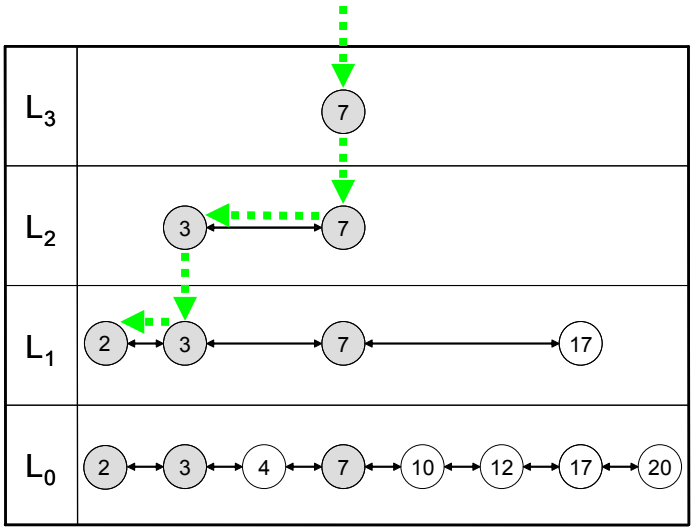


Figure 3. Searching For '2' In A Skip List. Grey Nodes Are Involved In The Search

Skip lists perform well as a local data structure, but they pose significant drawbacks for a distributed implementation such as a sensor network. A skip list lacks redundancy, which reduces overall robustness – a desirable sensor network feature. In our example, if node 7 fails, all communication links at all levels in the skip list are severed. The search will not be able to progress anywhere because node 7 was essential in maintaining large portions of the skip graph at the lower levels. That node becomes a single point of failure for the network, a situation that should be avoided.

Contention issues also arise due to the existence of relatively few nodes in higher levels of the skip list. In our example above, node 7 is a hot spot that is a central access point for all searches. This will quickly diminish the power levels for that particular node

since any search must be transmitted through this node – another undesirable feature in sensor networks.

2.2.2 Skip Graphs

A solution to these drawbacks is a skip graph. Figure 4 shows a skip graph that parallels the skip lists of Figure 2 and Figure 3. A skip graph is similar to a skip list in that L_0 still contains a single ordered linked list containing all of the nodes in the graph. However, a skip graph diverges from a skip list as higher levels are constructed.

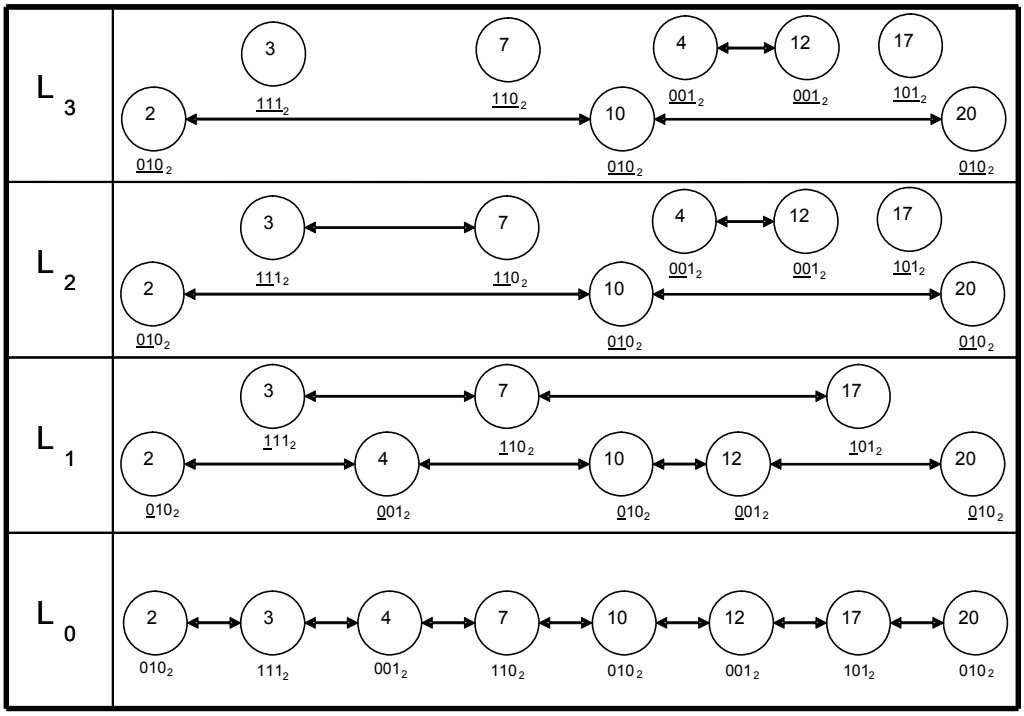


Figure 4. Skip Graph

In a skip graph, all nodes exist at all levels. There may be multiple linked lists at level $i > 0$, and every node is in one and only one of these lists at any particular level. A membership vector *m(x)* holds which list node *x* belongs to at a particular level *i*.

The membership vector *m(x)* is a bit sequence, computed randomly and independently within each node. A node keeps the same membership vector at each level. At the lowest level, every node is contained in the same linked list, and therefore the notion of a membership vector has no impact on which linked list a node resides in. However, as the number of levels increase, nodes are grouped into linked lists based on a membership vector prefix.

Consider the skip graph illustrated in Figure 4. L_1 separates L_0's linked list into two lists, where the first bit in each node's membership vector is the bit that determines which list that node belongs. L_2 takes this one step further, separating nodes into groups of linked lists with the first two bits of the membership vector determining which of the groups the node belongs. For example, in Figure 4 at L_2, node 3 and node 7 are contained in the same linked list, because the first two bits of both of their membership vectors are 11_2.

Searching for a key value in a skip graph is essentially the same process as it is in a skip list. In a skip graph, however, the search may start at *any* node in the top-most level. This alleviates top-level contention issues that pose a significant hot spot threat to nodes in a skip list. Figure 5 illustrates how there are at least three different paths for a search for a key value of 2 to find its way to the destination, with each path starting at a different node in the top level L_3. The solid line route begins the search at node 7, and takes two hops (node 7→node 3→node 2). The rectangular dotted line route begins at node 10, and happens to find the search parameter immediately, taking only one hop

(node 10→node 2). The circular dotted line route begins at node 12, and takes two hops to find node 2 (node 12→node 4→node 2), the same as the solid line route.

Skip graphs also alleviate single points of failure that existed in skip lists [AsS03]. Figure 5 illustrates that if node 7 were to fail, a new search could be initiated at any other node in the top level; the previous examples detailed searches beginning at node 7, node 10, or node 12.

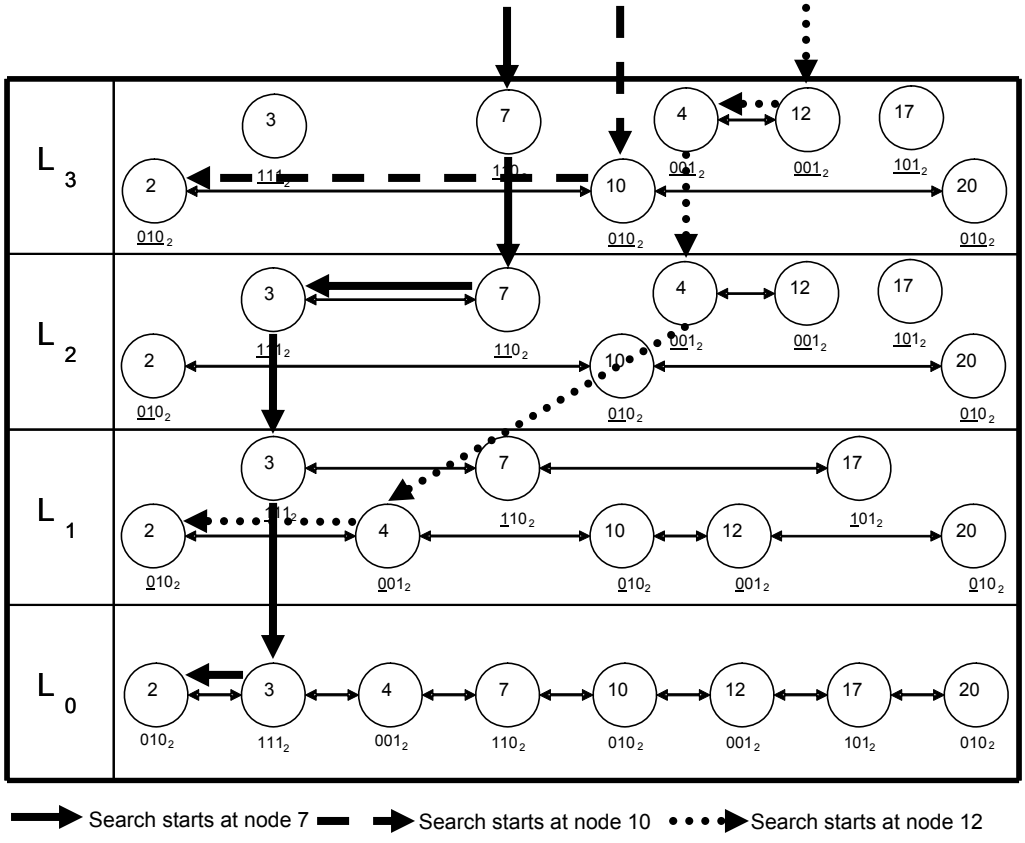


Figure 5. Searching For '2' In A Skip Graph

2.2.3 Multi-Dimensional Skip Graph

Section 2.2.2 presented the construct and searching mechanisms of a skip graph. That version of a skip graph will herein be referenced as a standard skip graph. Recall

that in a standard skip graph, each node is associated with a key value and its membership vector. The key value is single-dimensional, and may be derived directly from a scalar sensor reading such as temperature, or a linearization of a multi-dimensional attribute such as the node's geographical location. Section 2.3.1 describes one of these linearization techniques – the *z*-order curve. Also recall that nodes in a standard skip graph compute their own random membership vector *m(x)* independently, and it is this vector that determines which linked-list a particular node is a member of for each level.

This research modifies the standard skip graph, herein referenced as a multi-dimensional skip graph. While a standard skip graph can search for multi-dimensional keys (linearized into a single dimension using the *z*-order curve), nodes are grouped into random membership vectors. Nodes in a multi-dimensional skip graph compute a random key value, but the membership vectors are computed deterministically based on multi-dimensional information. For example, the membership vector may be computed from the linearization of the two-dimensional geographical position information; in a standard skip graph this information would have been used to compute node keys. The random key values computed by nodes in a multi-dimensional skip graph are synonymous to the randomly computed membership vectors of a standard skip graph. The randomization of key values is needed because multi-dimensional skip graphs no longer have randomized membership vectors. Some form of randomization is needed in any type of skip graph as specified in [AsS03]. Essentially, the roles of key values and membership vectors have been reversed.

A primary goal of this research is to determine whether this role reversal impacts query performance. If a majority of the queries issued in a DSN are of a multi-dimensional nature, such as returning temperature readings from all nodes within a certain geographical boundary, then grouping nodes within each level of the skip graph based on that multi-dimensional data may decrease the time it takes to find the set of nodes pertaining to the query.

2.3 Distributed Querying

A primary reason for sensor networks is to gather data. A query is injected into the network at an arbitrary node, and that query must propagate throughout the network and reach all of the nodes that should respond to that query. A skip graph facilitates these queries. Each node maintains its portion of the skip graph independently. When a query arrives at a particular node, that node forwards it on to other nodes based on the level the query is presently at (like the level of a key search as discussed in Section 2.2.1) and who the neighbors are for that node at that level. Section 2.3.1 presents background information regarding *z*-ordering, a type of Space-Filling-Curve (SFC). Sections 2.3.2 and 2.3.3 present how standard and multi-dimensional skip graphs execute range queries.

2.3.1 Space-Filling Curves

To simplify analysis, many disciplines use dimensional reduction techniques to linearize a *k*-dimensional space to a single dimension. This allows for all points in a two-dimensional or three-dimensional space to be “flattened” into a single linear series of points with a total ordering. A bounded 2-D geographical area may be linearized by drawing a continuous line throughout the bounds, thereby “filling the space” [Aug06].

There are several methods for traversing a bounded area to linearize a 2-D space. The simplest method is a straightforward concatenation of the two dimensions, and then sorting the resulting values. For instance, Figure 6 shows what the SFC would look like when using this simple sweep, or raster-scan method of 2-D linearization.

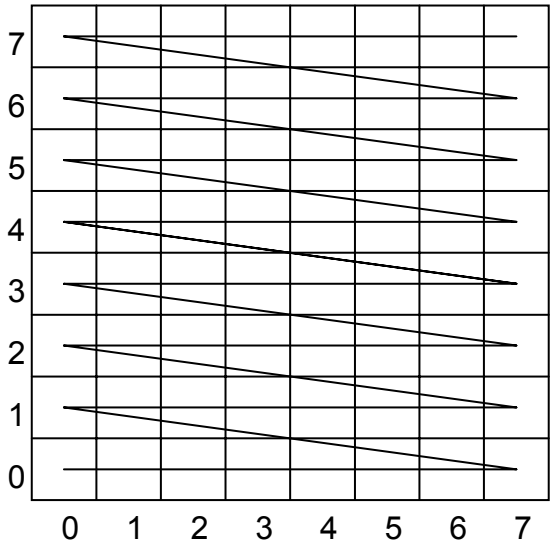


Figure 6. Sweep/Raster-Scan Space-Filling Curve

Note that the sorted concatenation of each of the box values follows the sweeping line, starting from the bottom: 00,01,…,06,07,10,11,…,16,17,… and so forth. While this is a simple and quick way to linearize a 2-D space into 1-D scalar values, there are other algorithms better suited for a geographical indexing application.

Figure 7 shows another SFC, the *z*-order. Its name stems from the *z*-like pattern it makes as it traverses the bounded area. This hierarchical curve is also called a Lebesgue curve, named after the French mathematician Henri Lebesgue (1875-1941) [Haw90]. Figure 7 shows how the *z*-pattern possesses similarity as one zooms in or out. This is an important feature and is discussed later in this section. A rigorous mathematical analysis of this curve can be found in Chapter 5 of [Sag94].

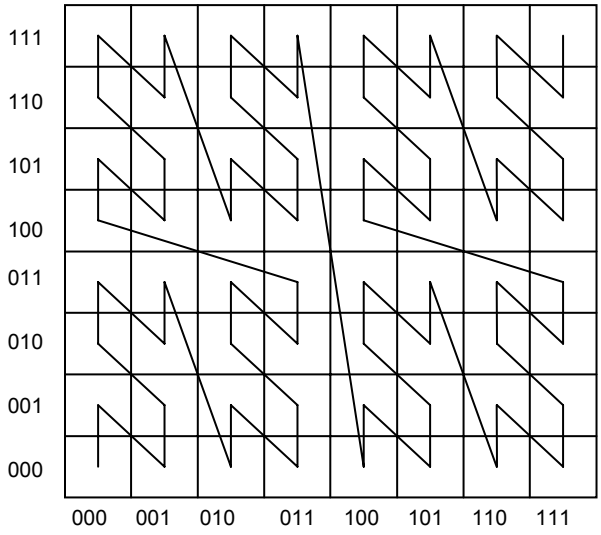


Figure 7. *Z*-order Space-Filling Curve

The *z*-order scalar value is used throughout this research to represent the geographical position of the nodes in the skip graph. A feature of the *z*-order curve is that the points along the line are constructed by interleaving the bits of the each of the dimensions. Figure 8 shows how the point (3, 0) can be expressed as a linearized *z*-order value. The axes in Figure 7 are listed in binary to help illustrate this process.

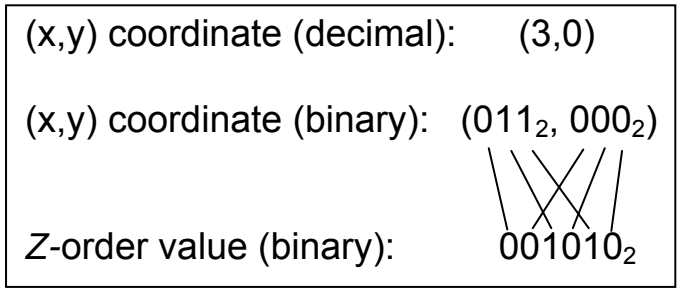


Figure 8. *Z*-order Bit Interleaving

Starting at the bottom-left grid location and following the line yields an interleaved count of 000 000, 000 001, 000 010,…,011 111, 100 000,…, 111 111. So, as

Figure 8 illustrates, a node within the coordinate grid of (3,0) will have a *z*-order value of 011 (3) interleaved with 000 (0) with a result of 00 10 10.

This interleaved value is the most significant bits of the *z*-order value of any node located within that grid location. If higher precision geographical position information specifies a node further, any node within that coordinate grid will have a *z*-order beginning with that prefix. This is an important property as the queries executed in this research return all nodes with a particular *z*-order prefix.

A *z*-order curve is hierarchical. For instance, the range of coordinates for the bottom left quadrant of the grid in Figure 7 is x:= (000 – 011), y:= (000 – 011), which equates to a *z*-order range of ($\underline{00}0000_2$ – $\underline{00}1111_2$). The common prefix in this range of values is 00_2. Therefore, any *z*-ordered coordinate value of a node whose prefix matches that value falls within that quadrant. Likewise, the range of the top-right sub-quadrant of this quadrant is x := (010 – 011), y := (010 – 011), which equates to a *z*-order range of ($\underline{0011}00_2$ – $\underline{0011}11_2$). The common prefix here is 0011_2. This illustrates an interesting property of *z*-order ranges. To find the prefix of any quadrant, you can hierarchically build it one sub-quadrant at a time.

For example, Figure 9 shows three levels of the hierarchy. In this example, to build the *z*-order prefix for a node having coordinates within the small square at (001,110), start with the highest level quadrant **01** (upper left-hand quadrant of the full grid). Then, recursively expand into the appropriate quadrants until you reach the level that matches the length of the desired prefix. In this example, the next level's quadrant is **01** (upper left-hand quadrant within the previous upper left-hand quadrant). The next and

final quadrant is **10** (lower right-hand quadrant within the previous upper left-hand quadrant). The combined *z*-order prefix then becomes 010110.

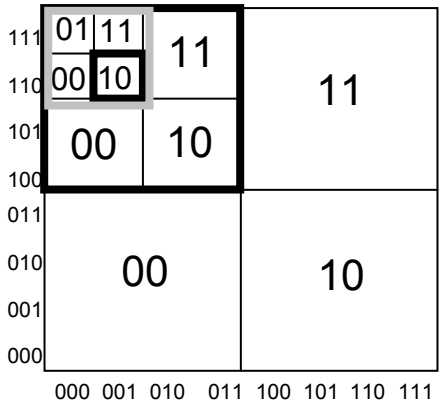


Figure 9. *Z*-order Hierarchy

The same result is obtained if the bits of each dimension (x, y) were interleaved. Following the pattern from Figure 8, the coordinate (001, 110) yields a *z*-order of 010110. It should be noted that 010110 is just the *prefix* of the *z*-order value of any node residing in that particular area. This research uses the notation “XX” to refer to “don’t cares”. For instance, 010110XXXXXX is the same as 010110, and both mean the *z*-order (prefix) denoting a particular area of interest.

2.3.2 Querying Using Standard Skip Graphs

Section 2.2.2 discusses the constructs of a standard skip graph and illustrates how a key search may be performed. Querying using standard skip graphs is a simple extension to a key search. Range queries are particularly efficient to execute using a skip graph, and this research focuses on those types of queries. Since each level in the skip graph contains an ordered list of nodes, the querying process is quite simple.

Each list is sorted based on node key values. The key must correspond with the type of data being queried. As mentioned earlier, this key may be single-dimensional or a

linearization of multi-dimensional information. In a range query, the desired result is the set of all nodes in the skip graph that contain a key value within the specified range. To accomplish this, a search is performed for any node with a key value within that range. Once found, the query can drop down to the lowest level L_0 and traverse the base list in both directions until a key value that falls outside of the desired range is found. At this point, the query knows it can terminate since the base list contains all nodes in the skip graph. Figure 10 illustrates this process; the range query is 2 <= key <= 4.

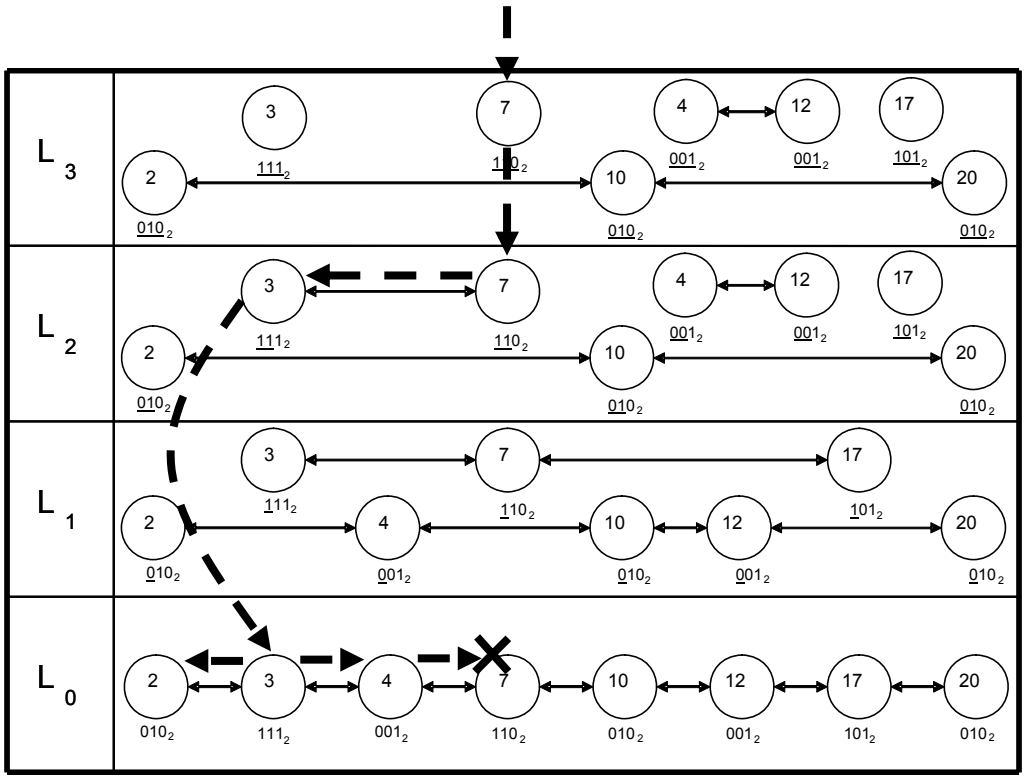


Figure 10. Range Query (2 <= key <= 4) In A Standard Skip Graph

In Figure 10, a range query is injected into the network at node 7. The intent of this query is to contact all nodes with key values within the range of 2 and 4, inclusive. Since the query range falls below the value of the key at node 7, and node 7 has no

references to any other nodes at L_3, the query must drop down to level L_2. At this point, node 7 now has a reference to node 3, and so the query transfers over to it. Node 3's key value (3) falls within the specified queried range, so the query can drop down to the base level L_0 at node 3 and advance in both directions, ensuring all nodes within the query range ($2 <= key <= 4$) are contacted. The query stops (as indicated by the X in L_0) when it finds a node in L_0 that has a key value (7) outside of the query range. In this case, that node happens to be the same node the query started with, node 7.

2.3.3 Querying Using Multi-Dimensional Skip Graphs

A multi-dimensional skip graph has the exact same construct as a standard skip graph. Nodes are collected into ordered linked lists at each level of the skip graph. However, the process of putting nodes into membership vectors is quite different which further implies a different querying mechanism is needed. The query is no longer searching for matching node keys – the query is searching for matching membership vectors.

Recall that in a standard skip graph, keys are deterministically produced from some scalar or multi-dimensional information. Within the context of this research, those key values happen to be the *z*-order value of the node's geographical coordinates. Membership vectors are randomly chosen and each node is grouped into a particular linked list at each level based on some prefix of the membership vector.

Also recall that a multi-dimensional skip graph is different in that node keys are chosen randomly, and it is the membership vector that contains the multi-dimensional information. The membership vector is deterministically created from the *z*-order value of

the node's geographical coordinates, much like the keys in a standard skip graph. Figure 11 illustrates a set of nodes in various geographical positions. The grid layout is similar to that of Figure 9. The circles represent nodes, and the numbers within the circles are the nodes identity.

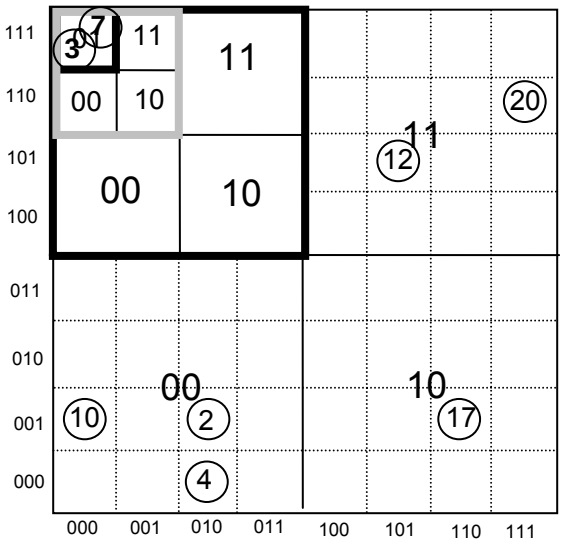


Figure 11. Geographical Layout of Nodes

Figure 12 is the complete multi-dimensional skip graph based on the node layout in Figure 11. The bit sequences below each node in the skip graph are the membership vectors. They are computed from the *z*-order linearization of each node's geographical coordinates, based on the coordinate system of Figure 11. The prefix of the membership vector that determines which list a node belongs is underlined. The prefix matching increases by two bits each successive level because the number of dimensions being linearized is two: latitude and longitude.

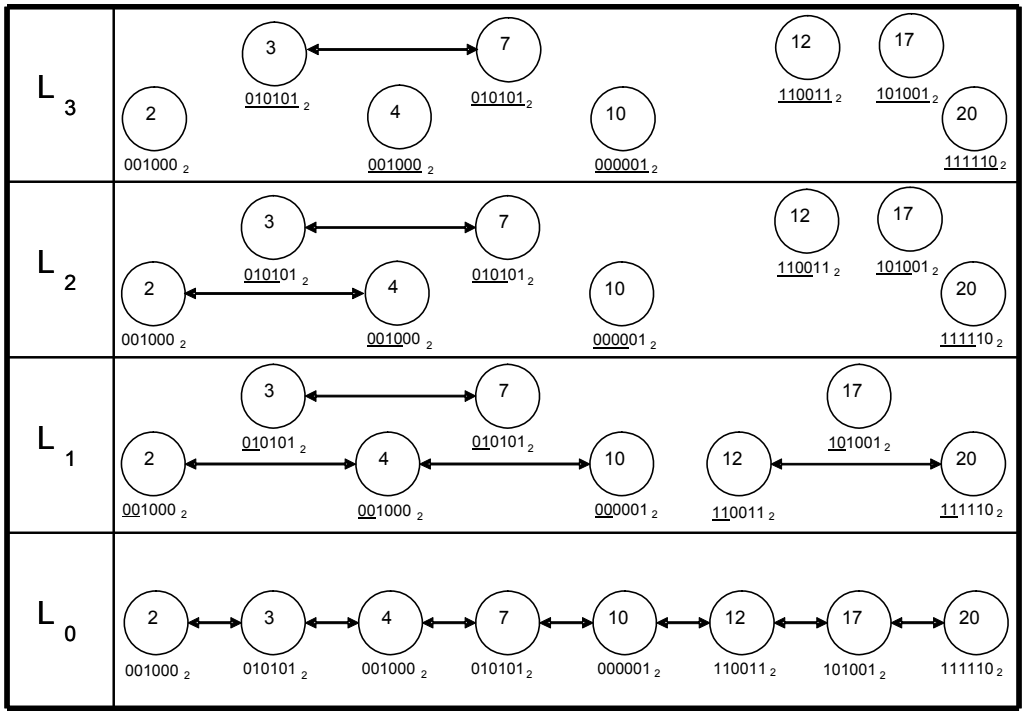


Figure 12. Multi-Dimensional Skip Graph

A query is injected at a particular node as in a standard skip graph. However, instead of the query starting at the topmost level, the query begins at the base level L_0 and traverses in both directions. The query traverses the base list until it finds a node that has an L_1 membership vector prefix that matches the query's prefix for L_1. When the query finds that node, it advances to L_1 and begins traversing the linked list associated with the matching membership vector. It follows a similar process as it did in the base list, but now the query is looking for a longer prefix to match. The query continues this process of longer prefix matching as it advances up the skip graph levels. The query stops when one of three conditions is met.

The first conditions arises when the query's prefix fully matches the prefix of a membership vector at some particular level. This indicates that every node within that

linked list should be notified of the query. The query progresses in both directions until reaching both ends of the list. For example, Figure 13 illustrates the execution of a range query “0010X”. (This query notation will be used throughout this document. “0010” is the query prefix, and the “X” indicates any value can be inserted.) Essentially, this is a query for all nodes within the bottom-right quadrant of the bottom-left quadrant of the main grid found in Figure 11. The query is initiated at node 12, and progresses in both directions in the base list L_0. The query that progresses to the left is transferred to node 10, where it is determined that the L_1 membership vector prefix (00) of node 10 matches that of the query (0010X) at that level. At this point, the query remains within node 10, but advances to L_1. The query transfers to node 4, where it is determined that the L_2 membership vector prefix (0010) of node 4 matches that of the query (0010X) at that level. The query remains at node 4, and advances to L_2. The query can now traverse the entire list in which node 4 resides, because the query prefix of “0010” exactly matches it.

The second condition that stops a query occurs when the query advances to the highest level of the skip graph. At this point, the query prefix is being compared with the longest possible membership vector prefix. The query must traverse the entire list at the top-most level, and at each node compare the entire query prefix with the appropriate prefix length of the membership vector. Unless every node in this list should be included in the query results, this condition can have a significantly negative impact on query response time because it must be sent to every node in the list.

The third condition that stops a query occurs when the query traverses an entire list without finding any L_{i+1} membership vector prefixes that match the query prefix for

that level. Continuing our example with Figure 13, if the query was "0011X," upon reaching L_1 the query would traverse through the entire list of node 10, node 4, and node 2, and not find any L_2 membership vector prefixes of "0011". Therefore, the query determines that no nodes exist with a membership vector prefix of "0011," which is in fact the case.

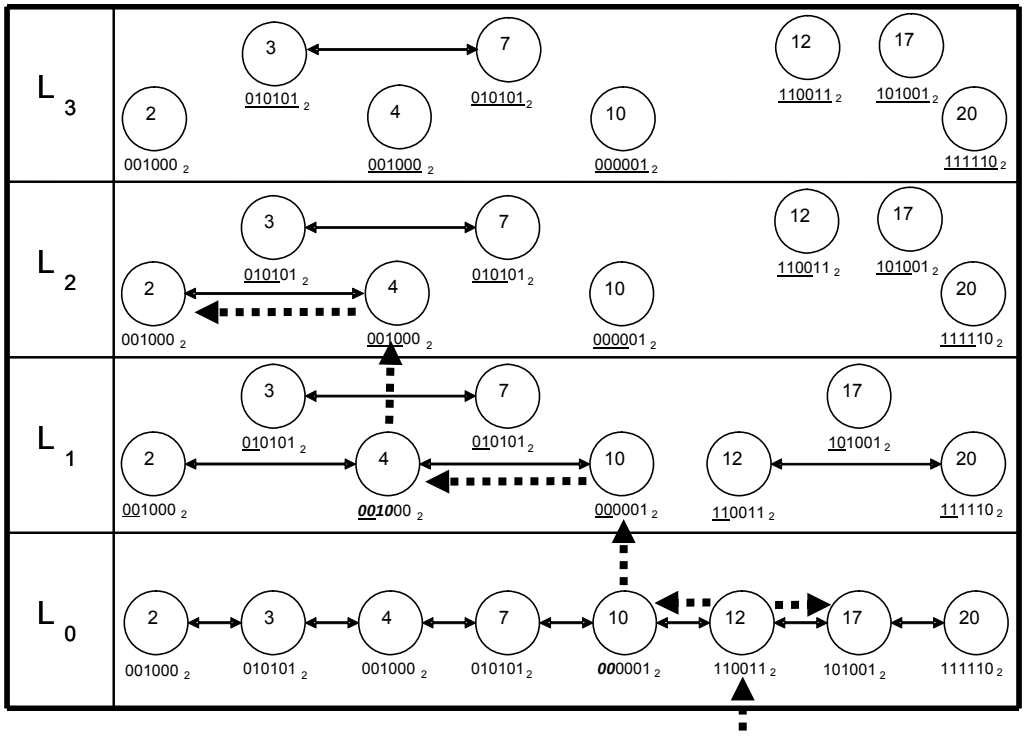


Figure 13. Range Query "0010X" In A Multi-Dimensional Skip Graph

This querying process is analogous to the hierarchical and self-similar properties of the *z*-order curve discussed in Section 2.3.1 and illustrated by Figure 7 and Figure 9. The base list L_0 is analogous to a maximally zoomed-out *z*-curve. A query locating a node with a matching L_1 membership vector prefix is analogous to locating a node that is in the quadrant that needs to be zeroed in. The process is repeated within only that quadrant (i.e., the L_1 list that had the matching membership vector prefix).

2.4 Mobility

If nodes in skip graph are able to move, their geographical locations changes over time. This impacts both types of skip graphs. First and foremost, node mobility causes link failures between nodes that were previously in communication, but because of increased distance, are not able to communicate anymore. These link failures invalidate node references throughout the skip graph. [AsS03] discusses how nodes maintain a valid skip graph structure. Figure 14 shows a node moving from (010_2, 001_2) to (101_2, 011_2).

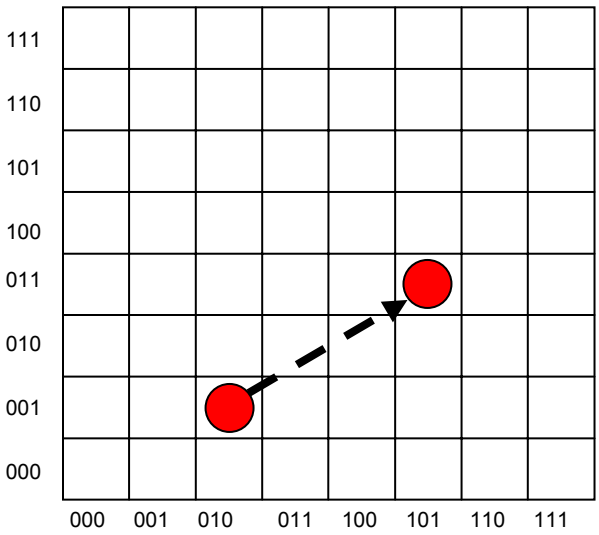


Figure 14. Node Mobility

For a standard skip graph, this corresponds to a key change from 001001_2 to 100111_2. The random membership vector may remain the same, but the position in each linked list at each level changes due to the new key. The opposite happens in a multi-dimensional skip graph. The random node keys remain the same, but the membership vector changes based on the new *z*-ordering of the new coordinates. This essentially moves the node into a new linked list at each level (except L_0).

Update messages are sent as nodes enter and leave geographic areas that the skip graph encompasses. Some nodes may not even realize the reference to an adjacent node is not valid until executing its part of a query. The number of update messages sent is dependent on the number of levels the skip graph must maintain. A skip graph with more levels must iterate a node through each level to determine which linked list the node belongs to at each level. In addition, a node's key in a standard skip graph changes upon a geographic move; the key is created based on the *z*-order construction of the node's geographic coordinates. This implies a search query (using its own key as the search parameter) is executed to determine where in the base list in L_0 the node is to be inserted. Likewise, in the proposed multi-dimensional skip graph, the node's random key remains the same, and so the node preserves its position in the base list in L_0. The node simply iterates through every level higher than L_0. The absence of executing search queries has a noted impact on message counts incurred due to mobility.

2.5 Related Work

Skip graphs are relatively recent compared to the other distributed data indexing techniques. William Pugh developed the skip list in 1990 [Pug90], which led to a more recent extension of skip graphs in 2003 [AsS03]. Other work has either extended skip lists or skip graphs, or used them for a particular application.

2.5.1 Quadtrees

Quadrees, developed in 1980 [EGS04], are regions defined by hierarchical divisions of a plane, such that any data points within the plane are within one of these divisions. Typically, regions are divided into *four* equal-sized squares, giving the

quadtree its name. Each division of the plane is further divided into four regions of equal size, and so on. This hierarchical division process continues until all data points within the plan are within a single division.

The precision of queries discussed in this research pertain directly to the notion of the quadtree. The number of levels of a query is a function to how many hierarchical divisions the geographical area of interest is divided into.

2.5.2 Skip Quadtree

Skip quadtrees [EGS04] combine skip lists and quad trees and results in a dynamic data structure for multi-dimensional data. Skip quadtrees can be applied to range searching algorithms, the basis of this research. This research, however, uses quadtrees with skip graphs, not skip lists. Essentially, that is what standard and multi-dimensional skip graphs in this research are based on: determining node keys or membership vectors within a skip graph based on a location in the hierarchy of the quadtree.

2.6 Summary

This chapter reviews fundamental concepts of retrieving query information from a skip graph. It presents the structures of skip lists and skip graphs and how multi-dimensional data such as geographic positions can be linearized into a single dimension using techniques such as *z*-ordering. Issues with node mobility are introduced and their impacts on skip graph functionality are discussed. Finally, other work related to skip graphs is presented.

III. Methodology

This chapter presents the methodology used to investigate the impact a multi-dimensional skip graph has on range queries and mobility performance in a distributed sensor network. Section 3.1 defines the problem statement, including the goals of the research and the approach used to achieve those goals. Section 3.2 discusses the scope of the problem. Section 3.3 introduces the three parts of the system model from a software engineering perspective: the model, view, and controller. Section 3.4 discusses metrics used to evaluate query and mobility performance. Sections 3.5 and 3.6 describe the parameters of the model, and which of those parameters are used as experimental factors. Section 3.7 addresses the evaluation techniques used, and section 3.8 discusses the design of the simulation.

3.1 Problem Statement

3.1.1 Goals and Hypothesis

The primary goal of this research is to determine the performance of range query execution after modifying the standard skip graph [AsS03]. The type of range query includes returning a set of nodes geographically bound by a certain area. A secondary goal of this research is to determine how node mobility affects the dynamics of both the standard and multi-dimensional skip graphs by examining the cost of sending the necessary update messages to neighboring nodes to sustain skip graph invariants [AsS03].

The number of levels, node count, geographical position precision, and geographical query size determines the performance of query execution. In an ideal skip

graph, a node is added to an existing skip graph distributed data structure by determining its membership vector and subsequent membership vector prefixes for each level, and being added to the appropriate list accordingly [AsS03]. It continues this process level by level until the node becomes a *singleton*. That is, there are no nodes that match its membership vector at a particular level. A node within a DSN may not have adequate resources for a skip graph. Memory and processing resources on portable sensor nodes may limit the number of levels that a skip graph can have. An increase in the number of levels in a skip graph implies an increase in neighbor node references that need to be maintained. An increase in the number of levels also allows more precise geographical queries to be executed efficiently. Precise geographical queries may still be executed with a limited number of levels, but as the node count increases, the possibility of wasted node communications becomes more probable.

The number of levels and node count determines mobility performance. When node changes geographical position, it affects the two skip graph types in different ways. In a standard skip graph, the key values are associated with the node's geographical position. If a node's position changes, then so does the node's key. This means re-ordering the node in each linked list the node is a member of, including the base list at L_0. Chapter 4 discusses the impacts of that key change. The node's membership vector remains the same. In a multi-dimensional skip graph, it is the node's membership vector that is modified when a node's geographical position changes. The node's key remains the same, but the node now moves to possibly different linked lists at each level, save L_0.

Since nodes don't need to re-order in L_0 in a multi-dimensional skip, this has positive impacts on node mobility. This is also discussed in Chapter 4.

3.1.2 Approach

Computer simulations are used to determine the performance of query execution and mobility scenarios. Modifications of skip graph levels, node counts, range query size (selecting all nodes within differing bounded geographical areas), and node geographical coordinates are incorporated for both standard and multi-dimensional skip graph simulations. A professional simulation product such as OPNET Modeler [Opn07] is not integrated into the simulations, and so model validation tests are required. Simulation measurements are analyzed to compare message counts and approximate hop counts between the different configurations.

3.2 System Scope

A DSN is a system of independently operating sensor nodes that communicate with one another using a routing protocol. Each node is a participant in a distributed data structure, a skip graph. Several assumptions have been made to scope the problem to a manageable size.

The primary focus of this research is executing geographical queries within a DSN. Therefore, each node has some form of on-board geographical positioning hardware such as Global Positioning System (GPS), or has the ability to determine its geographical position from neighboring nodes. This information is considered inherent to every node.

A real-world distributed network must compensate for message failures. Failed nodes, noisy transmission environments, and packet collisions are just a few of the problems in a wireless network. The simulations assume every skip graph message sent is received by the intended destination.

The constraints, or invariants, in an ideal skip graph include the following [AsS03]. Let x be any node in the skip graph, $x\mathrm{R}_i$ be the node reference to the **R**ight of node x at level i, $x\mathrm{L}_i$ be the node reference to the **L**eft of node x at level i, and $m(x)_i$ be the prefix of the membership vector of node x at level i.

1. If $x\mathrm{R}_i \neq \perp$, $x\mathrm{R}_i.\mathrm{key} > x.\mathrm{key}$ | If node x has a reference to a node to its right, then the node to the right's key value is greater than the key value at node x.
2. If $x\mathrm{L}_i \neq \perp$, $x\mathrm{L}_i.\mathrm{key} < x.\mathrm{key}$ | If node x has a reference to a node to its left, then the node to the left's key value is les than the key value at node x.
3. If $x\mathrm{R}_i \neq \perp$, $x\mathrm{R}_i\mathrm{L}_i = x$ | If node x has a node reference to its right, then the node to its right has a reference to node x on its left.
4. If $x\mathrm{L}_i \neq \perp$, $x\mathrm{L}_i\mathrm{R}_i = x$ | If node x has a node reference to its left, then the node to its left has a reference to node x on its right.

Essentially, the simulations execute as if the skip graph remains in a stable state where these invariants always hold. Invariants will not hold as the mobility simulations are performed, until the necessary update messages are sent to stabilize the skip graph. The simulations use a local data structure; therefore when a node moves, the references in

memory are updated accordingly, and the number of messages needed to stabilize the skip graph are recorded.

3.3 System Model

To perform query and mobility simulations on the standard and multi-dimensional skip graphs, the skip graphs are modeled in software. The Java Programming Language [Jav07] is used to implement both skip graph types and the simulations. Eclipse 3.1.2 [Ecl07] provides the Integrated Development Environment (IDE). The Java/Eclipse combo was chosen because of its widespread popularity and the author's familiarity. Representations of the system model use the Unified Modeling Language (UML) [UML05]. UML is an industry standard language to model and specify interaction between components of a software system. In each UML diagram, superfluous details are omitted to maximize clarity. The software design follows the standard Model-View-Controller (MVC) architecture. The following sections describe each part of the MVC architecture.

3.3.1 Model

Figure 15 is a high level Class Diagram for both the standard and multi-dimensional skip graph. Java is an object-oriented language well-suited for model abstraction and both versions of skip graphs are implemented as local data structures in Java for several reasons. The focal points of this research are query execution and mobility performance. The query simulations execute in a "well-behaved" skip graph. That is, nodes do not fail, and links between nodes are healthy. Skip graphs are inherently

a distributed data structure, but to help scope the problem to focus on query performance, certain issues arising from this distribution have been ignored.

As Figure 15 illustrates, an abstract SkipGraph class is sub-classed into the two different versions of skip graphs: the standard skip graph (StdSkipGraph) and the multi-dimensional skip graph (MultiSkipGraph). Note that the rangeQuery(Node, int, int) and updateNode(Node) methods are overridden in each subclass; each skip graph type implements its own version of querying and node update (for mobility) algorithms.

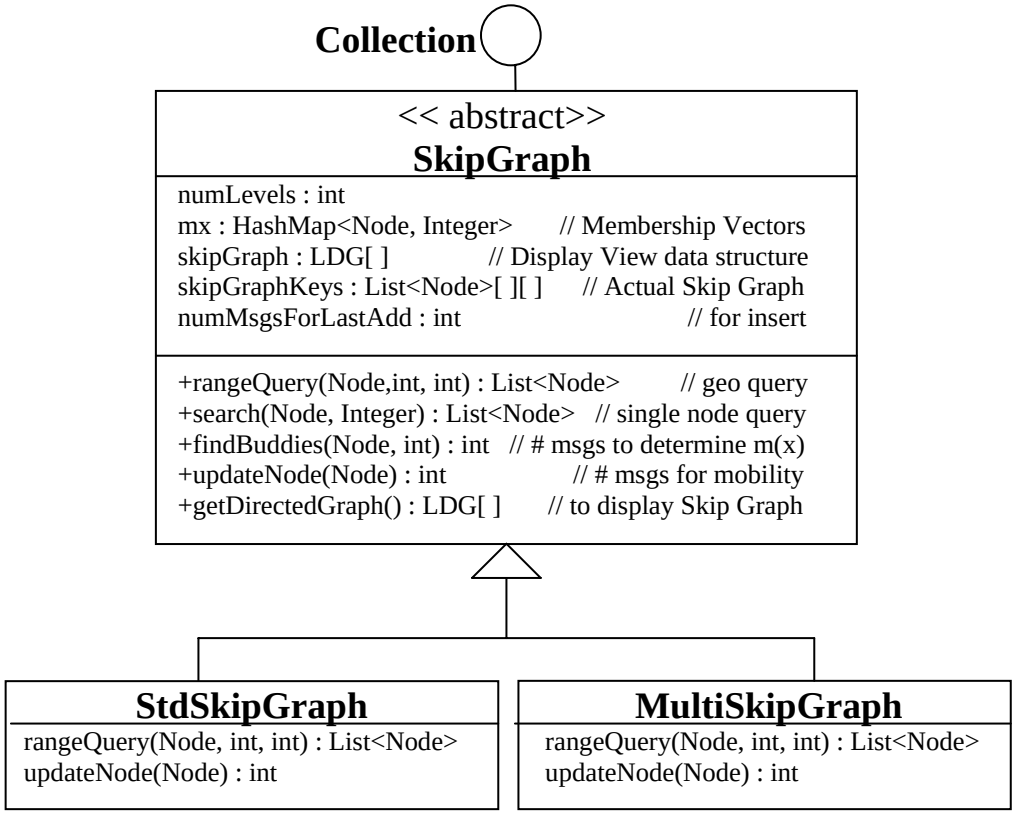


Figure 15. SkipGraph Class

Skip graphs incorporate methods associated with collections such as add( ) and delete( ), and also exhibit properties of order. Therefore, skip graphs implement a collection interface.

Figure 16 shows the composition of many nodes in a skip graph. Each node has several attributes, including geographical position (x,y,z coordinates), node id, and membership vector. Nodes are aware of what type of skip graph they belong using the Strategy design pattern [Fre04]. A node is constructed with one of two strategy classes: RandomMX or RandomKey. The RandomMX class creates a node using a standard skip graph, whereas RandomKey is used to create a node using a multi-dimensional skip graph.

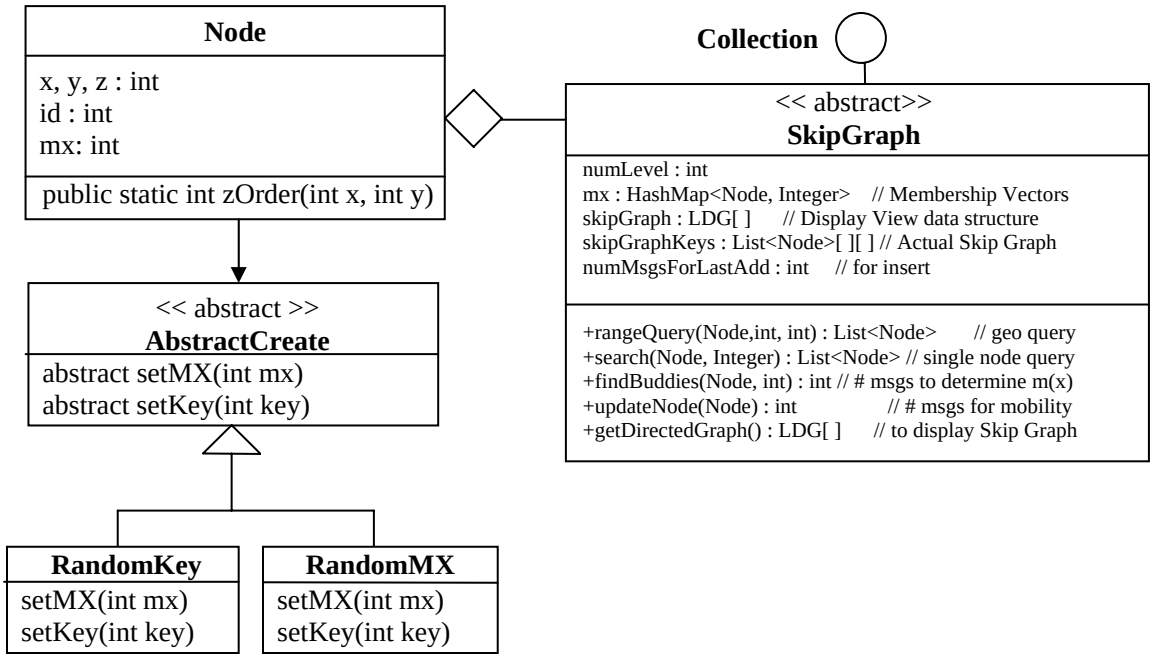


Figure 16. Node Class

As Figure 16 illustrates, a node references an abstract creation object, which is sub-classed into either RandomKey or RandomMX object. Each of the objects builds the node's key and membership vector differently, and using the strategy design pattern is an efficient way to accomplish this.

3.3.2 View

Figure 17 shows a snapshot of the simulation interface. The views were designed to serve two general purposes. First, the main window frame functions as a launch pad for different simulations that can be run which are selected via menu options. Second, the main window serves as a parent window for the internal frames generated to view the properties of a skip graph. Two views can be generated: the skip graph view and/or a geographical layout view. Each of these views is selected via menu options. Figure 17 shows the main simulation window with these two internal frames.

The top-right internal frame shows a graphical representation of the skip graph currently being modeled. Note the horizontal line separation indicating level changes. The arrowed line segments connect various nodes within the graph. These are analogous the horizontal link references between nodes as seen in earlier figures of skip lists and skip graphs.

The bottom-left internal frame shows the geographical layout of all the nodes in the skip graph. It serves as a visual check indicating the correct nodes were selected for a given multi-dimensional range query. The nodes are positioned within the internal frame based on their geographical coordinates relative to the size of the frame.

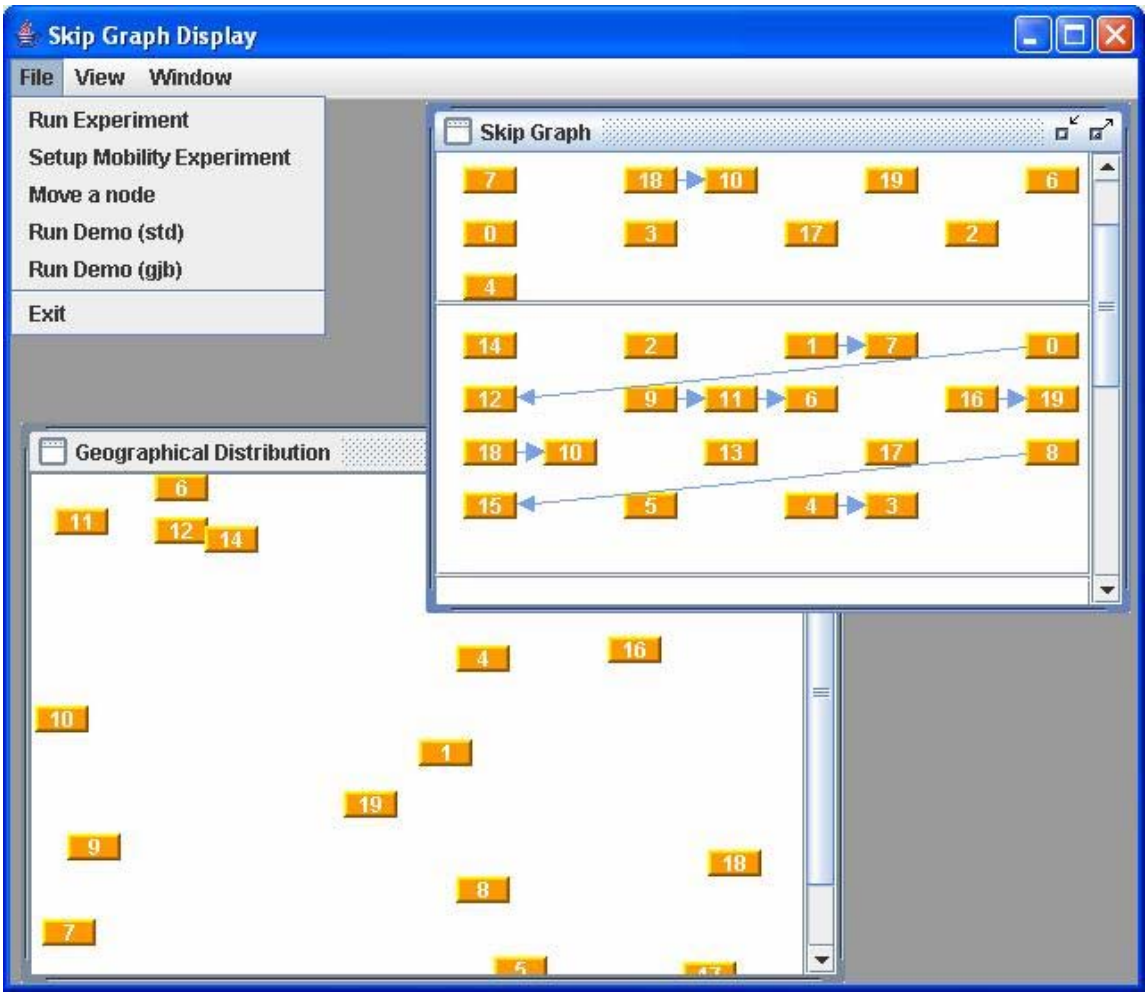


Figure 17. View Snapshot

3.3.3 Controller

The controller class brings the model and view classes together so simulations can be run and data gathered. Figure 18 shows the overall MVC architecture for the design implementation. When the user selects a menu option, the controller class creates all of the necessary objects to perform the requested operation.

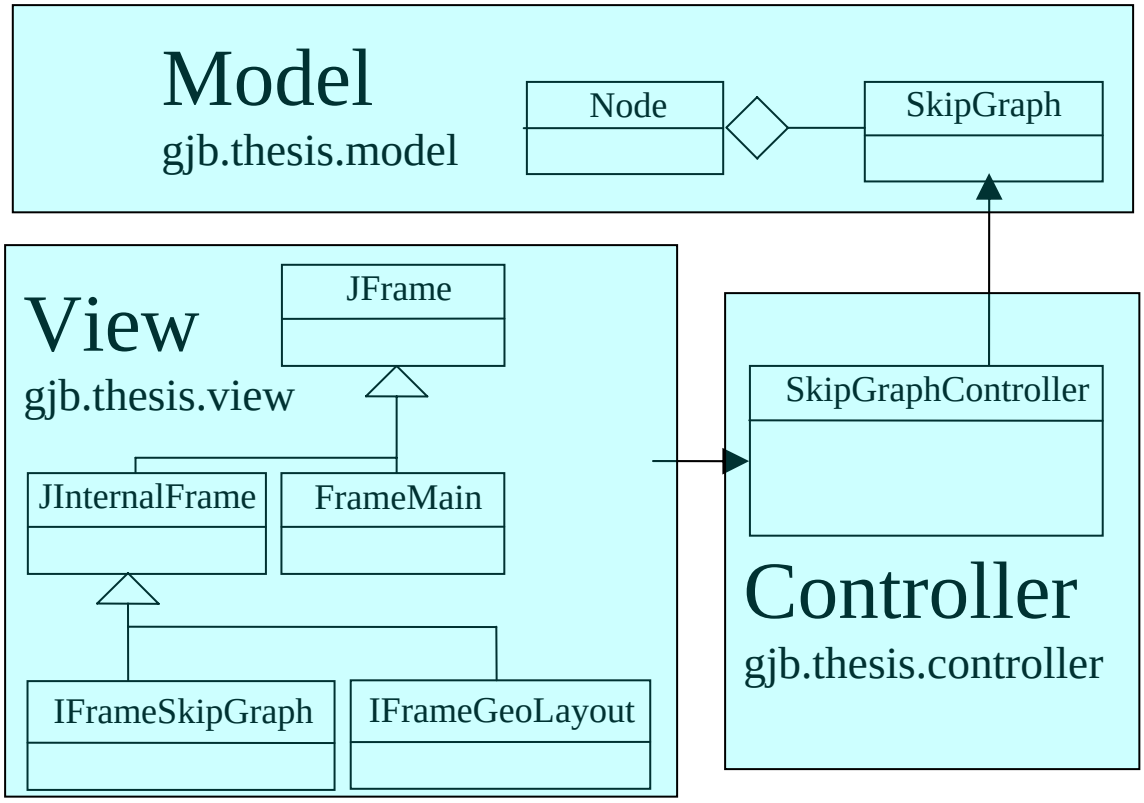


Figure 18. Summary of M-V-C UML

3.4 Performance Metrics

Two aspects of skip graphs were examined in this research: geographical querying performance and node mobility performance.

Query performance is compared in several ways. DSNs are typically concerned with power management performance, and so the less radio energy expended, the better. It may be more energy efficient to propagate the query through a series of three nodes instead of directly transmitting from the first node to the third. At the same time, however, it is also desirable to minimize query response time, so tradeoffs must be made.

In general, the fewer the nodes contacted in a particular query, the lower the response time will be.

The following metrics measure geographical query performance:

- Nodes contacted with skip graph messages: Measures the number of nodes that either sent a message to another node, or had a message sent to it.
- Number of routing layer messages: The number of underlying routing layer packets sent to complete the query. Skip graphs are an application layer-based routing system and so the underlying network routing protocol (e.g., AODV) is abstracted away.

Node mobility performance is primarily based on how many skip graph messages it takes to return the skip graph to a working state after nodes have been moved. As discussed in Chapter 2, if a node changes location, queries may need to be performed (if the node must reorder itself within the base list L_0 as in a standard skip graph) and update messages need to be sent to re-associate the node to possibly new linked lists based on the node's new membership vector. The following metrics measure node mobility performance:

- Messages Sent: When a node moves to a new geographical location as described above, it needs to be reinserted in the correct position in the correct membership vectors at all levels. This metric counts how many messages were sent to accomplish this task.

3.5 Parameters

The parameters of a skip graph and associated querying mechanisms are properties that, if changed, are likely to affect the performance of the query or the ability to maintain a skip graph if nodes are mobile. Each of the parameters listed below apply to both standard skip graphs and multi-dimensional skip graphs.

3.5.1 Skip Graph Parameters

- Levels: A node is added to a skip graph level by level, the number of levels should be increased until that particular node is a singleton [AsS03]. That is, until the node exists by itself in its own linked list at the top level. Ideally, this should be the case. However, in reality, sensor nodes have limited memory and processing power. Therefore, it's prudent to examine how the modification of the number of levels in the skip graph affects query performance.
- Node Count: Node count reflects the total node population in a given sensor network. Node count impacts query response time because as the population changes, on average fewer or more nodes will need to be notified about the query.
- Query size: This research notifies all nodes within a certain geographical range that they must report some arbitrary sensor value. Figure 19 shows the familiar geographical grid highlighting two different queries. A smaller query size implies a higher query precision.

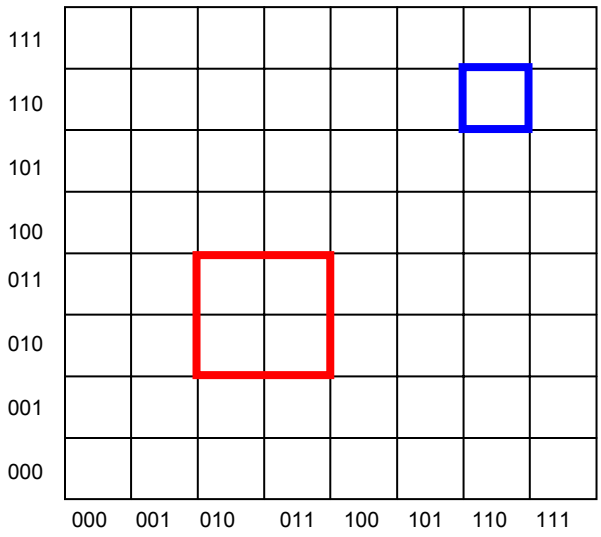


Figure 19. Two Range Queries

- Routing Protocol: The routing protocol has an impact on query and mobility performance.

3.5.2 Mobility Parameters

- Levels: It is prudent to examine how a differing number of skip graph levels affects mobility performance. As more levels are added, the number of update messages that need to be sent to maintain the skip graph invariants increases.
- Node count: Node count affects mobility performance because as the network size increases, so do the number of messages to send to maintain the skip graph invariants.
- Node speed: The speed nodes are moving impacts the ability of the skip graph to incorporate the changes. If a node moves so fast that it changes geographical bounds before the skip graph has the opportunity to update itself, this will significantly affect query performance.

- Number of moving nodes: The more variation in node positioning, the harder it is for a skip graph to fix itself quickly. The more moving nodes there are, the longer it will take for a skip graph to become stable.

3.6 Factors

Factors are a subset of the skip graph and mobility parameters. These parameters are chosen to limit the scope of analysis so it can be determined how they affect the performance of queries.

Any parameter that is not chosen as a factor is explained. Since skip graphs are application layer-based distributed data structures, the routing protocol is abstracted away. It is assumed that the routing protocol suites the needs of the sensor network application. The remaining parameters are chosen as factors and are listed below.

- Number of Levels: Based on preliminary simulations, level values of 4, 5, 6, 8, and 10 are chosen.
- Number of Nodes: Node counts of 100, 500, 1000, 2000, 5000, and 10000 are chosen.
- Query size: The geographical range query contacts all nodes within a specified coordinate range. Figure 20 shows two examples of range queries. The selected queries for this research are 10XXXXXXXXXX, 1001XXXXXXXX, 100110XXXXXX, 10011000XXXX, 1001100011XX, and 100110001101. The prefixes (e.g., 1001 in the second query) were selected arbitrarily. As Section 3.7 will discuss, each query is submitted 100 times. A query of 1001XXXXXXXX will, on average, return the same number of nodes as a query of

1011XXXXXXXXX. The simulations are more concerned with how many extra nodes are contacted to contact all of the required nodes, and modifying the prefix length of the query is a factor.

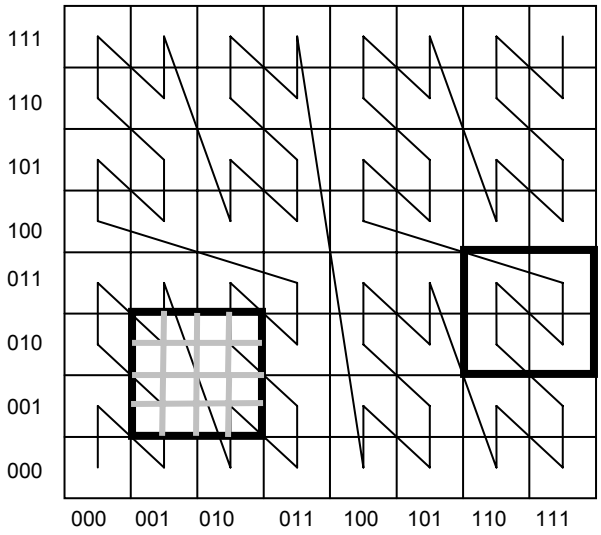


Figure 20. Good Query (Right) Versus Bad Query (Left)

In Figure 20, the *z*-curve slices in and out of the bounded query on the left, whereas the bounded query on the right contains a complete sub-section of the curve. The nature of *z*-ordering makes it difficult to execute range queries like the one on the left. There are two immediate ways to alleviate this problem. The first way, and the more complicated of the two, is to break up a given query into smaller sub-queries. To execute the query on the left, smaller queries are derived by slicing up the query into sixteen parts. The second method, the one this research chose to use, is to formulate queries that encompass entire sub-sections of the *z*-curve, as in the query on the right. Table 1 shows a summary of the factors.

Table 1. Summary of Query Factors

Factor	Values		
Levels	4	5	6
Nodes	100	500	1000
Query Size	10XXXXXXXXXX	1001XXXXXXXX	100110XXXXXX
Levels	8	10	
Nodes	2000	5000	10000
Query Size	10011000XXXX	1001100011XX	100110001101

Similar to query performance parameters, any mobility parameters that are not chosen as factors are explained. The mobility simulation considers a single node as it instantaneously transfers from one position to another. The simulation then determines how many messages are needed to fix each type of skip graph. The remaining node mobility parameters are factors and are listed below. Table 2 gives a summary of the node mobility factors

- Levels: Based on preliminary simulations, levels of 4, 6, 8, and 10 are chosen.
- Node count: Node counts of 100, 500, 1000, 2000, 5000, and 10000 are chosen.

Table 2. Summary of Node Mobility Factors

Factor	Values		
Levels	4	6	8
Nodes	100	500	1000
Levels	10		
Nodes	2000	5000	10000

3.7 Experimental Design

Each combination of the factors of Table 1 and Table 2 constitute parameters to the simulation to determine the performance of geographical range queries and node mobility, respectively.

As Table 1 shows, 5 levels * 6 node counts * 6 query sizes total 180 separate simulation runs. Each of those simulation runs is replicated 100 times to capture random fluctuations due to different node distributions. Each of the replications creates a standard and multi-dimensional skip graph with the same (random) geographical distribution of nodes. Java's Random class provides the random distributions; therefore different seed values are used for each of the iterations.

For instance, one of the 180 simulations calls for a skip graph with 4 levels, 500 nodes, to execute a 10XXXXXXXXXX query. That particular simulation iterates 100 times, creates the two different 4 level skip graphs, and populates each with 500 nodes of the same (random) geographical distribution. The query is then submitted to both skip graphs, and how many nodes are involved in the query execution is recorded.

As Table 2 shows, 4 levels * 6 node counts total 24 separate simulation runs. Each of those simulation runs are replicated 100 times to capture random fluctuations due to different node distributions and differing node selections. Similar to the query simulation, each of the iterations create a standard and multi-dimensional skip graph with the same (random) geographical distribution of nodes. Again, Java's Random class provides the random distributions; therefore different seed values are used for each of the iterations.

3.8 Evaluation Technique

The skip graph models are implemented in the Java programming language, and so a controller class is created to facilitate the simulations. All of the factors found in Table 1 and Table 2 are configurable in the software. Simulation was chosen as the evaluation technique for several reasons. Time and cost constraints do not make it feasible to set up a real-world sensor network consisting of thousands of nodes running skip graphs. That eliminates the option for obtaining empirical results. While there are analytic models for basic skip graph operations like searching for a single node, executing a geographical range query is beyond the scope of this research.

Since a professional-grade modeling tool is not used to run the simulations, it must be shown that the Java model is correct. This is done by executing geographically bounded queries on many random skip graph networks and comparing those results with an expected value. This model validation is analyzed in Chapter 4.

Query performance for each skip graph type is compared to each other by graphing average message count as total node counts increases. Recall that each query simulation is repeated 100 times, and so each message count data point represents an average of 100 trials. Many conclusions can be drawn from an analysis of the graphs alone. However, to state that one skip graph performs a query better than another, a more rigorous analysis is beneficial.

Confidence Intervals (C.I.s) provide insight to an expected mean value from sample data. The C.I.s presented in this research use the *t*-distribution because the actual

variance of the population is not known. The variance, or standard error, is estimated from the sample data using

$$s_m \frac{s}{\sqrt{N}} \tag{2}$$

where s_m is standard error, s is sample standard deviation, and N is the number of samples. The upper and lower limits of the mean can then be computed using the *t*-distribution formula or

$$\text{Upper/Lower Limits} = \text{Sample Mean} \pm t_{\text{C.L.}} * s_m \tag{3}$$

where $t_{\text{C.L.}}$ is the Confidence Level (1.98 for 95% confidence).

In summary, the upper and lower limits computed from (3) are used to determine with 95% confidence that the actual mean falls within those bounds, thus providing a basis for comparison in Chapter 4.

3.9 Summary

This chapter presents the methodology used in conducting this research. It describes in detail the Java implementation of the skip graph models, including the usage of the Model-View-Controller design pattern. This chapter also discusses the various parameters that affect query performance in a skip graph, as well as determining which of those parameters are evaluated in the simulations. Finally, the experimental design is described and how the simulation results are evaluated is presented.

IV. Analysis and Results

This chapter presents the results of the simulations and analyzes those results. Model validation results are presented in Section 4.1. Section 4.2 presents query performance results, and Section 4.3 presents node mobility results.

4.1 Model Validation

Part of this research includes implementing skip graphs in software using the Java programming language. Full program code and comments can be found in Appendix A. Models created in software must be validated to be confident of the correctness of simulation results. This section provides that validation.

Chapter 3 describes the implementation chosen for the skip graphs. This research examines query and node mobility performance, disregarding node and link failures. A suitable model validation technique involves executing a bounded range query in a skip graph populated with nodes uniformly distributed. The node count returned in the query is compared with the expected value of the node count based on query area and average node density.

Figure 21 shows 12 nodes distributed uniformly across the grid. If the highlighted query is executed, the resultant node count will be 3. The expected value of the number of returned nodes depends on node density and query range or

$$E[NodeCount] = TotalNodeCount \times \frac{QueryArea}{TotalArea}. \quad (4)$$

In this case, 12 nodes occupy the unit area. Querying the top-left quadrant yields a Query Area/Total Area ratio of 0.25. Therefore, the expected node count returned in the

query is 12*0.25 = 3 nodes. The actual node counts returned may not always be the same as the expected value of the node counts due to the random nature of the node distribution. Each of the 180 query simulations mentioned in Chapter 3 tracks how many nodes are returned in the query, as well as how many total nodes participated in the query. These node counts combined with the query size information, is enough to compute the model validation graphs as described.

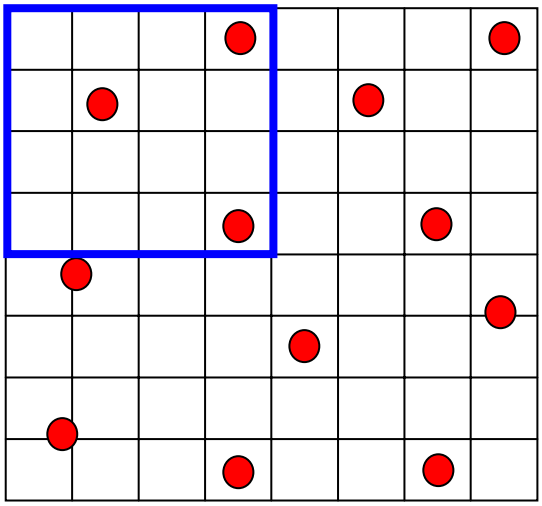
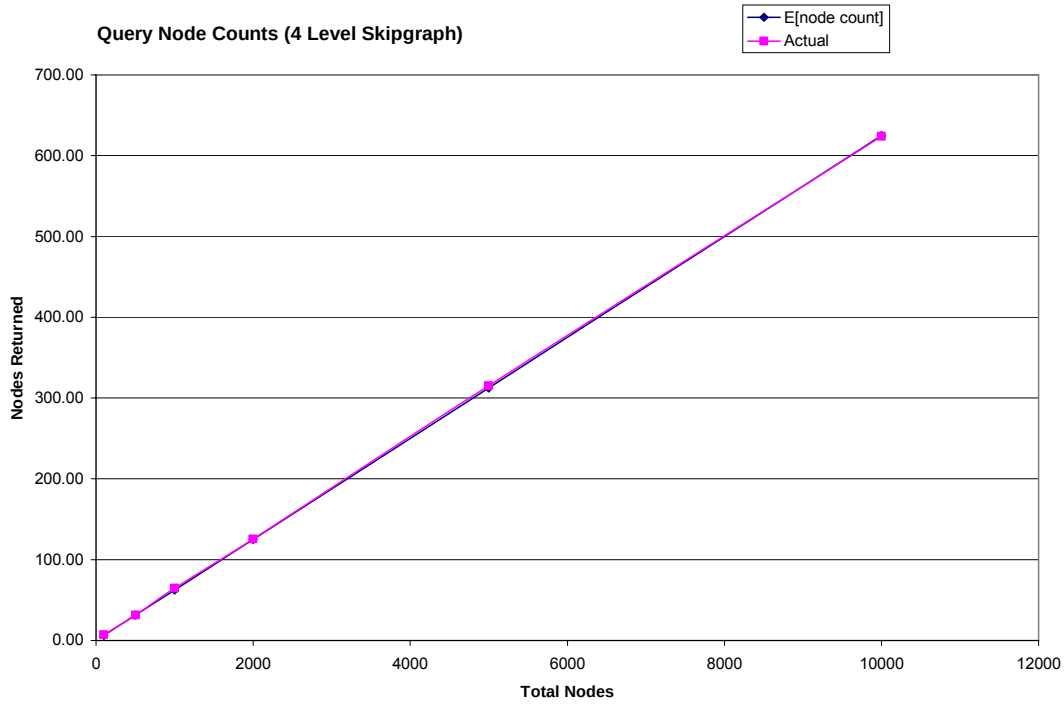
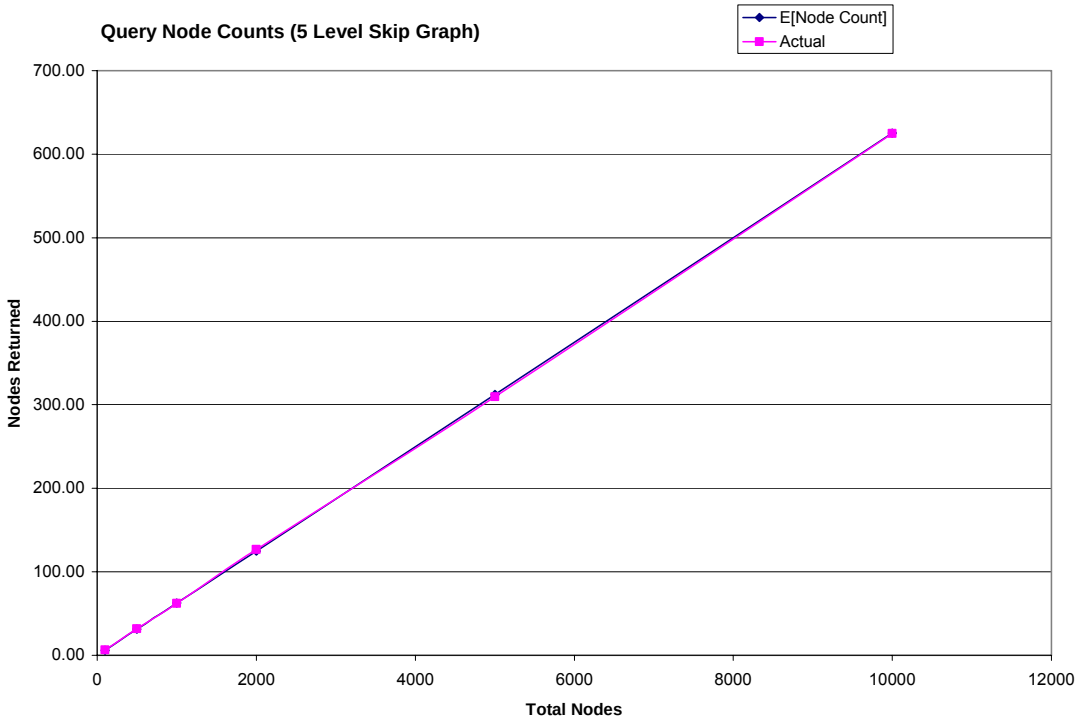


Figure 21. Top-Left Quadrant Query

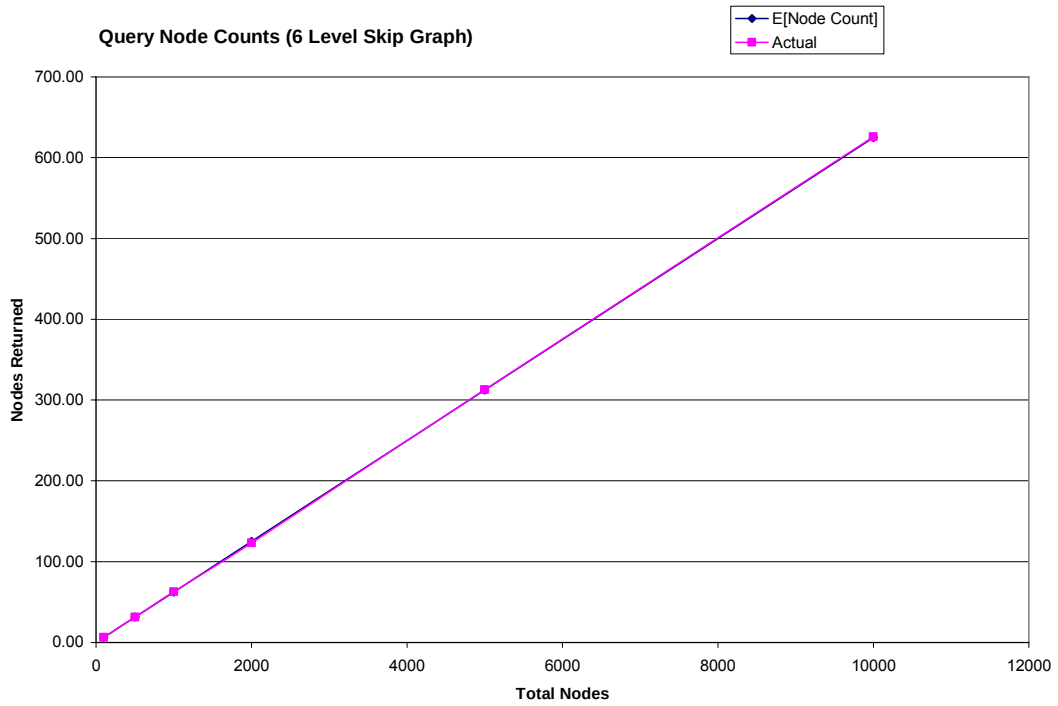
Figure 22a - Figure 22e shows validation results from 5 different skip graphs, each with a different number of levels, and each executing a query of "1001XXXXXXXX".



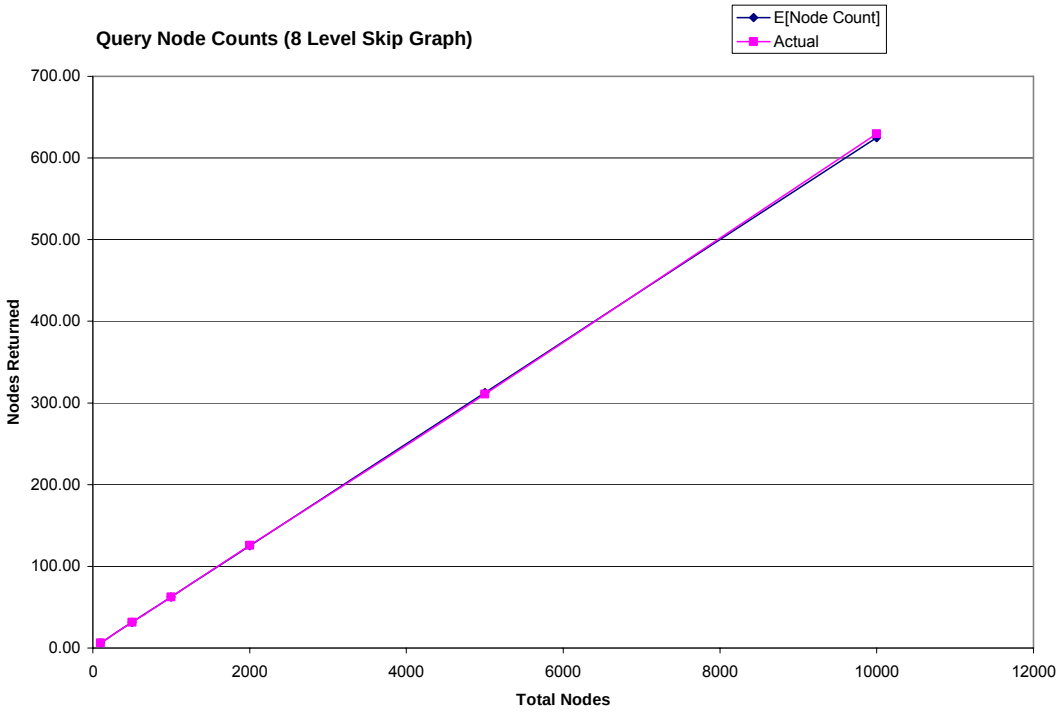
a) 4 Level Skip Graph



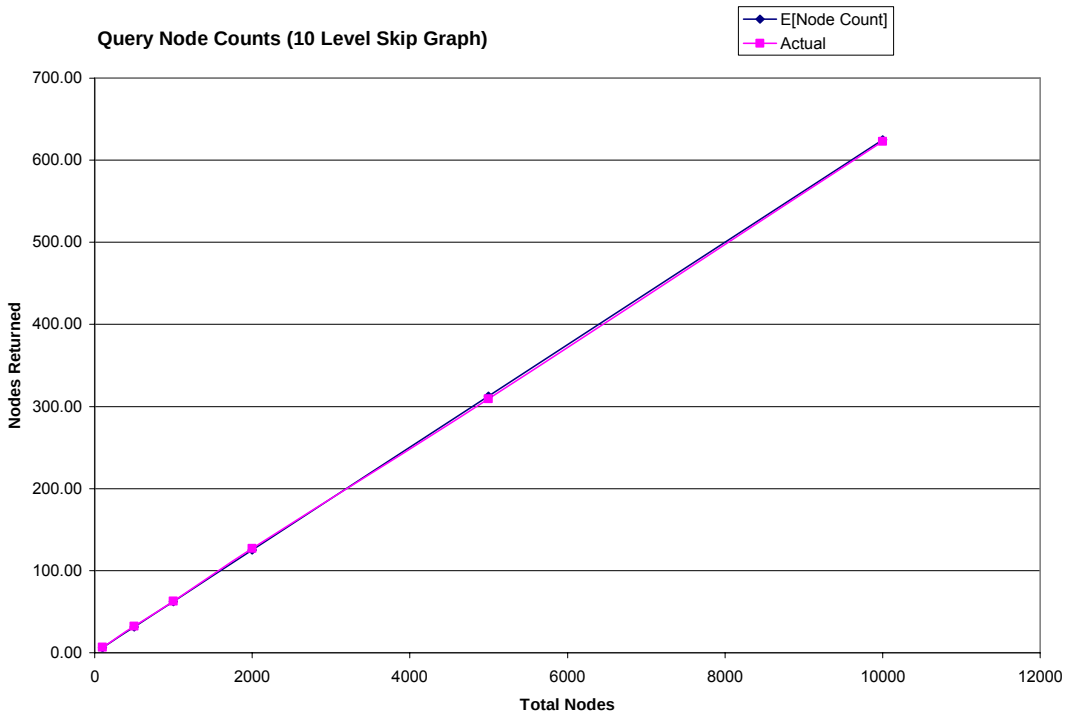
b) 5 Level Skip Graph



c) 6 Level Skip Graph



d) 8 Level Skip Graph



e) 10 Level Skip Graph

Figure 22. Query Node Count, Actual Versus Expected. Circular Points Represent Expected Node Counts. Square Points Represent Measured Node Counts

Each sub-figure from Figure 22 contains two graphs, one with circular data points (expected node counts), and one with square data points (measured node counts). Each sub-figure shows the number of returned nodes from the execution of the query 1001XXXXXXXX, both actual and expected, at different level settings of the skip graph. Recall that each of the query simulations iterates 100 times, and each of the iterations executes the same query on a different skip graph with random node distributions.

In each of the sub-figures, the actual results shadow the expected values almost exactly. Thus the query is executing properly, recording each node that should be

responding to the query. Table 3 shows a summary of Figure 22a, the results from a skip graph with 4 levels. The table compares the actual node counts returned with the expected value of the node count. In each of the simulations varying total node counts in the network, the actual returned node count from the query falls within the 95% confidence interval. The confidence intervals were computed using (3).

Table 3. Model Validation Results, 4 Level Skip Graph

Nodes	E[Node Count]	Actual Node Count	Confidence Interval
100	6.25	6.94	(6.40, 7.48)
500	31.25	31.34	(30.23, 32.45)
1000	62.50	64.48	(62.93, 66.03)
2000	125.00	125.38	(123.19, 127.57)
5000	312.50	315.35	(311.41, 319.29)
10000	625.00	624.15	(619.32, 628.98)

Similar results are obtained when evaluating the other skip graph levels in Figure 22. That data is omitted here because they all point to the same conclusion: the skip graph implementation is correct.

4.2 Analysis of Query Execution

Six query executions, each with varying geographical ranges, are simulated. The largest, 10XXXXXXXXXX, returns all nodes within the lower-right quadrant of the entire unit area. The remaining queries simulations are 1001XXXXXXXX, 100110XXXXXX, 10011000XXXX, 1001100011XX, and 100110001101, each query being more precise than the previous and thus, on average, return fewer nodes. Each simulation records how many nodes were involved in propagating the query to the nodes that need to be notified of the query. In an ideal query execution, only the nodes that fall

within the query bounds propagate the query. However, this is rarely the case, since the query may be injected anywhere in the network and must traverse the network towards the desired area of interest. It is desirable to minimize that extraneous communication. The fewer the nodes involved in query execution implies better query performance.

The following figures show the results of executing the six queries mentioned above. In each figure, ten sets of data are graphed. Five of the sets show the results of executing the query on the standard skip graph (Std) with levels of 4, 5, 6, 8, and 10. The other five sets show the results of executing the query on a multi-dimensional skip graph (Multi) with levels of 4, 5, 6, 8, and 10. The x-axis is the total number of nodes in the skip graphs, and the y-axis is how many nodes were involved in executing the query. Keep in mind that the main metric that to minimize are nodes contacted. Following each of the figures is a table with the 95% confidence interval for each of the data points in the figure that precedes it. The tables list all combinations of node count and skip graph levels, and for each combination have the mean and confidence interval ranges for the standard skip graph and multi-dimensional skip graph.

The last column in each of these tables is the difference in the lower bound of the 95% C.I. of the mean of the standard skip graph node counts and the upper bound of the 95% C.I. of the mean of the multi-dimensional skip graph node counts. A positive value indicates that there is no overlap in the C.I. ranges of the standard and multi-dimensional skip graph which means the mean node count of the multi-dimensional skip graph is lower than that of the standard skip graph for that particular query execution with 95% confidence. These “Difference” values are highlighted in gray. Negative values in the last

column of the tables indicates one of two things: there is an overlap in the C.I. range of the two skip graph types, or, the lower bound of the 95% C.I. of the multi-dimensional skip graph is higher than the upper bound of the 95% C.I. of the standard skip graph. The latter indicates the opposite of the values marked in gray. That is, it indicates that the mean node count of the multi-Dimensional skip graph is higher than that of the standard skip graph with 95% confidence. These are marked in bold. Figure 23 and Table 4 shows the results of a 10XXXXXXXXXX query.

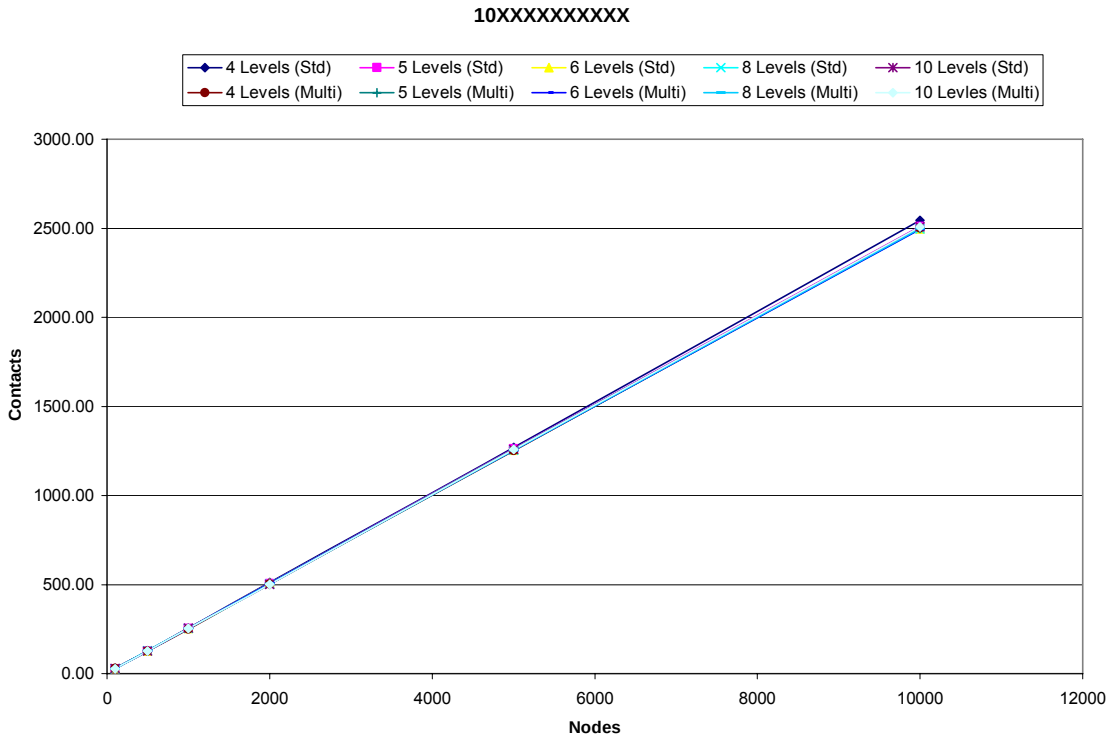


Figure 23. 10XXXXXXXXXX Query Results

Table 4. 10XXXXXXXXXX Mean Node Contacts 95% Confidence Intervals

		Std			Multi			
Levels	Nodes	L.L.	Mean	U.L.	L.L.	Mean	U.L.	Difference
4	100	28.14	29.04	29.94	27.96	28.97	29.98	-1.84
4	500	126.94	128.75	130.56	125.72	127.48	129.24	-2.30
4	1000	250.85	253.81	256.77	247.20	250.08	252.96	-2.11
4	2000	507.55	511.43	515.31	500.01	503.68	507.35	0.20
4	5000	1262.61	1269.46	1276.31	1245.29	1250.79	1256.29	6.32
4	10000	2531.33	2543.63	2555.93	2492.67	2501.24	2509.81	21.52
5	100	26.72	27.53	28.34	27.27	28.30	29.33	-2.61
5	500	125.52	127.41	129.30	125.30	127.30	129.30	-3.78
5	1000	251.60	254.20	256.80	251.22	253.87	256.52	-4.92
5	2000	501.47	505.32	509.17	500.29	504.07	507.85	-6.38
5	5000	1257.51	1264.00	1270.49	1253.57	1260.02	1266.47	-8.96
5	10000	2499.70	2509.53	2519.36	2490.30	2499.87	2509.44	-9.74
6	100	27.47	28.39	29.31	27.62	28.60	29.58	-2.11
6	500	125.40	127.25	129.10	125.42	127.39	129.36	-3.96
6	1000	251.53	254.33	257.13	251.19	254.09	256.99	-5.46
6	2000	505.31	508.97	512.63	504.62	508.31	512.00	-6.69
6	5000	1248.33	1254.31	1260.29	1248.06	1254.01	1259.96	-11.64
6	10000	2487.85	2495.68	2503.51	2486.25	2494.06	2501.87	-14.01
8	100	27.15	28.13	29.11	27.26	28.19	29.12	-1.96
8	500	126.84	128.53	130.22	126.64	128.39	130.14	-3.30
8	1000	250.45	253.36	256.27	250.84	253.83	256.82	-6.37
8	2000	498.88	502.99	507.10	498.69	502.79	506.89	-8.01
8	5000	1246.79	1252.95	1259.11	1247.25	1253.29	1259.33	-12.54
8	10000	2489.10	2497.62	2506.14	2489.23	2497.78	2506.33	-17.23
10	100	27.40	28.35	29.30	26.74	27.76	28.78	-1.37
10	500	125.39	127.23	129.07	125.53	127.33	129.13	-3.75
10	1000	252.55	255.11	257.67	252.40	255.10	257.80	-5.25
10	2000	498.16	502.09	506.02	497.66	501.61	505.56	-7.40
10	5000	1252.88	1258.95	1265.02	1252.93	1259.03	1265.13	-12.25
10	10000	2498.40	2507.11	2515.82	2498.40	2507.12	2515.84	-17.44

There is little differentiation between the ten data sets. The query is so large (25% of the total unit area) that the number of nodes involved in the query is almost equal to the number of nodes returned in the query for both skip graph types. This is logical, because the query quickly propagates to whichever quadrant is being queried, and then notifies all of the nodes within that quadrant. There are only three scenarios in which the multi-dimensional skip graph performs better than the standard skip graph – in large node

count networks with skip graphs with 4 levels. Figure 24 shows the results of a 1001XXXXXXXX query.

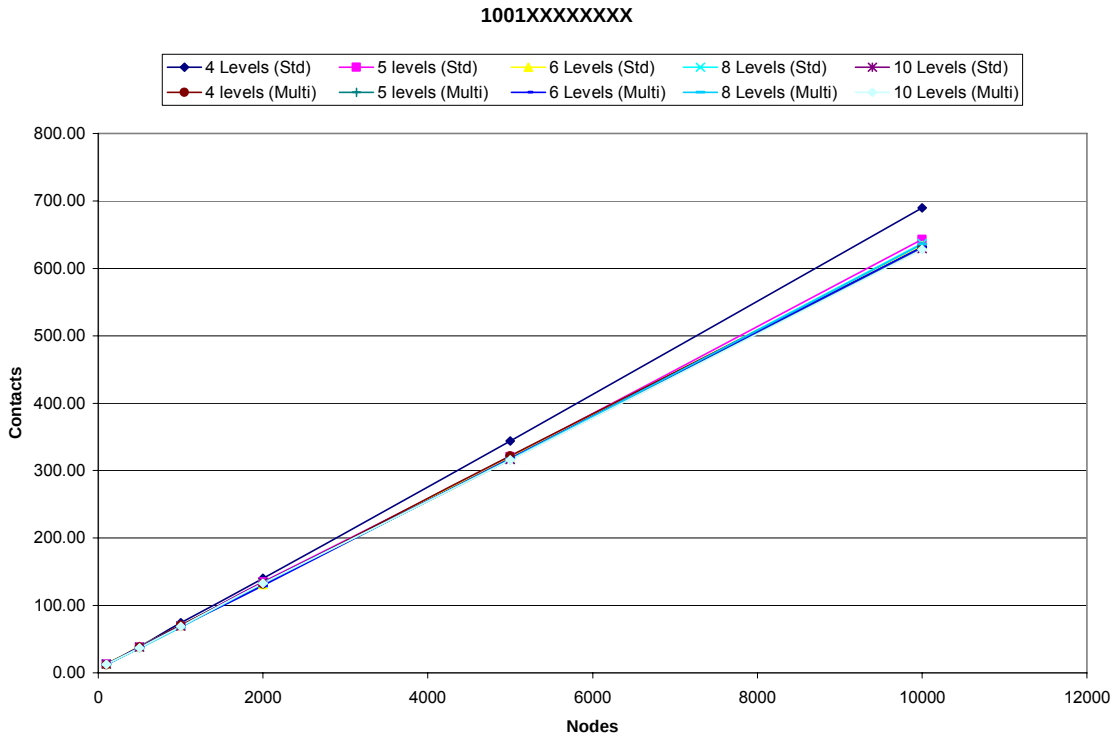


Figure 24. 1001XXXXXXXX Query Results

Figure 24 and Table 5 indicate some differences between the two skip graph types. Similar to the 10XXXXXXXXXX query, the multi-dimensional skip graph with 4 levels and node counts greater than 1000 outperform the standard skip graph. Multi-dimensional skip graph query performance begins to outperform the standard skip graph performance in 5 level skip graphs when node counts increase to 10000. There are no situations where the standard skip graph outperforms multi-dimensional skip graphs.

Table 5. 1001XXXXXXXX Mean Node Contacts 95% Confidence Intervals

		Std			Multi			
Levels	Nodes	L.L.	Mean	U.L.	L.L.	Mean	U.L.	Difference
4	100	12.31	12.95	13.59	11.39	12.21	13.03	-0.72
4	500	37.39	38.72	40.05	36.97	38.40	39.83	-2.44
4	1000	72.72	74.59	76.46	68.76	70.58	72.40	0.32
4	2000	137.24	140.16	143.08	129.53	131.85	134.17	3.07
4	5000	338.55	343.86	349.17	318.07	322.06	326.05	12.50
4	10000	679.94	690.01	700.08	626.00	630.80	635.60	44.34
5	100	12.28	13.02	13.76	11.60	12.61	13.62	-1.34
5	500	37.15	38.42	39.69	35.95	37.22	38.49	-1.33
5	1000	67.84	69.38	70.92	67.02	68.79	70.56	-2.72
5	2000	132.62	134.93	137.24	130.46	132.67	134.88	-2.26
5	5000	317.51	320.74	323.97	313.41	316.34	319.27	-1.76
5	10000	637.78	642.98	648.18	625.99	630.94	635.89	1.88
6	100	12.56	13.44	14.32	11.45	12.30	13.15	-0.59
6	500	37.67	39.18	40.69	36.12	37.61	39.10	-1.43
6	1000	68.17	69.87	71.57	67.10	68.84	70.58	-2.41
6	2000	128.63	131.28	133.93	127.30	129.73	132.16	-3.54
6	5000	317.45	320.48	323.51	315.60	318.42	321.24	-3.79
6	10000	629.53	634.19	638.85	627.12	631.85	636.58	-7.04
8	100	12.27	13.01	13.75	11.35	12.24	13.13	-0.85
8	500	36.87	38.10	39.33	35.28	36.53	37.78	-0.91
8	1000	68.67	70.36	72.05	66.23	67.87	69.51	-0.84
8	2000	130.61	132.92	135.23	130.15	132.37	134.59	-3.98
8	5000	313.42	317.05	320.68	313.23	316.87	320.51	-7.09
8	10000	632.04	637.11	642.18	630.75	635.72	640.69	-8.65
10	100	12.26	13.07	13.88	11.41	12.36	13.31	-1.04
10	500	37.24	38.53	39.82	35.95	37.21	38.47	-1.23
10	1000	68.05	69.69	71.33	66.99	68.85	70.71	-2.67
10	2000	131.96	134.33	136.70	130.81	133.10	135.39	-3.43
10	5000	313.52	316.70	319.88	312.65	315.82	318.99	-5.48
10	10000	624.71	629.35	633.99	623.40	628.00	632.60	-7.89

Figure 25 shows the results of the 100110XXXXXX query execution simulation. It is apparent from the figure that 4 level standard skip graphs perform this query poorly compared to multi-dimensional skip graphs as node count increases. Table 6 confirms the graphical analysis. In this query range, 4 level multi-dimensional skip graphs outperform standard skip graphs in networks with node counts greater than 1000. The C.I. ranges of

the mean node counts for these scenarios do not overlap, indicating the difference is statistically significant.

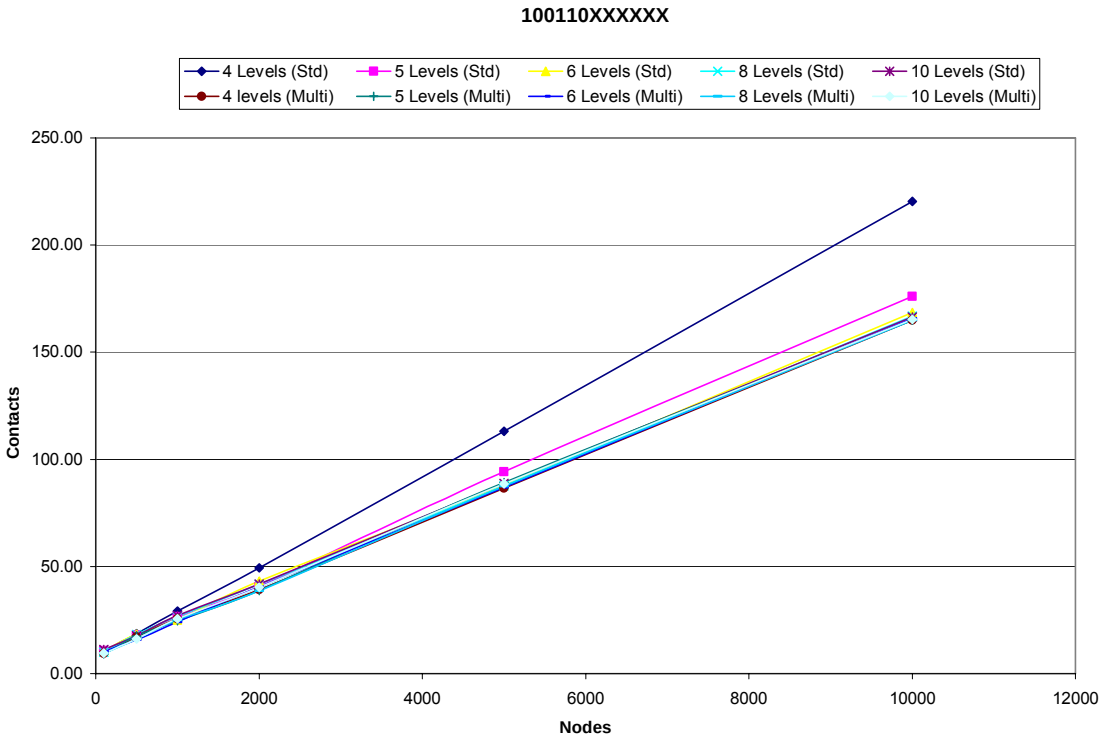


Figure 25. 100110XXXXXX Query Results

The main reason for the poor performance in the 4-level standard skip graph is as the node count increases, so does the length of the linked-lists in each level of the skip graph. The two different query methods for each type of skip graph impacts query performance, especially as node counts increase. In a standard skip graph, which starts the query at the topmost level, it is possible to traverse several nodes before dropping down to a lower level. A list is traversed as long as the query is approaching the specified key range and that the conditions for dropping down to a lower level have not been met

(Section 2.3). Until those conditions are met, the query is forced to continue to transfer to adjacent nodes at higher levels.

Table 6. 100110XXXXXX Mean Node Contacts 95% Confidence Intervals

		Std			Multi			
Levels	Nodes	L.L.	Mean	U.L.	L.L.	Mean	U.L.	Difference
4	100	9.68	10.53	11.38	8.65	9.50	10.35	-0.68
4	500	17.52	18.73	19.94	15.98	17.23	18.48	-0.97
4	1000	27.81	29.31	30.81	24.43	25.60	26.77	1.04
4	2000	46.92	49.32	51.72	37.56	39.14	40.72	6.21
4	5000	108.46	113.06	117.66	84.50	86.57	88.64	19.82
4	10000	211.10	220.42	229.74	162.35	164.88	167.41	43.69
5	100	9.84	10.81	11.78	8.21	9.16	10.11	-0.27
5	500	17.04	18.12	19.20	16.37	17.60	18.83	-1.78
5	1000	25.23	26.54	27.85	23.56	24.74	25.92	-0.69
5	2000	39.41	40.79	42.17	37.38	38.93	40.48	-1.07
5	5000	91.91	94.22	96.53	87.28	89.20	91.12	0.79
5	10000	172.68	176.08	179.48	162.29	164.88	167.47	5.21
6	100	9.84	10.77	11.70	9.35	10.30	11.25	-1.40
6	500	17.02	18.40	19.78	14.78	15.79	16.80	0.22
6	1000	24.00	25.10	26.20	22.91	24.19	25.47	-1.48
6	2000	41.45	43.10	44.75	38.92	40.45	41.98	-0.53
6	5000	86.34	88.31	90.28	85.08	86.94	88.80	-2.47
6	10000	165.86	168.54	171.22	163.20	165.68	168.16	-2.30
8	100	9.24	10.00	10.76	8.50	9.37	10.24	-0.99
8	500	16.98	18.24	19.50	15.15	16.27	17.39	-0.42
8	1000	25.52	26.85	28.18	23.86	25.08	26.30	-0.78
8	2000	40.17	41.71	43.25	37.15	38.65	40.15	0.02
8	5000	85.62	87.66	89.70	85.41	87.46	89.51	-3.89
8	10000	163.96	166.87	169.78	162.10	165.03	167.96	-4.00
10	100	10.36	11.29	12.22	8.69	9.57	10.45	-0.10
10	500	16.71	17.81	18.91	15.19	16.28	17.37	-0.66
10	1000	25.72	27.10	28.48	24.27	25.63	26.99	-1.27
10	2000	40.33	41.81	43.29	39.01	40.26	41.51	-1.17
10	5000	86.92	88.83	90.74	86.56	88.68	90.80	-3.87
10	10000	163.90	166.53	169.16	162.60	165.27	167.94	-4.04

As the node count increases in a standard skip graph with a fixed number of levels, the lists at the topmost level can become quite long, forcing the query to traverse extraneous nodes before dropping down to a lower level. In a multi-dimensional skip graph with a fixed number of levels, extraneous nodes are still contacted, but the process

is quite different. Recall that a multi-dimensional skip graph query begins at the base level L_0. The query traverses in both directions, looking for a matching prefix in the membership vector of each node. The node keys are random, and so the probability a matching prefix will be found is a random variable. As Figure 25 shows, multi-dimensional skip graph querying outperforms the standard skip graph's querying mechanism for the scenarios specified in the previous paragraphs.

Figure 26 and Table 7 show the results from executing the query 10011000XXXX. These results differ from the previous figures, in that there are now scenarios in which standard skip graphs outperform multi-dimensional skip graphs. The reason for this is related to the precision of the query.

Each hierarchical sub-quadrant of the query is a level in the skip graph. Figure 11 and Figure 12 help to visualize this concept. A query of the entire unit area will return all nodes, i.e. the base level of the skip graph. A query of 10XXXXXXXXXX returns the proper sub-list in L_1, that being the list containing node 17 as shown in Figure 12. This corresponds to the geographical layout in Figure 11; node 17 is the only node in the quadrant "10". A query of 10011000XXXX is a perfectly legal query to execute, however performance is hindered severely for multi-dimensional skip graphs of four levels or less. The precision of the query is such that instead of returning an entire linked-list in the skip graph, the query must traverse the entire list, returning only those nodes whose membership vectors match the entire prefix of the query.

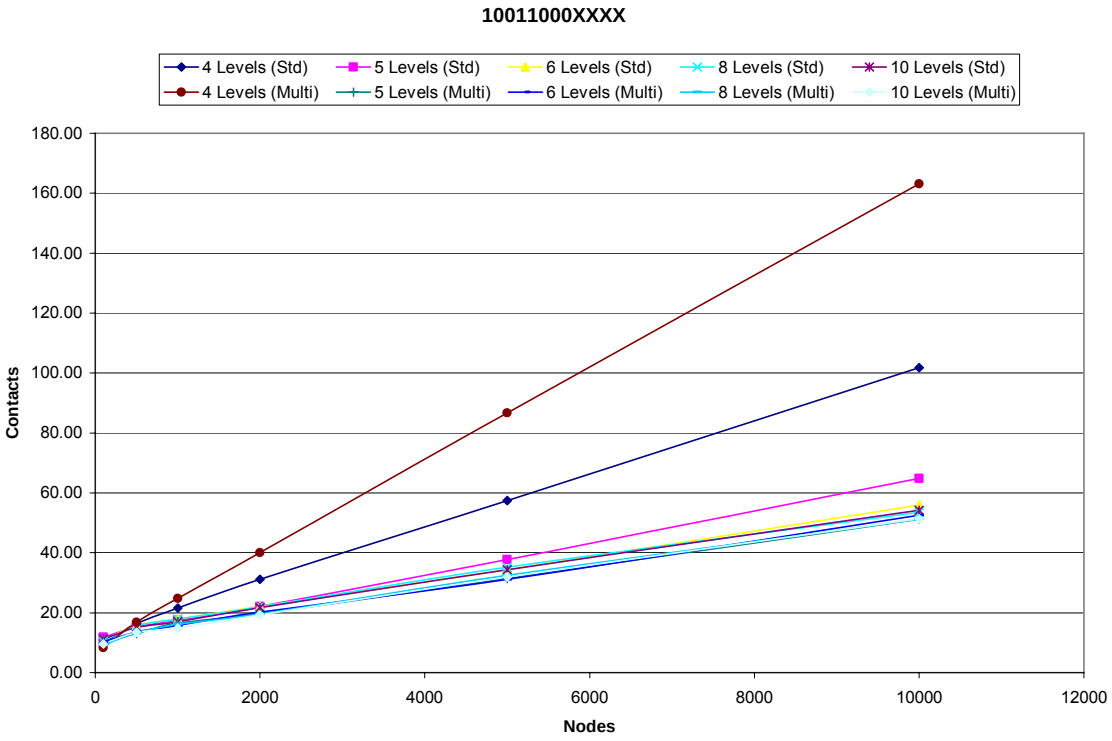


Figure 26. 10011000XXXX Query Results

The explanation above is further reinforced when the scenarios of skip graph levels greater than 5 are examined. For these scenarios, the multi-dimensional skip graph query performs comparably to the standard skip graph, as Table 7 indicates.

The effects suffered from extra query precision are not as prominent in the standard skip graph query results, as Figure 26 illustrates. A standard skip graph query drops down to the base level when it finds a node with a key value within range. Since the nodes are sorted in the base list, the query traverses in both directions until finding nodes with key values outside of the query range. The query does not waste time traversing through nodes that are ultimately not included in the final query results.

Table 7. 10011000XXXX Mean Node Contacts 95% Confidence Intervals

		Std			Multi			
Levels	Nodes	L.L.	Mean	U.L.	L.L.	Mean	U.L.	Difference
4	100	9.18	10.02	10.86	7.65	8.41	9.17	0.01
4	500	15.05	16.28	17.51	15.62	16.90	18.18	-3.13
4	1000	19.93	21.52	23.11	23.60	24.78	25.96	-6.03
4	2000	28.99	31.08	33.17	38.59	40.03	41.47	-12.49
4	5000	53.03	57.45	61.87	84.72	86.67	88.62	-35.59
4	10000	92.77	101.85	110.93	160.55	163.20	165.85	-73.08
5	100	10.80	11.82	12.84	9.54	10.57	11.60	-0.80
5	500	14.06	15.27	16.48	11.99	13.14	14.29	-0.23
5	1000	16.38	17.59	18.80	15.14	16.54	17.94	-1.56
5	2000	20.78	22.13	23.48	18.30	19.66	21.02	-0.24
5	5000	35.68	37.78	39.88	30.01	31.40	32.79	2.89
5	10000	62.09	64.92	67.75	49.31	51.26	53.21	8.88
6	100	9.92	10.74	11.56	9.17	10.23	11.29	-1.37
6	500	14.60	15.79	16.98	12.42	13.64	14.86	-0.26
6	1000	16.61	17.92	19.23	14.44	15.82	17.20	-0.60
6	2000	20.96	22.07	23.18	18.71	20.18	21.65	-0.69
6	5000	32.87	34.40	35.93	29.49	31.10	32.71	0.16
6	10000	54.09	55.91	57.73	50.79	52.56	54.33	-0.24
8	100	9.64	10.46	11.28	8.61	9.36	10.11	-0.47
8	500	14.59	15.68	16.77	11.89	13.19	14.49	0.10
8	1000	16.51	17.89	19.27	14.82	16.16	17.50	-0.99
8	2000	20.64	22.00	23.36	18.12	19.50	20.88	-0.24
8	5000	33.71	35.23	36.75	30.87	32.50	34.13	-0.41
8	10000	51.89	53.47	55.05	49.38	51.22	53.06	-1.17
10	100	10.19	11.23	12.27	8.51	9.58	10.65	-0.46
10	500	14.09	15.09	16.09	12.14	13.43	14.72	-0.62
10	1000	15.99	16.98	17.97	13.73	14.79	15.85	0.14
10	2000	20.44	21.74	23.04	17.89	19.32	20.75	-0.30
10	5000	32.81	34.34	35.87	30.52	32.00	33.48	-0.67
10	10000	52.42	54.25	56.08	49.60	51.33	53.06	-0.63

Figure 27 and Table 8 illustrate the higher query precision issue further. As the query becomes more precise (1001100011XX), the multi-dimensional skip graph shows poorer performance at lower skip graph levels. This is the same problem as previously stated, but in these figures it is more pronounced.

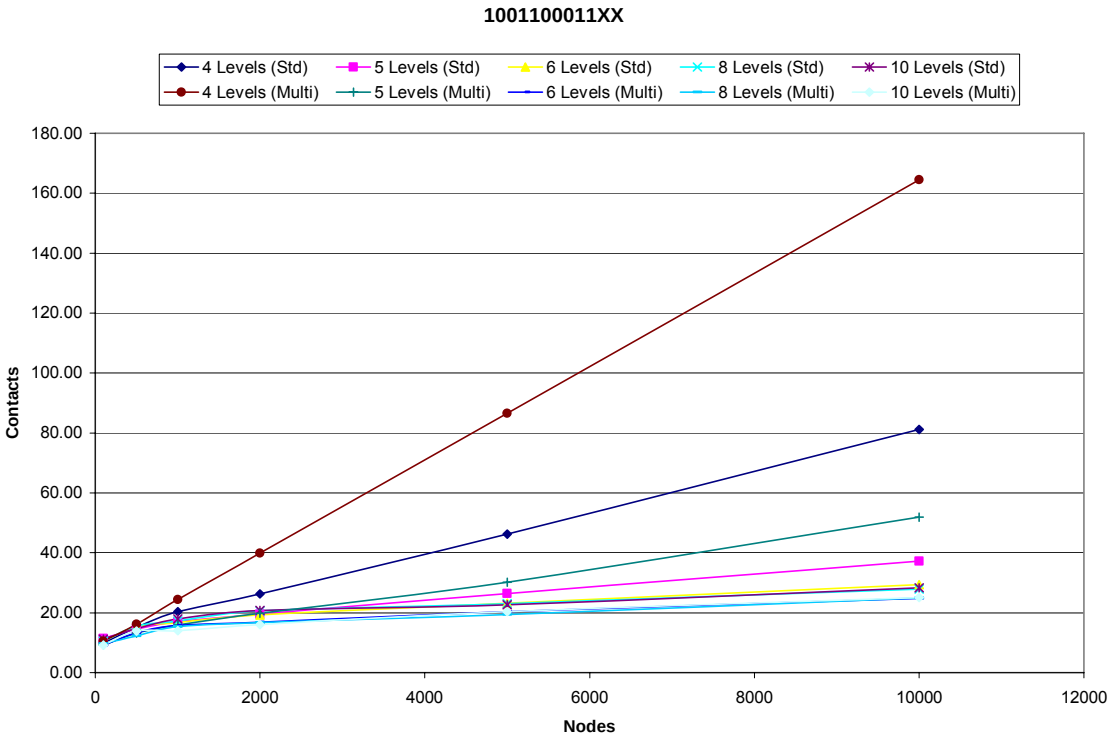


Figure 27. 1001100011XX Query Results

Note the bold data in Table 8. The data indicates that standard skip graphs outperform multi-dimensional skip graphs when executing a 1001100011XX with a 4 level skip graph and increasing node counts. Also, the 5 level skip graph simulations take a higher total node count in the network to observe standard skip graphs outperform multi-dimensional skip graphs for this query. As the skip graph level count increases to more than 6, the query performances of standard skip graphs become comparable to those of multi-dimensional skip graphs, and in some scenarios even outperformed by multi-dimensional skip graphs.

Table 8. 1001100011XX Mean Node Contacts 95% Confidence Intervals

		Std			Multi			
Levels	Nodes	L.L.	Mean	U.L.	L.L.	Mean	U.L.	Difference
4	100	9.04	9.98	10.92	9.59	10.55	11.51	-2.47
4	500	13.90	15.14	16.38	15.10	16.19	17.28	-3.37
4	1000	18.58	20.37	22.16	23.16	24.36	25.56	-6.98
4	2000	24.34	26.34	28.34	38.41	39.97	41.53	-17.20
4	5000	41.66	46.23	50.80	84.64	86.53	88.42	-46.75
4	10000	72.15	81.20	90.25	161.93	164.48	167.03	-94.88
5	100	10.52	11.49	12.46	8.37	9.28	10.19	0.33
5	500	13.44	14.56	15.68	12.00	13.20	14.40	-0.96
5	1000	15.77	17.03	18.29	14.79	15.91	17.03	-1.27
5	2000	18.34	19.56	20.78	18.41	19.80	21.19	-2.85
5	5000	24.57	26.42	28.27	28.77	30.19	31.61	-7.04
5	10000	34.79	37.15	39.51	50.20	51.86	53.52	-18.73
6	100	9.78	10.62	11.46	8.26	9.18	10.10	-0.33
6	500	14.25	15.40	16.55	11.98	13.34	14.70	-0.45
6	1000	15.35	16.67	17.99	14.35	15.72	17.09	-1.74
6	2000	17.96	19.27	20.58	15.49	16.83	18.17	-0.21
6	5000	21.73	23.17	24.61	18.87	20.31	21.75	-0.03
6	10000	27.93	29.36	30.79	23.35	24.77	26.19	1.74
8	100	9.74	10.58	11.42	8.66	9.74	10.82	-1.08
8	500	14.37	15.56	16.75	11.05	12.21	13.37	1.00
8	1000	15.91	17.23	18.55	13.92	15.34	16.76	-0.85
8	2000	19.07	20.59	22.11	15.16	16.66	18.16	0.92
8	5000	21.53	23.05	24.57	17.77	19.41	21.05	0.48
8	10000	26.40	27.91	29.42	23.30	24.97	26.64	-0.24
10	100	10.26	11.23	12.20	8.20	9.16	10.12	0.14
10	500	13.71	14.76	15.81	12.53	13.88	15.23	-1.52
10	1000	16.48	17.92	19.36	12.80	14.07	15.34	1.15
10	2000	19.24	20.82	22.40	14.69	16.07	17.45	1.79
10	5000	21.06	22.61	24.16	18.75	20.41	22.07	-1.01
10	10000	26.70	28.28	29.86	23.45	25.10	26.75	-0.05

The results of the final query range of 100110001101 are shown in Figure 28 and Table 9. The results are similar to those of Figure 27 and Table 8, with similar explanations for why the results look they way they do.

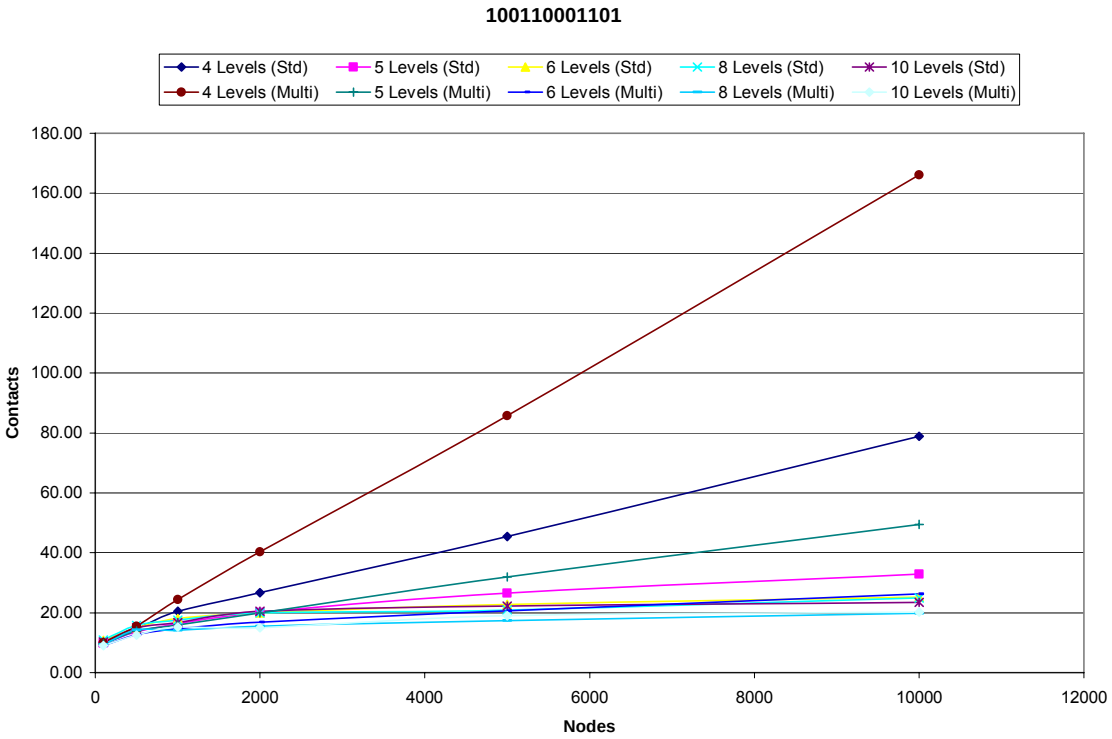


Figure 28. 100110001101 Query Results

Figure 23 through Figure 28 and Table 4 through Table 9 provide interesting insight into the relationship between query precision and skip graph level count. The figures and tables illustrate that if query precision increases and the number of skip graph levels remain the same, an increase in extraneous node contacts occur. These extraneous node contacts degrade query performance because the increased response time needed for all of the extra messages to be sent. Both types of skip graphs suffer in query performance if the precision is higher than the number of skip graph levels.

Table 9. 100110001101 Mean Node Contacts 95% Confidence Intervals

		Std			Multi			
Levels	Nodes	L.L.	Mean	U.L.	L.L.	Mean	U.L.	Difference
4	100	9.27	10.13	10.99	9.01	9.96	10.91	-1.64
4	500	13.55	14.55	15.55	14.46	15.55	16.64	-3.09
4	1000	18.83	20.46	22.09	23.00	24.41	25.82	-6.99
4	2000	24.42	26.76	29.10	38.86	40.35	41.84	-17.41
4	5000	41.03	45.48	49.93	83.78	85.76	87.74	-46.71
4	10000	70.71	78.82	86.93	163.35	166.10	168.85	-98.14
5	100	9.02	9.90	10.78	8.62	9.63	10.64	-1.62
5	500	12.65	13.68	14.71	12.57	13.86	15.15	-2.50
5	1000	15.18	16.33	17.48	14.54	15.90	17.26	-2.08
5	2000	18.63	20.04	21.45	18.45	19.89	21.33	-2.70
5	5000	24.75	26.60	28.45	30.24	31.97	33.70	-8.95
5	10000	30.29	32.96	35.63	47.35	49.46	51.57	-21.28
6	100	9.85	10.74	11.63	8.02	8.97	9.92	-0.07
6	500	13.82	15.03	16.24	11.52	12.66	13.80	0.02
6	1000	16.86	18.12	19.38	13.36	14.73	16.10	0.75
6	2000	18.47	19.92	21.37	15.33	16.83	18.33	0.14
6	5000	21.31	22.78	24.25	19.10	20.62	22.14	-0.83
6	10000	23.77	25.25	26.73	24.80	26.31	27.82	-4.05
8	100	9.97	10.88	11.79	8.40	9.43	10.46	-0.49
8	500	14.63	15.78	16.93	12.96	14.31	15.66	-1.02
8	1000	16.22	17.57	18.92	13.03	14.18	15.33	0.89
8	2000	18.74	20.02	21.30	14.13	15.55	16.97	1.77
8	5000	19.51	20.91	22.31	16.03	17.42	18.81	0.70
8	10000	23.29	25.00	26.71	18.35	19.83	21.31	1.98
10	100	9.34	10.27	11.20	8.21	9.09	9.97	-0.63
10	500	13.91	14.92	15.93	11.31	12.50	13.69	0.22
10	1000	15.54	16.71	17.88	13.87	15.25	16.63	-1.10
10	2000	19.01	20.53	22.05	13.65	15.00	16.35	2.66
10	5000	20.67	22.25	23.83	17.60	19.02	20.44	0.24
10	10000	21.83	23.47	25.11	18.65	20.22	21.79	0.05

Another conclusion that can be drawn from the six figures is that increasing the skip graph level count leads to similar performances for both types of skip graphs. That is, if the number of skip graph levels is above the required number for efficient query performance for a particular precision, increasing the number of skip graph levels while keeping total node count fixed causes the query performances of both skip graphs to be comparable. In that situation, the total node count in the network needs to increase to see

any gains in query performance. Table 6 illustrates this. The multi-dimensional skip graph outperforms the standard skip graph in almost all node counts above 1000 nodes in the 4 level scenarios. In the 5 level scenarios immediately below those, node counts need to increase to over 5000 to see similar performance gains by the multi-dimensional skip graph.

Table 10 summarizes the analysis of all the query simulations performed. The values inside the grid indicate which skip graph performs better (as node counts increase) at the specified number of skip graph levels and precision of query.

Table 10. Summary of Query Performances

Query Precision	Number of Levels				
	4	5	6	8	10
10XXXXXXXXXX	**Multi**	either	either	either	either
1001XXXXXXXX	**Multi**	**Multi**	either	either	either
100110XXXXXX	**Multi**	**Multi**	either	either	either
10011000XXXX	Standard	**Multi**	**Multi**	either	either
1001100011XX	Standard	Standard	either	**Multi**	**Multi**
100110001101	Standard	Standard	either	**Multi**	**Multi**

At low level counts (e.g., 4) and low query precision (e.g., 10XXXXXXXXXX), multi-dimensional skip graphs outperform standard skip graphs, especially as node counts increase. As query precision increases at low level counts, the performance of standard skip graphs increase compared with the multi-dimensional skip graphs. As level count increases, with low precision queries the query performances of both skip graph types are comparable. Finally, as level count increases and higher precision queries are executed, the multi-dimensional skip graph outperforms the standard skip graph.

4.3 Analysis of Mobility

The secondary goal of this research pertains to node mobility. Mobility is an important factor to consider when discussing the design of skip graphs. Chapter 2 is a summary of the skip graph invariants that must hold true at all times. Refer to [AsS03] for a full presentation on maintaining a proper skip graph. Allowing nodes to move has a significant impact on the structure of a skip graph.

Consider a skip graph in which the keys and membership vectors are independently computed from geographical positioning. Mobile nodes will certainly cause pre-existing links to fail, which in turn invalidates neighbor node references in each level of the skip graph. These invalid references need to be removed, and new references need to be associated with the node's new position.

Now consider a skip graph in which the keys and/or membership vectors are computed directly from geographical position, as in this research. When a node moves to a new location grid its *z*-order geographical location value will change as well. In the standard skip graph, this changes the node's key value. In the multi-dimensional skip graph, this changes the node's membership vector.

The metric used to assess mobility impact is update message count. When a node determines that its key value or membership vector has changed, the node removes itself from the skip graph and attempts to reinsert it with the new values.

In a standard skip graph, a query must be performed on the skip graph to determine where in the base list at L_0 the node should needs to be inserted. A query must be performed because the node's key has changed and therefore its position in the base

list has changed. Then, the node adds itself to the proper list at each level. In a multi-dimensional skip graph, a query does not need to be performed because the node's key (a random value) will not change. The node does, however, need to add itself to the proper list at each level.

Figure 29 shows the results of the mobility simulation. A full factorial simulation is performed with levels of 4, 6, 8, and 10, with node counts of 100, 500, 1000, 2000, 5000, and 10000. Similar to the query simulations, each simulation iterates 100 times. In each of the iterations, a new skip graph of each type is constructed and populated with random node layouts. A single node is chosen at random, and its coordinates are changed to other random coordinates. It doesn't matter how far the node is being moved, the same node update process is followed.

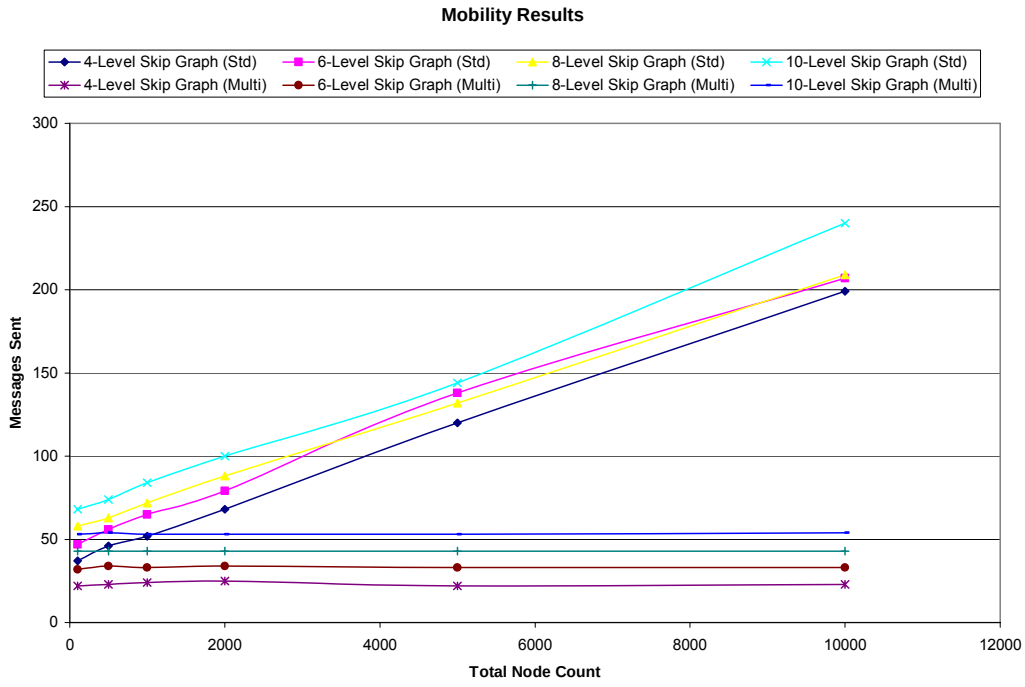


Figure 29. Mobility Results

Figure 29 contains 8 data sets. The four data sets that increase linearly are the results of the standard skip graph simulation. It's evident that as total node count in the network is increased, it takes more update messages to fix the skip graph. Also note that the four data sets that have a relatively flat response. Those are multi-dimensional skip graph simulations. As the total node count in the network increases, it takes about the same amount of update messages to fix the skip graph.

The primary reason for the two different responses is that a node insertion into a standard skip graph must perform a query within the network to determine its position in the base list L_0. As network size increases, that query requires a greater number of messages to execute. An increase in messages sent implies an increase in the time it takes to insert the node.

Likewise, a node insertion into a multidimensional query does not need to perform a query within the network. The node's key does not change, so its position in the base list remains the same. The message counts result from the node's re-assignment to new linked lists in each level of the skip graph due to the change in the node's membership vector. These re-assignment message counts are significantly smaller than the messages incurred from executing a query within the entire network.

The small increase in the number of messages sent as the number of levels in the skip graph are increased are because more messages need to be sent as the node iterates through the levels during an insertion.

Figure 29 shows that multi-dimensional skip graphs are well suited in mobile networks due to the speed of completing the fix of the skip graph compared with a

standard skip graph. The amount of time it takes to fix a skip graph is directly proportional to the number of messages that are sent.

4.4 Summary

This chapter analyzes query execution of both standard and multi-dimensional skip graphs. Model validation analysis ensures the model implementation is correct. An analysis of the simulation of six different query ranges is performed, as well as an analysis on a mobility simulation.

V. Conclusions and Recommendations

This chapter presents the conclusions of this research. A summary of the conclusions drawn from the simulation results prefaces a discussion of the significance of those conclusions. Finally, recommendations for future work are discussed.

5.1 Problem Summary

Skip graphs are application layer-based distributed data structure intended for use in a sensor network where users query sensor data. Nodes in a standard skip graph are sorted by their key value into successively smaller groups where group membership is based on random membership vectors computed locally at each node. Multi-dimensional skip graphs invert the node key and membership vector roles, whereby group membership is computed deterministically and node keys are computed randomly.

Simulations determine if this role reversal enables multi-dimensional skip graphs to increase query performance. Query performance is determined by the number of skip graph messages sent throughout the network to accomplish the query. A greater number of messages lead to an increase in network traffic, and longer response times to finish the query. It is this metric that is minimized. Simulations also determine whether this role reversal affects performance in light of node mobility.

5.2 Summary of Conclusions Drawn From Results

The results show that the proposed multi-dimensional skip graph executes queries better under certain conditions. First, the number of levels in the skip graph must be greater than the precision of the query for multi-dimensional skip graphs to have any chance of outperforming standard skip graphs. Chapter 4 shows that if this condition is

not met, the performance of multi-dimensional skip graph query execution suffers. In the simulations performed, the standard skip graph outperformed the multi-dimensional skip graph by 54.5% for a query precision of 6 levels (100110001101) running on 4 level skip graphs, with a node count of 10000 (cf. Figure 28 and Table 9). When the number of skip graph levels accommodates query precision, the multi-dimensional skip graph outperformed the standard skip graph by 8.7%. The scenario that this occurred at was a query precision of 3 levels (100110XXXXXX) running on 4 levels skip graphs, with a node count of 10000. Figure 25 and Table 6 show these results.

Second, if the number of skip graph levels accommodates query precision, multi-dimensional skip graphs outperform standard skip graphs as node counts increase. This is shown in all of the figures in which the query precision is equal to or less than the number of levels in the skip graph. As a corollary, as the number of skip graph levels are increased for each of the skip graph types, a higher node count is required to notice performance differences. In low node count networks, there is no significant impact on query performance gained by using one type of skip graph or the other. The query propagates through the network so fast that both skip graph types process the queries comparably.

Lastly, if the number of skip graph levels accommodates query precision, as the range of the query increases, the query performance of both skip graph types become comparable. The ratio of queried nodes to total nodes is so large that both skip graph types are able to process the queries comparably.

Multi-dimensional skip graphs are capable of processing mobile node updates within the skip graph with fewer skip graph messages. In a 10,000 node network, multi-

dimensional skip graphs process a mobile node's skip graph update with 4X fewer messages. When a node changes geographical locations in the context of a standard skip graph, a query is required to re-assign the node in the proper position in the base list in L_0. This query is what causes poor mobility performance in a standard skip graph. Without this query, a multi-dimensional skip graph has an almost flat response as node count increases. This is in stark contrast to a standard skip graph's response to increasing node counts: a corresponding increase in update message counts.

5.3 Significance of Research

DSN technology research is still in its infancy. This research explores avenues that might lead to more efficient networks. It explores an alternative structure to skip graphs and shows gains could be made if certain query and skip graph conditions are met. The research adds to the existing body of work relating to skip graphs, advancing the state of the art in skip graphs as applied to distributed sensor networks.

5.4 Recommendations for Future Research

This research has several opportunities for future research. Well-structured Java skip graph models were developed that future research can use. Java allows the models created in this research to be easily extensible into other versions of skip graphs. The separation of the model from the simulation controller means skip graph parameter adjustments can be made and more simulations can be run to evaluate the performance differences those adjustments caused.

One path of future research involves the amount of underlying network traffic that involved using skip graphs – both standard and multi-dimensional. Recall that skip

graphs reside at the application layer, they are ignorant of the underlying structure of the routing layer. It would be interesting to analyze how the underlying network infrastructure is affected using standard or multi-dimensional skip graphs. That particular information is not addressed in this research, because it is assumed those network costs are worth the performance gained by using skip graphs, both standard and multi-dimensional.

Another research path that could be explored is the implementation of a hybrid skip graph – a combination of the standard and multi-dimensional skip graphs. The results of this research indicate that there are scenarios in which the multi-dimensional skip graph outperforms the standard skip graph, but performance suffers when the previously stated conditions are not met. An analysis of when skip graph usage should switch between the two types would be beneficial.

5.5 Summary

This chapter presents the conclusions from the simulation results. It discusses the impact this research has, and offers suggestions for future research.

Appendix A: Java Source Code

```
//Controller Class
package gjb.thesis.controller;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Random;
import org._3pq.jgrapht.graph.ListenableDirectedGraph;
import gjb.thesis.model.MultiSkipGraph;
import gjb.thesis.model.Node;
import gjb.thesis.model.SkipGraph;
import gjb.thesis.model.StdSkipGraph;
import gjb.thesis.strategy.RandomKey;
import gjb.thesis.strategy.RandomMX;
/**
 * This is the Controller Class. The methods are called from the View
 * Class in response to menu clicks. It's a Singleton Class.
 * @author Greg Brault
 */
public class SkipGraphController {
    private SkipGraph sg = null;
    private int f_numLevels = 4;
    private static final int STD = 0;
    private static final int GJB = 1;
    private static final int KNOWNLAYOUT = 2;
    private static final int RNDLAYOUT = 3;
    private static SkipGraphController ref = null;

    SkipGraphController() {
        sg = new StdSkipGraph(f_numLevels);
    }

    /**
     * The Singleton Constructor
     * @return the Reference to the object
     */
    public static SkipGraphController getInstance() {
        if (ref == null) {
            ref = new SkipGraphController();
            return ref;
        } else return ref;
    }

    /**
```

```
     * Inits a standard skip graph
     */
    public void initStd() {
        sg = new StdSkipGraph(f_numLevels);
        String zOrderMin = "10000000000000000000000000000000";
        String zOrderMax = "10111111111111111111111111111111";
        int xMin = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
        int yMin = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
        int xMax = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
        int yMax = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
        System.out.println("X range: " + xMin + " to " + xMax + ". Y range: " + yMin + "
to " + yMax);

        ArrayList<Node> nodes = buildNodes(STD, KNOWNLAYOUT, 0);
        LinkedList<Node> messages = new LinkedList<Node>();
        LinkedList<Node> route = new LinkedList<Node>();
        LinkedList preDist = new LinkedList();
        LinkedList postDist = new LinkedList();

        for (Node n : nodes) {
            n.setMX();
            sg.add(n);
            if (n.getId() % 2 == 0)
                System.out.print("Node " + n.getId() + " m(x) = " + n.getBinaryMX() + " | ");
            else System.out.println("Node " + n.getId() + " m(x) = " + n.getBinaryMX());
        }


        /*
        sg.search(nodes.get(17), nodes.get(10).getKey(), route, messages);
        System.out.println("Route:");
        for (Node n : route) {
            System.out.print(n.getId() + ", ");
        }
        System.out.println();
        System.out.println("Messages:");
        for (Node n : messages) {
            System.out.print(n.getId() + ", ");
        }
        System.out.println();
```

```
        LinkedList<Node> query = sg.rangeQueryStd(nodes.get(3), includedNodes,
preDist, postDist,                    xMin, yMin, xMax, yMax);
        System.out.print("Nodes involved in query (" + includedNodes.size() + "): ");
        for (Node n : includedNodes)
            System.out.print(n.getId() + ", ");
        System.out.println();
        System.out.print("Range query results (" + query.size() + "): ");
        for (Node n : query)
            System.out.print(n.getId() + ", ");
        System.out.println("Pre Distances:");
        for (Object dist : preDist)
            System.out.println((Double) dist);
        System.out.println("Post Distances:");
        for (Object dist : postDist)
            System.out.println((Double) dist);
        */
    }

    /**
     * Inits a multi-dimensional skip graph
     */
    public void initGjb() {
        sg = new MultiSkipGraph(f_numLevels);
        String zOrderMin = "10000000000000000000000000000000";
        String zOrderMax = "10111111111111111111111111111111";
        int xMin = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
        int yMin = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
        int xMax = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
        int yMax = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
        System.out.println("X range: " + xMin + " to " + xMax + ". Y range: " + yMin + "
to " + yMax);

        ArrayList<Node> nodes = buildNodes(GJB, KNOWNLAYOUT, 0);
        LinkedList<Node> includedNodes = new LinkedList<Node>();
        LinkedList preDist = new LinkedList();
        LinkedList postDist = new LinkedList();

        for (Node n : nodes) {
            n.setMX();
            sg.add(n);
            if (n.getId() % 2 == 0)
```

```
                System.out.print("Node " + n.getId() + " m(x) = " + n.getBinaryMX() + " | ");
            else System.out.println("Node " + n.getId() + " m(x) = " + n.getBinaryMX());
        }
        /*
        System.out.println("Node 0 key = " + nodes.get(0).getKey());
        System.out.println("x = " + nodes.get(0).getCoords()[0] + ", y = " +
nodes.get(0).getCoords()[1]);
        int newCoords[] = new int[3];
        newCoords[0] = 62000;
        newCoords[1] = 62000;
        nodes.get(0).setCoords(newCoords);
        nodes.get(0).setMX();
        //nodes.get(0).setKey();
        sg.updateNode(nodes.get(0));
        System.out.println("x = " + nodes.get(0).getCoords()[0] + ", y = " +
nodes.get(0).getCoords()[1]);
        System.out.println("Node 0 key = " + nodes.get(0).getKey());
        System.out.println("Node 0 m(x) = " + nodes.get(0).getBinaryMX());
        LinkedList<Node> query = sg.rangeQuery(nodes.get(10), includedNodes, preDist,
postDist, xMin,                yMin, xMax, yMax);
        System.out.print("Nodes involved in query (" + includedNodes.size() + "): ");
        for (Node n : includedNodes)
            System.out.print(n.getId() + ", ");
        System.out.println();
        System.out.print("Range query results (" + query.size() + "): ");
        for (Node n : query)
            System.out.print(n.getId() + ", ");
        System.out.println("Pre Distances:");
        for (Object dist : preDist)
            System.out.println((Double) dist);
        System.out.println("Post Distances:");
        for (Object dist : postDist)
            System.out.println((Double) dist);
        */
    }

    /**
     * Builds a skip graph based on the parameters
     * @param type Skip graph type (standard or multi-dim)
     * @param layout Known Layout or Randomized Layout
     * @param size # of nodes
     * @return the Skip graph
     */
    private ArrayList<Node> buildNodes(int type, int layout, int size) {
        ArrayList<Node> nodes = new ArrayList<Node>();
```

```
        if (layout == KNOWNLAYOUT) {
            if (type == STD) {
                nodes.add(0,  new Node(0,  new RandomMX(2), f_numLevels, 0, 0, 0));
                nodes.add(1,  new Node(1,  new RandomMX(2), f_numLevels, 33000,
33000, 0));
                nodes.add(2,  new Node(2,  new RandomMX(2), f_numLevels, 34000,
51000, 0));
                nodes.add(3,  new Node(3,  new RandomMX(2), f_numLevels,  6000,  4000,
0));
                nodes.add(4,  new Node(4,  new RandomMX(2), f_numLevels, 36000,
44000, 0));
                nodes.add(5,  new Node(5,  new RandomMX(2), f_numLevels, 39000,  7500,
0));
                nodes.add(6,  new Node(6,  new RandomMX(2), f_numLevels, 12000,
64000, 0));
                nodes.add(7,  new Node(7,  new RandomMX(2), f_numLevels,  3000, 12000,
0));
                nodes.add(8,  new Node(8,  new RandomMX(2), f_numLevels, 36000,
17000, 0));
                nodes.add(9,  new Node(9,  new RandomMX(2), f_numLevels,  5000, 22000,
0));
                nodes.add(10, new Node(10, new RandomMX(2), f_numLevels,  2500,
37000, 0));
                nodes.add(11, new Node(11, new RandomMX(2), f_numLevels,  4000,
60000, 0));
                nodes.add(12, new Node(12, new RandomMX(2), f_numLevels, 12000,
59000, 0));
                nodes.add(13, new Node(13, new RandomMX(2), f_numLevels, 34000,
4000, 0));
                nodes.add(14, new Node(14, new RandomMX(2), f_numLevels, 16000,
58000, 0));
                nodes.add(15, new Node(15, new RandomMX(2), f_numLevels, 34000,
63000, 0));
                nodes.add(16, new Node(16, new RandomMX(2), f_numLevels, 48000,
45000, 0));
                nodes.add(17, new Node(17, new RandomMX(2), f_numLevels, 54000,
7000, 0));
                nodes.add(18, new Node(18, new RandomMX(2), f_numLevels, 56000,
20000, 0));
                nodes.add(19, new Node(19, new RandomMX(2), f_numLevels, 27000,
27000, 0));
            } else if (type == GJB) {
                nodes.add(0,  new Node(0,  new RandomKey(2), f_numLevels,  2000,  2000,
0));
```

```
                nodes.add(1,  new Node(1,  new RandomKey(2), f_numLevels, 33000,
33000, 0));
                nodes.add(2,  new Node(2,  new RandomKey(2), f_numLevels, 34000,
51000, 0));
                nodes.add(3,  new Node(3,  new RandomKey(2), f_numLevels,  6000,  4000,
0));
                nodes.add(4,  new Node(4,  new RandomKey(2), f_numLevels, 36000,
44000, 0));
                nodes.add(5,  new Node(5,  new RandomKey(2), f_numLevels, 39000,  7500,
0));
                nodes.add(6,  new Node(6,  new RandomKey(2), f_numLevels, 12000,
64000, 0));
                nodes.add(7,  new Node(7,  new RandomKey(2), f_numLevels,  3000, 12000,
0));
                nodes.add(8,  new Node(8,  new RandomKey(2), f_numLevels, 36000,
17000, 0));
                nodes.add(9,  new Node(9,  new RandomKey(2), f_numLevels,  5000, 22000,
0));
                nodes.add(10, new Node(10, new RandomKey(2), f_numLevels,  2500,
37000, 0));
                nodes.add(11, new Node(11, new RandomKey(2), f_numLevels,  4000,
60000, 0));
                nodes.add(12, new Node(12, new RandomKey(2), f_numLevels, 12000,
59000, 0));
                nodes.add(13, new Node(13, new RandomKey(2), f_numLevels, 33000,
2000, 0));
                nodes.add(14, new Node(14, new RandomKey(2), f_numLevels, 16000,
58000, 0));
                nodes.add(15, new Node(15, new RandomKey(2), f_numLevels, 34000,
63000, 0));
                nodes.add(16, new Node(16, new RandomKey(2), f_numLevels, 48000,
45000, 0));
                nodes.add(17, new Node(17, new RandomKey(2), f_numLevels, 54000,
7000, 0));
                nodes.add(18, new Node(18, new RandomKey(2), f_numLevels, 56000,
20000, 0));
                nodes.add(19, new Node(19, new RandomKey(2), f_numLevels, 27000,
27000, 0));
            }
        } else if (layout == RNDLAYOUT) {
            if (type == STD) {
                for (int a = 0; a < size; a++) {
                    nodes.add(a, new Node(a, new RandomMX(2), f_numLevels));
                }
            } else if (type == GJB) {
```

```
                for (int a = 0; a < size; a++) {
                    nodes.add(a, new Node(a, new RandomKey(2), f_numLevels));
                }
            }
        }
        return nodes;
    }

    /**
     * Runs the main simulation
     */
    public void runExperiment() {
        Random startId = new Random();
        //double nodeCntStd = 0;
        //double nodeCntGjb = 0;
        //double querySizeStd = 0;
        //double querySizeGjb = 0;
        int xMin, xMax, yMin, yMax;
        String zOrderMin = null;
        String zOrderMax = null;
        SkipGraph sgExpStd = null;
        SkipGraph sgExpGjb = null;

        int numIterations = 100;
        int numNodes = 0;

        for (int d = 2; d < 3; d++) {
            switch (d) {
            case 0:
                zOrderMin = new String("10000000000000000000000000000000");
                zOrderMax = new String("10111111111111111111111111111111");
                xMin = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                yMin = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                xMax = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                yMax = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                break;
            case 1:
                zOrderMin = new String("10010000000000000000000000000000");
                zOrderMax = new String("10011111111111111111111111111111");
                xMin = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
```

```
                yMin = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                xMax = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                yMax = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                break;
            case 2:
                zOrderMin = new String("10011000000000000000000000000000");
                zOrderMax = new String("10011011111111111111111111111111");
                xMin = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                yMin = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                xMax = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                yMax = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                break;
            case 3:
                zOrderMin = new String("10011000000000000000000000000000");
                zOrderMax = new String("10011000111111111111111111111111");
                xMin = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                yMin = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                xMax = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                yMax = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                break;
            case 4:
                zOrderMin = new String("10011000110000000000000000000000");
                zOrderMax = new String("10011000111111111111111111111111");
                xMin = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                yMin = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                xMax = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                yMax = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                break;
            case 5:
                zOrderMin = new String("10011000110100000000000000000000");
```

```
                zOrderMax = new String("10011000110111111111111111111111");
                xMin = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                yMin = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMin,
2)).intValue());
                xMax = Node.zOrderX(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                yMax = Node.zOrderY(Long.valueOf(Long.parseLong(zOrderMax,
2)).intValue());
                break;
            default:
                zOrderMin = new String();
                zOrderMax = new String();
                xMin = 0;
                yMin = 0;
                xMax = 0;
                yMax = 0;
                break;
            }
            System.out.println("X range: " + xMin + " to " + xMax + ". Y range: " + yMin +
" to " + yMax);
            System.out.println("Z-order min: " + zOrderMin);
            System.out.println("Z-order max: " + zOrderMax);
            System.out.println("** Type * Iter * Levels *  Nodes  * Contacts * Query Size
**");
            for (int c = 0; c < 5; c++) {
                switch (c) {
                case 0:
                    f_numLevels = 4;
                    break;
                case 1:
                    f_numLevels = 5;
                    break;
                case 2:
                    f_numLevels = 6;
                    break;
                case 3:
                    f_numLevels = 8;
                    break;
                case 4:
                    f_numLevels = 10;
                    break;
                }
                for (int b = 0; b < 6; b++) {
                    switch (b) {
```

```
                case 0:
                    numNodes = 100;
                    break;
                case 1:
                    numNodes = 500;
                    break;
                case 2:
                    numNodes = 1000;
                    break;
                case 3:
                    numNodes = 2000;
                    break;
                case 4:
                    numNodes = 5000;
                    break;
                case 5:
                    numNodes = 10000;
                    break;
                }
                //nodeCntStd = 0;
                //nodeCntGjb = 0;
                //querySizeStd = 0;
                //querySizeGjb = 0;
                for (int a = 0; a < numIterations; a++) {
                    sgExpStd = new StdSkipGraph(f_numLevels);
                    sgExpGjb = new MultiSkipGraph(f_numLevels);
                    ArrayList<Node> nodesStd = buildNodes(STD, RNDLAYOUT,
numNodes);
                    ArrayList<Node> nodesGjb = new ArrayList<Node>();
                    int ctr = 0;
                    Node n2 = null;
                    for (Node n : nodesStd) {
                        n.setMX();
                        sgExpStd.add(n);
                        nodesGjb.add(ctr, new Node(ctr, new RandomKey(2), f_numLevels,
                                n.getCoords()[0], n.getCoords()[1], 0));
                        n2 = nodesGjb.get(ctr);
                        n2.setMX();
                        sgExpGjb.add(n2);
                        ctr++;
                    }
                    LinkedList<Node> includedNodesStd = new LinkedList<Node>();
                    LinkedList<Node> includedNodesGjb = new LinkedList<Node>();
                    LinkedList preDistStd = new LinkedList();
                    LinkedList postDistStd = new LinkedList();
```

```
                    LinkedList preDistGjb = new LinkedList();
                    LinkedList postDistGjb = new LinkedList();

                    int start = startId.nextInt(nodesStd.size());
                    //if ((a+1) % 25 == 0)
                    //   System.out.println("Iteration " + (a+1) + ", query start = Node " +
start);
                    LinkedList<Node> queryStd =
sgExpStd.rangeQuery(nodesStd.get(start),
                        includedNodesStd, preDistStd, postDistStd, xMin, yMin, xMax,
yMax);
                    LinkedList<Node> queryGjb =
sgExpGjb.rangeQuery(nodesGjb.get(start),
                        includedNodesGjb, preDistGjb, postDistGjb, xMin, yMin, xMax,
yMax);
                    int avgPreDistStd = 0;
                    int avgPostDistStd = 0;
                    int avgPreDistGjb = 0;
                    int avgPostDistGjb = 0;
                    // preDistStd
                    for (Object dist : preDistStd)
                        avgPreDistStd += (Double)dist;
                    if (preDistStd.size() != 0)
                        avgPreDistStd /= preDistStd.size();
                    // postDistStd
                    for (Object dist : postDistStd)
                        avgPostDistStd += (Double)dist;
                    if (postDistStd.size() != 0)
                        avgPostDistStd /= postDistStd.size();
                    // preDistGjb
                    for (Object dist : preDistGjb)
                        avgPreDistGjb += (Double)dist;
                    if (preDistGjb.size() != 0)
                        avgPreDistGjb /= preDistGjb.size();
                    // postDistGjb
                    for (Object dist : postDistGjb)
                        avgPostDistGjb += (Double)dist;
                    if (postDistGjb.size() != 0)
                        avgPostDistGjb /= postDistGjb.size();
                    System.out.print("Std, "+a+", "+f_numLevels+", "+numNodes+",
                                "+includedNodesStd.size()+", "+avgPreDistStd+",
"+avgPostDistStd+", , ");
                    System.out.println("Gjb, "+a+", "+f_numLevels+", "+numNodes+",
                                "+includedNodesGjb.size()+", "+avgPreDistGjb+",
"+avgPostDistGjb+",                                 "+queryGjb.size());
```

```
                    //nodeCntStd += includedNodesStd.size();
                    //nodeCntGjb += includedNodesGjb.size();
                    //querySizeStd += queryStd.size();
                    //querySizeGjb += queryGjb.size();
                }
                System.out.println();
                System.out.println();
                //nodeCntStd /= numIterations;
                //nodeCntGjb /= numIterations;
                //querySizeStd /= numIterations;
                //querySizeGjb /= numIterations;
                //System.out.println("Std (" + numLevels + " levels, " + numNodes + "
total nodes) : " +                    nodeCntStd + " (avg) nodes contacted. " +
querySizeStd + " (avg) nodes returned in                    query.");
                //System.out.println("Gjb (" + numLevels + " levels, " + numNodes + "
total nodes) : " +                    nodeCntGjb + " (avg) nodes contacted. " +
querySizeGjb + " (avg) nodes returned in                    query.");

            }
        }
    }
}

/**
 * Runs the Mobility Simulation
 */
public void runMobilityExperiment() {
    int numNodes = 0;
    int numIterations = 100;
    SkipGraph stdSG = null;
    SkipGraph multiSG = null;
    for (int numLevels = 4; numLevels <= 10; numLevels+=2) {
        for (int numNodeCnt = 0; numNodeCnt < 6; numNodeCnt++) {
            switch (numNodeCnt) {
            case 0:
                numNodes = 100;
                break;
            case 1:
                numNodes = 500;
                break;
            case 2:
                numNodes = 1000;
                break;
            case 3:
                numNodes = 2000;
```

```
                break;
            case 4:
                numNodes = 5000;
                break;
            case 5:
                numNodes = 10000;
                break;
            }
            int numStdMsgs = 0;
            int numMultiMsgs = 0;
            for (int a = 0; a < numIterations; a++) {
                stdSG = new StdSkipGraph(numLevels);
                multiSG = new MultiSkipGraph(numLevels);
                // build symmetric skip graphs
                ArrayList<Node> nodesStd = buildNodes(STD, RNDLAYOUT,
numNodes);
                ArrayList<Node> nodesMulti = new ArrayList<Node>();
                int ctr = 0;
                Node n2 = null;
                for (Node n : nodesStd) {
                    n.setMX();
                    stdSG.add(n);
                    nodesMulti.add(ctr, new Node(ctr, new RandomKey(2), f_numLevels,
                                    n.getCoords()[0], n.getCoords()[1], 0));
                    n2 = nodesMulti.get(ctr);
                    n2.setMX();
                    multiSG.add(n2);
                    ctr++;
                }
                // move the same node the same distance in each graph, keep track of
message counts
                int randomNode = (new Random()).nextInt(numNodes);
                Node movStdNode = nodesStd.get(randomNode);
                Node movMultiNode = nodesMulti.get(randomNode);
                int newStdCoords[] = new int[3];
                int newMultiCoords[] = new int[3];
                int newX = (new Random()).nextInt() & 0xFFFF;
                int newY = (new Random()).nextInt() & 0xFFFF;
                newStdCoords[0] = newX;
                newStdCoords[1] = newY;
                newStdCoords[2] = 0;
                newMultiCoords[0] = newX;
                newMultiCoords[1] = newY;
                newMultiCoords[2] = 0;
                // modify Standard SG coordinates
```

```
                movStdNode.setCoords(newStdCoords);
                movStdNode.setKey();
                numStdMsgs += stdSG.updateNode(movStdNode);
                // modify Multi-Dim SG coordinates
                movMultiNode.setCoords(newMultiCoords);
                movMultiNode.setMX();
                numMultiMsgs += multiSG.updateNode(movMultiNode);
            }

    System.out.println("Levels\t"+numLevels+"\tNodes\t"+numNodes+"\tStdMessages\t"
+

    numStdMsgs/numIterations+"\tMultiMessages\t"+numMultiMsgs/numIterations);
            }
        }


        LinkedList<Node> messages = new LinkedList<Node>();
        LinkedList<Node> route = new LinkedList<Node>();
    }

    /**
     * Moves a node within the skip graph
     */
    public void moveANode() {
        ArrayList<Node> nodes = sg.getNodes();
        int newCoords[] = new int[3];
        newCoords[0] = 64000;
        newCoords[1] = 64000;
        nodes.get(0).setCoords(newCoords);
        nodes.get(0).setKey();
        nodes.get(0).setMX();
        System.out.println("Node " + nodes.get(0).getId() + ": " +
sg.updateNode(nodes.get(0)));
    }

    public LinkedList<HashMap<Integer, ListenableDirectedGraph>> getDirectedGraph()
{
        return sg.getDirectedGraph();
    }
}

//Node Class
package gjb.thesis.model;
import gjb.thesis.strategy.AbstractCreate;
```

```
import gjb.thesis.strategy.KeyMXCreation;

import java.util.Random;

/**
 * Node Class. Nodes belong to a skip graph.
 * @author Chris Augeri
 * @author Greg Brault
 */
public class Node implements KeyMXCreation, Comparable {
    public static final int maxCoord = 65535;      // max 16 bit number
    // strategy
    private AbstractCreate f_strategy;
    // id of this mote
    private int f_id = 0;
    // key
    private Integer f_key = new Integer(0);
    // membership vector
    private int f_mx = 0;
    // used to calculate the membership vectors
    private int[] f_coords = new int[3];

    private int numLevels = 0;

    public Node(int id, AbstractCreate strategy, int numLevels) {
        assert (strategy != null);
        this.numLevels = numLevels;
        f_id = id;
        f_strategy = strategy;
        // set coordinates
        Random random = new Random();
        f_coords[0] = random.nextInt() & 0xFFFF;
        f_coords[1] = random.nextInt() & 0xFFFF;
        f_coords[2] = 0;   // keep it in 2D
        setKey();
    }

    /**
     * Additional constructor to allow for geographical positioning
     * @param id
     * @param strategy
     * @param x x coordinate location
     * @param y y coordinate location
     * @param z z coordinate location
     */
```

```
    public Node(int id, AbstractCreate strategy, int numLevels, int x, int y, int z) {
        assert (strategy != null);
        this.numLevels = numLevels;
        f_id = id;
        f_strategy = strategy;
        // set coordinates
        f_coords[0] = x & 0xFFFF;
        f_coords[1] = y & 0xFFFF;
        f_coords[2] = 0;   // keep it in 2D
        setKey();
    }

    public int getId() {
        return f_id;
    }

    /**
     * Creates a z-ordering from the individual integers of x and y. Assumes that x and y
are 16 bit
     * integers, and returns a 32 bit interleaving of their bits.
     * @param x
     * @param y
     * @return
     */
    public static int zOrder(int x, int y) {
        int mask = 0x8000;              // start with bit 16 of coordinates
        int interleavedBits = 0;                    // initialize m(x)

        // interleave bits of x and y to get our mx
        for (int a = 16; a > 0; a--) {
            interleavedBits |= ((x & mask) << a);       // OR x bit into mx
            interleavedBits |= ((y & mask) << (a-1));  // OR y bit into mx (one to the left of x
bit)
            mask >>>= 1;                            // prepare to mask next bit of x and y
coordinates
        }
        return interleavedBits;
    }

    /**
     * @param zOrder 32 bit zOrder value
     * @return X coordinate from the zOrder
     */
    public static int zOrderX(int zOrder) {
        int mask = 0x80000000;
```

```
    int result = 0;
    for (int a = 16; a > 0; a--) {
        result |= ((zOrder & mask) >>> a);
        mask >>>= 2;      // right shift by 2 (interleaved bits in zOrder)
    }
    return result;
}

/**
 * @param zOrder 32 bit zOrder value
 * @return Y coordinate from the zOrder
 */
public static int zOrderY(int zOrder) {
    int mask = 0x40000000;
    int result = 0;
    for (int a = 15; a >= 0; a--) {
        result |= ((zOrder & mask) >>> a);
        mask >>>= 2;      // right shift by 2 (interleaved bits in zOrder)
    }
    return result;
}
/**
 * Gets the prefix mask bits for two z-ordered numbers (x and y)
 * @param x z-ordered number
 * @param y z-ordered number
 * @return a bit sequence representing which bits are "don't care"
 * a 1 means we care about the bit value at that position
 * a 0 means we don't care about the bit value at that position
 */
public static int getMxPrefixMask(int x, int y) {
    int mxPrefixMask = 0;
    int mask = 0x80000000;

    for (int a = 0; a < 32; a++) {
        if ((x & mask) == (y & mask))
            mxPrefixMask |= mask;
        mask >>>= 1;
    }
    return mxPrefixMask;
}

/**
 * Gets the prefix of two z-ordered numbers. getMxPrefixMask() will tell us which
 * bit locations we should care about whether they are 1 or 0.
 * @param x - z-ordered number
```

```
 * @param y - z-ordered number
 * @return - bit sequence in which identical bit locations between params x and y
 * are set to their respective values, and all other bits will be 0 (the prefixMask
 * will be used to ignore those locations).
 */
public static int getMxPrefix(int x, int y) {
    int mxPrefix = 0;
    int mask = 0x80000000;

    for (int a = 0; a < 32; a++) {
        if ((x & mask) == (y & mask))
            mxPrefix |= (x & mask);
        mask >>>= 1;
    }
    return mxPrefix;
}
/**
 * Used to sort the nodes in an Array (list at a particular level of the SkipGraph)
 * Always sorted on Key.
 * @param arg0
 * @return
 */
public int compareTo(Object arg0) {
    if (this.getClass() != arg0.getClass())
        throw new ClassCastException();
    Node other = (Node)arg0;
    if (this.getKey() < other.getKey()) return -1;
    else if (this.getKey() > other.getKey()) return 1;
    else return 0;
}

/**
 * Compare this node's key with another node. Returns in the same way
 * compareTo returns.
 * @param otherNode
 */
public int compareKey(Node otherNode) {
    if (this.getKey() < otherNode.getKey()) return -1;
    else if (this.getKey() > otherNode.getKey()) return 1;
    else return 0;
}

/**
 * Compare this node's m(x) with another node. Returns in the same way
 * compareTo returns.
```

```
 * @param otherNode
 */
public int compareMX(Node otherNode) {
    if (this.getMX() < otherNode.getMX()) return -1;
    else if (this.getMX() > otherNode.getMX()) return 1;
    else return 0;
}

public int[] getCoords() {
    return f_coords;
}

public void setCoords(int[] coords) {
    f_coords = coords;
}

public int getMX() {
    return f_mx;
}

public int getNumLevels() {
    return numLevels;
}

public String getBinaryMX() {
    StringBuffer mx = new StringBuffer(Integer.toBinaryString(f_mx));
    if (mx.length() < (2*numLevels - 2)) {
        int length = mx.length();
        for (int a = 0; a < ((2*numLevels - 2) - length); a++)
            mx = mx.insert(0, "0");
    }
    return mx.toString();
}

public Integer getKey() {
    return f_key;
}

public void setMX() {
    f_mx = f_strategy.setMX(this);
}

public void setKey() {
    f_key = f_strategy.setKey(this);
}
```

```
    public void setKey(Integer key) {
        f_key = key;
    }
}

//SkipGraph Class
package gjb.thesis.model;

import java.util.Collections;
import java.util.Collection;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.List;
import java.util.LinkedList;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Random;
import java.util.Set;
import java.lang.reflect.Array;

import org._3pq.jgrapht.graph.ListenableDirectedGraph;
import org._3pq.jgrapht.edge.DirectedEdge;

/**
 * Abstract Skip Graph Class. Sub-classed into Standard
 * or Multi-Dimensional Skip graphs.
 *
 * @author Chris Augeri
 * @author Greg Brault
 */
public abstract class SkipGraph implements Collection {
    protected boolean debug = false;
    protected int numLevels = 6;
    protected HashMap<Node, Integer> mxMap;
    protected ListenableDirectedGraph[] skipGraph;
    protected ArrayList<Node>[][] skipGraphKeys;
    protected int numMsgsForLastAdd = 0;

    public int getNumLevels() {
        return numLevels;
    }

    public void setDebug(boolean debug) {
        this.debug = debug;
```

```
}

/**
 * @see java.util.Collection#size()
 */
public int size() {
    if (skipGraphKeys[0][0] == null) {
        return 0;
    } else {
        return skipGraphKeys[0][0].size();
    }
}

/**
 * @see java.util.Collection#isEmpty()
 */
public boolean isEmpty() {
    if (skipGraphKeys[0][0] == null || skipGraphKeys[0][0].size() == 0)
        return true;
    else return false;
}

/**
 * @see java.util.Collection#contains(java.lang.Object)
 */
public boolean contains(Object o) {
    if (skipGraphKeys[0][0] != null && skipGraphKeys[0][0].contains(o)) {
        return true;
    } else return false;
}

/**
 * @see java.util.Collection#add(E)
 */
@SuppressWarnings("unchecked")
public boolean add(Object toAdd) {
    Integer mx;
    numMsgsForLastAdd = 0;
    Node node = (Node)toAdd;
    ArrayList<Node> baseList = skipGraphKeys[0][0];
    LinkedList<Node> route = new LinkedList<Node>();
    LinkedList<Node> messages = new LinkedList<Node>();

    // find a random node to begin insert at
    Node insertAt = null;
```

```
        if (baseList != null)
            insertAt = baseList.get((new Random()).nextInt(baseList.size()));

        // initialize to # of messages it takes to find where we are to insert at base list
        numMsgsForLastAdd = search(insertAt, node.getKey(), route, messages);
        // 1 msg from new node to existing node, and from existing node to other side node.
And from other              // side node to new node
        numMsgsForLastAdd += 3;

        if (!mxMap.containsKey(node)) {
            mx = node.getMX();
            mxMap.put(node, mx);
        } else {
            mx = mxMap.get(node);
        }

        // add ourselves to the appropriate membership vector at each level
        for (int level = 0; level < numLevels; level++) {
            mx = getMXleft(level, node);
            int i = 0;
            if (skipGraphKeys[level][mx] == null) {
                skipGraphKeys[level][mx] = new ArrayList<Node>();
            } else {
                i = Collections.binarySearch((List) skipGraphKeys[level][mx], node);
            }
            if (i < 0 || (i == 0 && skipGraphKeys[level][mx].size() == 0)) {
                skipGraphKeys[level][mx].add(node);
                Collections.sort(skipGraphKeys[level][mx]);
            }
            numMsgsForLastAdd += findBuddies(node, level, mx);
        }
        return true;
    }

    /**
     * @return the number of skip graph messages required for the most
     * recent node addition
     */
    public int getNumMessagesFromAdd() {
        return numMsgsForLastAdd;
    }

    public ArrayList<Node> getNodes() {
        return skipGraphKeys[0][0];
    }
```

```
    public boolean remove(Object toRemove) {
        Node node = (Node)toRemove;
        if (skipGraphKeys[0][0].contains(node)) {
            for (int i = 0; i < numLevels; i++) {
                int mx = getMXleft(i, node);
                skipGraphKeys[i][mx].remove(node);
                if (skipGraphKeys[i][mx].size() == 0)
                    skipGraphKeys[i][mx] = null;
            }
            mxMap.remove(node);
            return true;
        } else return false;
    }

    /**
     * Clears the "view" related variables used to display the skip graph
     */
    @SuppressWarnings("unchecked")
    public void clear() {
        // a Map from our Mote to the (full) membership vector of that Mote
        mxMap = new HashMap<Node, Integer>();
        // list of Motes in a particular membership vector at a particular level
        ArrayList<Node> t = new ArrayList<Node>();
        // http://www.developertutorials.com/tutorials/java/java-theory-generic-
050323/page3.html
        ArrayList<Node>[] tSubList = (ArrayList<Node>[])
Array.newInstance(t.getClass(), 1);
        // http://java.sun.com/docs/books/tutorial/reflect/array/newInstance.html
        skipGraphKeys = (ArrayList<Node>[][]) Array.newInstance(
                tSubList.getClass(), numLevels);
        // add an array of ArrayList<E> at each level
        for (int level = 0; level < numLevels; level++) {
            t = new ArrayList<Node>();
             skipGraphKeys[level] = (ArrayList<Node>[]) Array.newInstance(
                    t.getClass(), (1 << (2*level)));
        }
        clearGraph();
    }

    /**
     * Calculates number of messages needed to find buddies on either side of node
     * @param needsBuddies Node that needs MX buddies
     * @param level Which level we're in
     * @param val needsBuddies's MX value
```

```
     * @return number of messages it took
     */
    protected int findBuddies(Node needsBuddies, int level, Integer val) {
        // only calculate msgs for levels 1 and up
        if (level > 0) {
            int numMessages = 1;   // send buddyOp to nodeLeft on level below
            ArrayList<Node> mxList = skipGraphKeys[level-1][getMXleft(level-1,
needsBuddies)];
            boolean foundBuddy = true;
            int j = mxList.indexOf(needsBuddies);
            j--;

            if (j >= 0) {
                while (j >= 0 && getMXleft(level, needsBuddies) != getMXleft(level,
mxList.get(j))) {
                    j--;
                    numMessages++;     // send buddyOp to next node in list to the left
                }
                numMessages++;              // send reply for buddyOp
            }
            // if we reached the end of the list then check to the right
            if (j < 0) {
                j = mxList.indexOf(needsBuddies);
                j++;
                if (j < mxList.size()) {
                    while (j < mxList.size() && getMXleft(level, needsBuddies) !=
getMXleft(level,                                mxList.get(j))) {
                        j++;
                        numMessages++;     // send buddyOp to next node in list to the left
                    }
                    numMessages++;     // send reply for buddyOp
                    if (j == mxList.size()) foundBuddy = false;
                } else foundBuddy = false;
            }
            // only count messages if we've found other nodes with our membership vector.
Otherwise, we                   // are in our own
            // membership vector and so no messages are sent
            if (foundBuddy)
                numMessages += 3;      // send linkOps to re-link surrounding buddies
            return numMessages;
        } else return 0;
    }

    /**
```

```
     * Abstract moveNode method must be implemented differently for Standard
SkipGraphs
     * and MultiDim SkipGraphs
     *
     * @return number of messages sent to accomplish the update
     */
    public abstract int updateNode(Node toUpdate);

    /**
     * Initialize the graph structure -- could grow one at a time, but makes
     * listening more challenging.
     */
    protected void initGraph() {
        int i;
        skipGraph = new ListenableDirectedGraph[numLevels];
        for (i = 0; i < numLevels; i++) {
            skipGraph[i] = new ListenableDirectedGraph();
        }
    }

    /**
     * Clear all entries in a given graph -- do not want to erase graphs in case
     * someone is listening.
     */
    private void clearGraph() {
        for (int i = 0; i < numLevels; i++) {
            Set v = skipGraph[i].vertexSet();
            if (v != null) {
                skipGraph[i].removeAllVertices(v);
            }
        }
    }

    /**
     * @param l
     *            Skip Graph Level, 0 <= (l)evel < MAX_SKIP_LEVELS
     * @param o
     *            Key, 0 <= (k)ey <= Integer.MAX_VALUE
     * @return Membership vector for a given (l)evel and (k)ey
     */
    protected int getMXleft(int level, Node node) {
        // removed leading left bits
        int mx = mxMap.get(node);
        // remove trailing right bits;
        // L0: mx >>>= 8-0-2 = 6
```

```
        // L1: mx >>>= 8-2-2 = 4
        // L2: mx >>>= 8-4-2 = 2
        // L3: mx >>>= 8-6-2 = 0
        mx >>>= (2*numLevels - (2*level) - 2);
        return mx;
    }

    /**
     * This method returns a LinkedList (representing the different levels of the skip
graph)
     * of LinkedLists (representing the different membership vectors) of
ListenableDirectedGraphs
     * (representing the individual nodes in each m(x)
     * @return
     */
    public LinkedList<HashMap<Integer,ListenableDirectedGraph>> getDirectedGraph()
{
        int level, mx;
        LinkedList<HashMap<Integer, ListenableDirectedGraph>> result =
            new LinkedList<HashMap<Integer, ListenableDirectedGraph>>();
        // not very clean, done as I don't know how to reference the same
        // object in a JGraph
        for (level = 0; level < numLevels; level++) {
            result.add(new HashMap<Integer, ListenableDirectedGraph>());
            for (mx = 0; mx < (1 << (2*level)); mx++) {
                if (skipGraphKeys[level][mx] != null) {
                    result.get(level).put(Integer.valueOf(mx), new
ListenableDirectedGraph());

    result.get(level).get(Integer.valueOf(mx)).addAllVertices(skipGraphKeys[level][mx]);
                    //skipGraph[level].addAllVertices(skipGraphKeys[level][mx]);
                    ListIterator<Node> vl = skipGraphKeys[level][mx].listIterator();
                    Node v1 = vl.next();
                    Node v2;
                    while (vl.hasNext()) {
                        v2 = vl.next();
                        result.get(level).get(mx).addEdge(new DirectedEdge(v1, v2));
                        //skipGraph[level].addEdge(new DirectedEdge(v1, v2));
                        v1 = v2;
                    }
                }
            }
        }
        return result;
    }
```

```
    /**
     * Searches for a node with the closest matching of the search key
     * @return number of messages the search took.
     */
    public int search(Node startNode, Integer key, LinkedList<Node> route,
LinkedList<Node> messages) {
        // check if startNode exists
        if (mxMap.containsKey(startNode)) {
            route.add(startNode);
            if (startNode.getKey() != key)
                return search(startNode, startNode, key, route, messages, numLevels -1);
            else return 0;
        } else return 0;
    }

    /**
     * Recursive internal method that checks to see if it can continue searching in the
current
     * level. If so, it will call itself again with the next-in-line node. If not, it will drop
     * down to the next level and continue with the current node.
     *
     */
    private int search(Node startNode, Node fromNode, Integer key, LinkedList<Node>
route,                          LinkedList<Node> messages, int level) {
        int j;
        ArrayList<Node> list;

        switch (fromNode.getKey().compareTo(key)) {
        case 0: {
            // fromNode.key = key, we found a direct match
            route.add(startNode);
            messages.add(startNode);
            return messages.size();
        }
        case -1: {
            // fromNode.key < key, go right
            list = skipGraphKeys[level][getMXleft(level, fromNode)];
            j = list.indexOf(fromNode);
            j++;
            // traverse in this membership vector until we match, hit the end, or go to far
keywise
            while (j < list.size() && list.get(j).getKey().compareTo(key) == -1) {
                if (messages.size() == 0 || messages.getLast() != list.get(j))
                    messages.add(list.get(j));
```

```
            route.add(list.get(j));
            j++;
        }
        if (j == list.size()) {
            // end of list. Go to lower level if we're not already at level 0
            if (level > 0)
                return search(startNode, list.get(j-1), key, route, messages, level-1);
            else return messages.size();
        } else {
            // include the message to the node that either had the key, or key out of range
            if (messages.size() == 0 || messages.getLast() != list.get(j))
                messages.add(list.get(j));
            // we've found a node that has that key
            if (list.get(j).getKey().compareTo(key) == 0) {
                // add the node to the route, and final message to start node
                route.add(list.get(j));
                messages.add(startNode);
                return messages.size();
            } else {
                // key is out of range. Go to lower level if we're not already at level 0
                if (level > 0)
                    return search(startNode, list.get(j-1), key, route, messages, level-1);
                else return messages.size();
            }
        }
    }
    case 1: {
        // fromNode.key > key, go left
        list = skipGraphKeys[level][getMXleft(level, fromNode)];
        j = list.indexOf(fromNode);
        j--;
        // traverse in this membership vector until we match, hit the beginning, or go to
far keywise
        while (j >= 0 && list.get(j).getKey().compareTo(key) == 1) {
            if (messages.size() == 0 || messages.getLast() != list.get(j))
                messages.add(list.get(j));
            route.add(list.get(j));
            j--;
        }
        if (j < 0) {
            // beginning of list. Go to lower level if we're not already at level 0
            if (level > 0) {
                return search(startNode, list.get(0), key, route, messages, level-1);
            } else {
                return messages.size();
```

```
                }
            } else {
                // include the message to the node that either had the key, or key out of range
                if (messages.size() == 0 || messages.getLast() != list.get(j))
                    messages.add(list.get(j));
                // we've found a node that has that key
                if (list.get(j).getKey().compareTo(key) == 0) {
                    // add the node to the route, and final message to start node
                    route.add(list.get(j));
                    messages.add(startNode);
                    return messages.size();
                } else {
                    // key is out of range. Go to lower level if we're not already at level 0
                    if (level > 0)
                        return search(startNode, list.get(0), key, route, messages, level-1);
                    else return messages.size();
                }
            }
        }
        default: return 0;  // will never get here. Switch is on a compareTo() => -1, 0, 1
        }
    }

    /**
     * Executes a range query in the skip graph
     * @param minKey - Inclusive minimum key value to look for
     * @param maxKey - Inclusive maximum key value to look for
     * @param startKey - node Id to start with
     * @return
     */
    abstract public LinkedList<Node> rangeQuery(Node startNode, LinkedList<Node>
includedNodes,          LinkedList preDist, LinkedList postDist, int xMin, int yMin, int
xMax, int yMax);

    /**
     * @see java.util.Collection#iterator()
     */
    public Iterator<Node> iterator() {
        return skipGraphKeys[0][0].iterator();
    }

    /**
     * @see java.util.Collection#toArray()
     */
    public Object[] toArray() {
```

```
        return skipGraphKeys[0][0].toArray();
    }

    /**
     * @see java.util.Collection#containsAll(java.util.Collection)
     */
    public boolean containsAll(Collection c) {
        boolean containsAll = true;
        for (Object o : c) {
            if (!skipGraphKeys[0][0].contains(o)) {
                containsAll = false;
                break;
            }
        }
        return containsAll;
    }

    /**
     * @see java.util.Collection#addAll(java.util.Collection)
     */
    public boolean addAll(Collection c) {
        return false;
    }

    /**
     * @see java.util.Collection#removeAll(java.util.Collection)
     */
    public boolean removeAll(Collection c) {
        return false;
    }

    /**
     * @see java.util.Collection#retainAll(java.util.Collection)
     */
    public boolean retainAll(Collection c) {
        return false;
    }


    @SuppressWarnings("unchecked")
    public Object[] toArray(Object[] arg0) {
        return null;
    }
}
```

```
//StandardSkipGraph Class
package gjb.thesis.model;

import java.util.ArrayList;
import java.util.LinkedList;

/**
 * Standard sub-class of the Abstract Skip Graph Class
 * @author Greg Brault
 */
public class StdSkipGraph extends SkipGraph {
    private boolean debug = false;

    public StdSkipGraph(int numLevels) {
        super();
        this.numLevels = numLevels;
        initGraph();
        clear();
    }

    public void setDebug(boolean debug) {
        this.debug = debug;
    }

    /**
     * Executes a range query in the skip graph that is configured in the standard
     * way (Deterministic z-ordered keys and random membership vectors)
     * @param minKey - Inclusive minimum key value to look for
     * @param maxKey - Inclusive maximum key value to look for
     * @param startKey - node Id to start with
     * @return
     */
    @Override
    public LinkedList<Node> rangeQuery(Node startNode, LinkedList<Node>
includedNodes, LinkedList             preDist, LinkedList postDist, int xMin, int yMin, int
xMax, int yMax) {
        int zOrderMin = Node.zOrder(xMin, yMin);
        int zOrderMax = Node.zOrder(xMax, yMax);
        LinkedList<Node> results = new LinkedList<Node>();
        if (mxMap.containsKey(startNode)) {
            if (debug) System.out.println("Starting Std query at node " + startNode.getId() +
" at L" +                         (numLevels-1));
            includedNodes.add(startNode);
            nextNodeRangeQuery(startNode, null, zOrderMin, zOrderMax, results,
includedNodes, preDist,                     postDist, numLevels -1);
```

```
            return results;
        } else return results;
    }

    @SuppressWarnings("unchecked")
    private LinkedList<Node> nextNodeRangeQuery(Node root, Node from, int
zOrderMin, int zOrderMax, LinkedList<Node> results,
LinkedList<Node> includedNodes, LinkedList preDist, LinkedList postDist, int
level) {
        int j;
        if ((root.getKey() >= zOrderMin) && (root.getKey() <= zOrderMax)) {
            // In range. Drop down to lowest level and collect all nodes that are in range
            // add ourselves
            results.add(root);
            ArrayList<Node> baseList = skipGraphKeys[0][0];
            if (debug) System.out.println("In range. Adding Node " + root.getId() + " at
L0.");
            // go Right
            j = skipGraphKeys[0][0].indexOf(root);
            j++;
            while (j < skipGraphKeys[0][0].size() &&
skipGraphKeys[0][0].get(j).getKey()<=zOrderMax){

                if (debug) {
                    System.out.println("In range. Contacting Node " +
skipGraphKeys[0][0].get(j).getId() + " at L0.");
                    System.out.println("In range. Adding Node " +
skipGraphKeys[0][0].get(j).getId() + " at L0.");
                }
                results.add(skipGraphKeys[0][0].get(j));
                includedNodes.add(skipGraphKeys[0][0].get(j));
                postDist.add(Math.sqrt(Math.pow(baseList.get(j).getCoords()[0] -
baseList.get(j-1).getCoords()[0], 2) +
Math.pow(baseList.get(j).getCoords()[1] - baseList.get(j-
    1).getCoords()[1], 2)));
                j++;
            }
            // add border contact if we're not at the end of the list
            if (j < skipGraphKeys[0][0].size() && skipGraphKeys[0][0].get(j) != from) {
                if (debug) {
                    System.out.println("Out of range. Contacting Node " +
                            skipGraphKeys[0][0].get(j).getId() + " at L0.");
                }
                includedNodes.add(skipGraphKeys[0][0].get(j));
```

```
                postDist.add(Math.sqrt(Math.pow(baseList.get(j).getCoords()[0] -
baseList.get(j-                                1).getCoords()[0], 2) +
Math.pow(baseList.get(j).getCoords()[1] - baseList.get(j-                                2)));
            }
            // go Left
            j = skipGraphKeys[0][0].indexOf(root);
            j--;
            while (j >= 0 && skipGraphKeys[0][0].get(j).getKey() >= zOrderMin) {
                if (debug) {
                    System.out.println("In range. Contacting Node " +
skipGraphKeys[0][0].get(j).getId() + "                          at L0.");
                    System.out.println("In range. Adding Node " +
skipGraphKeys[0][0].get(j).getId() + " at                          L0.");
                }
                results.add(skipGraphKeys[0][0].get(j));
                includedNodes.add(skipGraphKeys[0][0].get(j));
                postDist.add(Math.sqrt(Math.pow(baseList.get(j).getCoords()[0] -
                                baseList.get(j+1).getCoords()[0], 2) +
Math.pow(baseList.get(j).getCoords()[1] -
    baseList.get(j+1).getCoords()[1], 2)));
                j--;
            }
            // add border contact if we're not at the beginning of the list
            if (j >= 0 && skipGraphKeys[0][0].get(j) != from) {
                if (debug) {
                    System.out.println("Out of range. Contacting Node " +
                                skipGraphKeys[0][0].get(j).getId() + " at L0.");
                }
                includedNodes.add(skipGraphKeys[0][0].get(j));
                postDist.add(Math.sqrt(Math.pow(baseList.get(j).getCoords()[0] -
                                baseList.get(j+1).getCoords()[0], 2) +
Math.pow(baseList.get(j).getCoords()[1] -
    baseList.get(j+1).getCoords()[1], 2)));
            }
            return results;
        } else if (root.getKey() < zOrderMin) {
            // go right
            ArrayList<Node> list = skipGraphKeys[level][getMXleft(level, root)];
            j = list.indexOf(root);
            j++;
            while (j < list.size() && list.get(j).getKey() < zOrderMin) {
                if (debug) System.out.println("Contacting Node " + list.get(j).getId() + " on
L" + level);
                includedNodes.add(list.get(j));
```

```
                preDist.add(Math.sqrt(Math.pow(list.get(j).getCoords()[0] - list.get(j-
1).getCoords()[0], 2) +
                        Math.pow(list.get(j).getCoords()[1] - list.get(j-1).getCoords()[1], 2)));
                j++;
            }
            if (j == list.size())
                if (level > 0)
                    if (j == 1)
                        return nextNodeRangeQuery(list.get(j-1), from, zOrderMin,
zOrderMax, results,                                 includedNodes, preDist, postDist, level - 1);
                    else return nextNodeRangeQuery(list.get(j-1), list.get(j-2), zOrderMin,
zOrderMax,                                  results, includedNodes, preDist, postDist, level -
1);
                else return results;
            else {
                if (debug) System.out.println("Contacting Node " + list.get(j).getId() + " on
L" + level);
                includedNodes.add(list.get(j));
                preDist.add(Math.sqrt(Math.pow(list.get(j).getCoords()[0] - list.get(j-
1).getCoords()[0], 2) +
                        Math.pow(list.get(j).getCoords()[1] - list.get(j-1).getCoords()[1], 2)));
                if (list.get(j).getKey() <= zOrderMax)
                    return nextNodeRangeQuery(list.get(j), list.get(j-1), zOrderMin,
zOrderMax, results,                                 includedNodes, preDist, postDist, 0);
                else if (level == 0)
                    return results;
                else if (j == 1)
                    return nextNodeRangeQuery(list.get(j-1), from, zOrderMin, zOrderMax,
results,                                    includedNodes, preDist, postDist, level - 1);
                else return nextNodeRangeQuery(list.get(j-1), list.get(j-2), zOrderMin,
zOrderMax, results,                                 includedNodes, preDist, postDist, level - 1);
            }
        } else {
            // go left
            ArrayList<Node> list = skipGraphKeys[level][getMXleft(level, root)];
            j = list.indexOf(root);
            j--;
            while (j >= 0 && list.get(j).getKey() > zOrderMax) {
                if (debug) System.out.println("Contacting Node " + list.get(j).getId() + " at L"
+ level);
                includedNodes.add(list.get(j));
                preDist.add(Math.sqrt(Math.pow(list.get(j).getCoords()[0] -
list.get(j+1).getCoords()[0], 2) +
                        Math.pow(list.get(j).getCoords()[1] - list.get(j+1).getCoords()[1], 2)));
                j--;
```

```
            }
            if (j < 0)
                if (level > 0)
                    if (list.size() == 1)
                        return nextNodeRangeQuery(list.get(0), from, zOrderMin, zOrderMax,
results,                                includedNodes, preDist, postDist, level - 1);
                        else return nextNodeRangeQuery(list.get(0), list.get(1), zOrderMin,
zOrderMax,                                    results, includedNodes, preDist, postDist,level -
1);
                else return results;
            else {
                if (debug) System.out.println("Contacting Node " + list.get(j).getId() + " at L"
+ level);
                includedNodes.add(list.get(j));
                preDist.add(Math.sqrt(Math.pow(list.get(j).getCoords()[0] -
list.get(j+1).getCoords()[0], 2) +
                        Math.pow(list.get(j).getCoords()[1] - list.get(j+1).getCoords()[1], 2)));
                if (list.get(j).getKey() >= zOrderMin)
                    return nextNodeRangeQuery(list.get(j), list.get(j+1), zOrderMin,
zOrderMax, results,                            includedNodes, preDist, postDist, 0);
                else if (level == 0)
                    return results;
                else if (j == (list.size() - 2))
                    return nextNodeRangeQuery(list.get(j+1), from, zOrderMin, zOrderMax,
results,                                includedNodes, preDist, postDist, level - 1);
                else return nextNodeRangeQuery(list.get(j+1), list.get(j+2), zOrderMin,
zOrderMax, results,                    includedNodes, preDist, postDist, level - 1);
            }
        }
    }

    /**
     * To calculate total number of messages for a node move in a Standard SkipGraph
with
     * deterministic keys based on z-order of geo coordinates and random membership
vectors,
     * we must include:
     * 1) Link re-assignment messages due to the now-absent node
     * 2) Messages incurred by re-adding the node at each level. This requires the
     * standard query-method to be used, since a new key has been generated based
     * on the z-order of its new geo coordinates
     */
    @Override
    public int updateNode(Node toUpdate) {
        int messages = 0;
```

```
        ArrayList<Node> list = null;
        int index = 0;
        // reassign links for the surrounding nodes in each list at each level
        // assume old link will fail when trying to communicate and no response
        /*
        for (int i = 0; i < numLevels; i++) {
            list = skipGraphKeys[i][getMXleft(i, toUpdate)];
            index = list.indexOf(toUpdate);
            if ((index-1) >=0)
                messages += 2;    // BuddyRequest + BuddyAck
            if (index+1 < list.size())
                messages += 2;    // BuddyRequest + BuddyAck

        }*/
        // remove the node
        remove(toUpdate);
        // re-add the node based on nodes new key and geo-position
        // For a Standard SkipGraph, with deterministic keys based on geo
        // position and random MXs, a full add needs to take place
        add(toUpdate);
        messages += numMsgsForLastAdd;
        return messages;
    }
}

//MultiDimensionalSkipGraph Class
package gjb.thesis.model;

import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

/**
 * Multi-Dimensional sub-class of the Abstract Skip Graph Class
 * @author Greg Brault
 */
public class MultiSkipGraph extends SkipGraph {

    public MultiSkipGraph(int numLevels) {
        super();
        this.numLevels = numLevels;
        initGraph();
        clear();
    }
```

```
    /**
     * Executes a range query in the skip graph that is configured in the proposed
     * way (Random keys and Deterministic z-ordered membership vectors)
     * @param minKey - Inclusive minimum key value to look for (z-ordered)
     * @param maxKey - Inclusive maximum key value to look for (z-ordered)
     * @param startKey - node Id to start with
     * @return
     */
    public LinkedList<Node> rangeQuery(Node startNode, LinkedList<Node>
includedNodes, LinkedList preDist, LinkedList postDist, int xMin, int yMin, int xMax,
int yMax) {
        int zOrderMin = Node.zOrder(xMin, yMin);
        int zOrderMax = Node.zOrder(xMax, yMax);
        int mxPfxMask = (Node.getMxPrefixMask(zOrderMin, zOrderMax) >>> (32 -
2*numLevels + 2));
        int mxPfx = (Node.getMxPrefix(zOrderMin, zOrderMax) >>> (32 - 2*numLevels
+ 2));
        LinkedList<Node> results = new LinkedList<Node>();
        //System.out.println("z-order min: " + Integer.toBinaryString(zOrderMin) +
        //       ", z-order max: " + Integer.toBinaryString(zOrderMax));
        //System.out.println("mxPrefixMask: " + Integer.toBinaryString(mxPfxMask) + ",
mxPfx: " +                  Integer.toBinaryString(mxPfx));
        if (debug)
            System.out.println("Starting Gjb query at node " + startNode.getId() + " on
L0.");
        includedNodes.add(startNode);
        recurseRangeQuery(startNode, results, includedNodes, preDist, postDist,
zOrderMin, zOrderMax,                mxPfx, mxPfxMask, 0);
        return results;
    }

    /**
     * Recursive routine
     * @param node - current node we're at
     * @param results - common data list holding results
     * @param pfx - mx prefix of the current node at the current level "level"
     * @param pfxMask - bitmask indicating which bits are involved in matching.
     * @param level - our current level
     *
     * Returns when all paths for all levels have been explored
     */
    @SuppressWarnings("unchecked")
```

```
    private void recurseRangeQuery(Node node, LinkedList<Node> results,
LinkedList<Node>                    includedNodes, LinkedList preDist, LinkedList
postDist, int zOrderMin, int zOrderMax, int pfx, int              pfxMask, int level) {
        boolean found11 = false;
        int j;
        // this is the prefix that we are comparing to at this particular level
        int thisPfx = (pfx >>> ((2*numLevels) - (2*(level+1)) - 2));
        // this is the prefix mask that we are using at this particular level
        int thisPfxMask = (pfxMask >>> ((2*numLevels) - (2*(level+1)) - 2));
        int nodeMx = getMXleft(level, node);
        ArrayList<Node> list = skipGraphKeys[level][nodeMx];

        // Stop Condition #1 : Highest level?
        if (level + 1 == numLevels) {
            // and full prefix match based on prefix mask?
            if ((nodeMx & pfxMask) == (pfx & pfxMask)) {
                // we have to traverse the entire list and check whether the nodes are inside
the query range!
                // possibly include ourself
                int nodeZOrder = Node.zOrder(node.getCoords()[0], node.getCoords()[1]);
                if (nodeZOrder >= zOrderMin && nodeZOrder <= zOrderMax) {
                    if (debug)
                        System.out.println("Adding Node " + node.getId() + " at maxLevel L"
+ level);
                    results.add(node);
                }
                // go right, possibly including the nodes we see
                j = list.indexOf(node);
                j++;
                while (j < list.size()) {
                    nodeZOrder = Node.zOrder(list.get(j).getCoords()[0],
list.get(j).getCoords()[1]);
                    if (debug)
                        System.out.println("Contacting Node " + list.get(j).getId() + " heading
right at                                maxLevel L" + level);
                    if (nodeZOrder >= zOrderMin && nodeZOrder <= zOrderMax) {
                        if (debug)
                            System.out.println("Adding Node " + list.get(j).getId() + " at
maxLevel L" +                                    level);
                        results.add(list.get(j));
                    }
                    includedNodes.add(list.get(j));
                    postDist.add(Math.sqrt(Math.pow(list.get(j).getCoords()[0] - list.get(j-
1).getCoords()[0],                    2) + Math.pow(list.get(j).getCoords()[1] -
list.get(j-1).getCoords()[1], 2)));
```

```
                j++;
            }
            // go left
            j = list.indexOf(node);
            j--;
            while (j >= 0) {
                nodeZOrder = Node.zOrder(list.get(j).getCoords()[0],
list.get(j).getCoords()[1]);
                if (debug)
                    System.out.println("Contacting Node " + list.get(j).getId() + " heading
left at                                 maxLevel L" + level);
                if (nodeZOrder >= zOrderMin && nodeZOrder <= zOrderMax) {
                    if (debug)
                        System.out.println("Adding Node " + list.get(j).getId() + " at
maxLevel L" +                                   level);
                    results.add(list.get(j));
                }
                includedNodes.add(list.get(j));
                postDist.add(Math.sqrt(Math.pow(list.get(j).getCoords()[0] -
list.get(j+1).getCoords()[0],                   2) + Math.pow(list.get(j).getCoords()[1] -
list.get(j+1).getCoords()[1], 2)));
                j--;
            }
            return;
        } else return;      // we're at the highest level and no match, return
    }
    // not at the highest level yet, switch between the two bits being scrutinized
    switch (thisPfxMask & 0x03) {
    case 3:     // exact match for pfx
        // check if we are a match
        if (getMXleft(level + 1, node) == thisPfx) {
            if (debug)
                System.out.println("Found Node " + node.getId() + " immediately at L" +
level);
            recurseRangeQuery(node, results, includedNodes, preDist, postDist,
zOrderMin, zOrderMax,               pfx, pfxMask, level + 1);
        } else {
            found11 = false;
            j = list.indexOf(node);
            // go right, checking for match
            j++;
            while ((j < list.size()) && !found11) {
                if (debug)
                    System.out.println("Contacting Node " + list.get(j).getId() + " heading
right at L" +                       level);
```

```
                includedNodes.add(list.get(j));
                preDist.add(Math.sqrt(Math.pow(list.get(j).getCoords()[0] - list.get(j-
1).getCoords()[0],                    2) + Math.pow(list.get(j).getCoords()[1] -
list.get(j-1).getCoords()[1], 2)));
                if (getMXleft(level + 1, list.get(j)) == thisPfx) {
                    found11 = true;
                    //System.out.println("11: Found rightward, node " + list.get(j).getId() +
", level " +                    level);
                    recurseRangeQuery(list.get(j), results, includedNodes, preDist,
postDist, zOrderMin,                    zOrderMax,pfx, pfxMask, level + 1);
                } else j++;
            }
            j = list.indexOf(node);
            j--;
            // go left, checking for match
            while ((j >= 0) && !found11) {
                if (debug)
                    System.out.println("Contacting Node " + list.get(j).getId() + " heading
left at L" +                    level);
                includedNodes.add(list.get(j));
                preDist.add(Math.sqrt(Math.pow(list.get(j).getCoords()[0] -
list.get(j+1).getCoords()[0],                    2) + Math.pow(list.get(j).getCoords()[1] -
list.get(j+1).getCoords()[1], 2)));
                if (getMXleft(level + 1, list.get(j)) == thisPfx) {
                    found11 = true;
                    //System.out.println("11: Found leftward, node " + list.get(j).getId() +
", level " +                    level);
                    recurseRangeQuery(list.get(j), results, includedNodes, preDist,
postDist, zOrderMin,                    zOrderMax, pfx, pfxMask, level + 1);
                } else j--;
            }
        }
        break;
    case 2:     // 10. exact match for left bit of pfx, right bit don't care
    case 1:
    case 0:
        // 00. Both left and right bits don't care. For now, return all nodes in this sub
quadrant
        // include ourself
        if (debug)
            System.out.println("Adding Node " + node.getId() + " in X don't care list X at
L" + level);
        results.add(node);
        // go right
        j = list.indexOf(node);
```

```
            j++;
            while (j < list.size()) {
                if (debug) {
                    System.out.println("Contacting Node " + list.get(j).getId() + " heading
rightward at X                    don't care list X at L" + level);
                    System.out.println("Adding Node " + list.get(j).getId() + " in X don't care
list X at L" +                    level);
                }
                results.add(list.get(j));
                includedNodes.add(list.get(j));
                postDist.add(Math.sqrt(Math.pow(list.get(j).getCoords()[0] - list.get(j-
1).getCoords()[0], 2)                    + Math.pow(list.get(j).getCoords()[1] - list.get(j-
1).getCoords()[1], 2)));
                j++;
            }
            // go left
            j = list.indexOf(node);
            j--;
            while (j >= 0) {
                if (debug) {
                    System.out.println("Contacting Node " + list.get(j).getId() + " heading
rightward at X                    don't care list X at L" + level);
                    System.out.println("Adding Node " + list.get(j).getId() + " in X don't care
list X at L" +                    level);
                }
                results.add(list.get(j));
                includedNodes.add(list.get(j));
                postDist.add(Math.sqrt(Math.pow(list.get(j).getCoords()[0] -
list.get(j+1).getCoords()[0], 2)                    + Math.pow(list.get(j).getCoords()[1] -
list.get(j+1).getCoords()[1], 2)));
                j--;
            }
        }
    }

    /**
     * To calculate total number of messages for a node move in a Multi-Dim SkipGraph
     * with a random key and deterministic membership vector based on z-order of the
     * geo coordinates, we must include:
     * 1) Position in base level will not change because the key is the same, it's only
     * the coordinates that have changed. Therefore, we do not need to do the standard
     * query from the normal add process
     * 2) The full membership vector has changed because our geo coordinates have
changed.
     * Therefore at each level we must change membership vector lists.
```

```
     */
    @SuppressWarnings("unchecked")
    @Override
    public int updateNode(Node toUpdate) {
        // This code borrowed from the "add(Node)" method, but without the initial base
        // level query since we already know where we are in that list.
        // Redo the mx mapping
        mxMap.remove(toUpdate);
        mxMap.put(toUpdate, toUpdate.getMX());
        int mx;
        int messages = 0;
        // add ourselves to the appropriate membership vector at each level, starting
        // with level 1
        for (int level = 1; level < numLevels; level++) {
            mx = getMXleft(level, toUpdate);
            int i = 0;
            if (skipGraphKeys[level][mx] == null) {
                skipGraphKeys[level][mx] = new ArrayList<Node>();
            } else {
                i = Collections.binarySearch((List) skipGraphKeys[level][mx], toUpdate);
            }
            if (i < 0 || (i == 0 && skipGraphKeys[level][mx].size() == 0)) {
                skipGraphKeys[level][mx].add(toUpdate);
                Collections.sort(skipGraphKeys[level][mx]);
            }
            messages += findBuddies(toUpdate, level, mx);
        }
        return messages;
    }
}

//AbstractCreate Class
package gjb.thesis.strategy;

import java.util.Random;

import gjb.thesis.model.Node;

/**
 * Abstract Node creation class. Sub-classed into
 * RandomKey or RandomMX.
 * @author Greg Brault
 */
public abstract class AbstractCreate {
    static Random f_random = null;
```

```
    abstract public int setMX(Node mote);
    abstract public Integer setKey(Node mote);
}

//KeyMXCreation Class
package gjb.thesis.strategy;

/**
 * The interface used for the sub-classes of AbstractCreate
 * (RandomKey and RandomMX)
 * @author Greg Brault
 */
public interface KeyMXCreation {
    public int getMX();
    public Integer getKey();
    public void setMX();
    public void setKey();
}

//RandomKey Class
package gjb.thesis.strategy;

import java.util.Random;

import gjb.thesis.model.Node;

/**
 * Configures the node for Random Key : Deterministic Membership Vector.
 * The idea is to set the membership vector according to a z-ordering of
 * the coordinates of the node (x,y, and/or z), and then choose the node's
 * key randomly.
 * @author Greg Brault
 */
public class RandomKey extends AbstractCreate {
    public RandomKey(int seed) {
        if (f_random == null)
            //f_random = new Random(seed);
            f_random = new Random();
    }

    @Override
    public int setMX(Node mote) {
        int x = mote.getCoords()[0];   // x coordinate
        int y = mote.getCoords()[1];   // y coordinate
        //int z = mote.getCoords()[2]; // z coordinate
```

```
        int mx = Node.zOrder(x,y);
        mx >>>= (32 - 2*mote.getNumLevels() + 2);
        return mx;
    }

    @Override
    public Integer setKey(Node node) {
        return new Integer(f_random.nextInt());
        //int[] coords = node.getCoords();
        // key is the x-value of the coordinate (x,y,z)
        //return new Integer(coords[0]);
    }
}

//RandomMX Class
package gjb.thesis.strategy;

import java.util.Random;
import gjb.thesis.model.Node;

/**
 * Configures the node for Random Key : Deterministic Membership Vector.
 * The idea is to set the membership vector according to a z-ordering of
 * the coordinates of the node (x,y, and/or z), and then choose the node's
 * key randomly.
 * @author Greg Brault
 */
public class RandomMX extends AbstractCreate {
    public RandomMX(int seed) {
        if (f_random == null)
            //f_random = new Random(seed);
            f_random = new Random();
    }
    @Override
    public int setMX(Node mote) {
        int mx = f_random.nextInt();
        mx >>>= (32 - 2*mote.getNumLevels() + 2);
        return mx;
    }

    @Override
    public Integer setKey(Node node) {
        int x = node.getCoords()[0];   // x coordinate
        int y = node.getCoords()[1];   // y coordinate
        //int z = mote.getCoords()[2]; // z coordinate
```

```
        int key = Node.zOrder(x,y);        // initialize m(x)
        //key >>>= (32 - 2*SkipGraph.MAX_SKIP_LEVELS + 2);
        return new Integer(key);
    }
}
```

Bibliography

[AsS03] Aspnes, James., Shah, Gauri. "Skip Graphs". *Proceedings of the 14th ACM/SIAM Symposium on Discrete Algorithms*, 384-393. 2003.

[Aug06] Augeri, Christopher. "Using Spectral Analysis and Trie-Based Skip Graphs to Enable Multi-Dimensional Queries In Mobile 3-D Wireless Networks". Air Force Institute of Technology, 2007.

[AuM05] Augeri, Christopher., Mullins, Barry. "Data-Centric Indexing and Routing". Air Force Institute of Technology, 2005.

[Ecl07] Eclipse IDE. URL http://www.eclipse.org/

[EGS04] Eppstein, David., Goodrich, Michael T., Sun, Jonathan Z. "The Skip Quadtree: A Simple Dynamic Data Structure for Multidimensional Data". 2004.

[FiB74] Finkel, R. A., Bentley, J.L. "Quad trees: a data structure for retrieval on composite keys". *Acta Informatica19*, 4(1):1-9, 1974

[Fre04] Freeman, Eric., Freeman, Elisabeth., Sierra, K., and Bates, B. Head First Design Patterns. Sebastopol, CA: O'Reilly, 2004.

[Jav07] Java Programming Language. URL http://java.sun.com

[HJS04] Harvey, Nicholas J.A., Jones, Michael B., Saroiu, Stefan., Theimer, Marvin., Wolman, Alec. "SkipNet: A Scalable Overlay Network with Practical Locality Properties". 2004

[Haw90] Hawkins, T. *Dictionary of Scientific Biography*. New York, 1990.

[KaW05] Karl, Holger., Willig, Andreas. "Protocols and Architectures for Wireless Sensor Networks". West Sussex, England: Wiley, 2005.

[LEG05] Arge, Lars., Eppstein, David., Goodrich, Michael T. "Skip-webs: efficient distributed data structures for multi-dimensional data sets". *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 69-76. ACM Press, New York, NY, 2005.

[Opn07] Opnet Simulation Software. URL http://www.opnet.com

[Ore82] Orenstein, J.A. "Multidimensional tries used for associative searching". *Inform. Process. Lett.*, 13:150-157, 1982.

[PBD03] Perkins, C., Belding-Royer, E., Das, S. “Ad hoc On-Demand Distance Vector (AODV) Routing”, July 2003. URL http://www.ietf.org/rfc/rfc3561.txt.

[PeB94] Perkins, Charles E., Bhagwat, Pravin. “Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers”. *ACM SIGCOMM’94 Conference on Communications Architectures, Protocols and Applications*, 1994

[Pug90] Pugh, William. “Skip Lists: a probabilistic alternative to balanced trees”. *Communications of the ACM*, 33(6):668-676, 1990. ISSN 0001-0782.

[PWW02] Ptak, Stephen., Webster, Charles R. Jr., Wilson, Tony W. “Effective Decision-Making Processes for the Joint Force Commander. *Air Land and Sea Bulletin.*

[RoM04] Romer, Kay., Mattern, Friedemann. “The Design Space of Wireless Sensor Networks. *IEEE Wireless Communications*, Dec 2004.

[Sag94] Sagan, Hans. *Space-Filling Curves*. Springer-Verlag, New York, NY, 1994.

[Sam06] Samet, Hanan. Foundations of Multidimensional and Metric Data Structures. San Francisco: Morgan Kaufmann Publishers, 2006.

[UML05] Unified Modeling Language: Superstructure, version 2.0. URL http://www.omg.org/docs/formal/05-07-04.pdf

[ZhG04] Zhao, Feng., Guibas, Leonidas. Wireless Sensor Networks: An Information Processing Approach. San Francisco, CA: Morgan Kaufmann Publishers, 2004.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
22-03-2007	Master's Thesis	March 2006 – March 2007

4. TITLE AND SUBTITLE

MULTI-DIMENSIONAL RANGE QUERYING USING A MODIFICATION OF THE SKIP GRAPH

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Brault, Gregory J., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GE/ENG/07-04

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Scott Gardner, P.E.
Air Force Communications Agency/Dynamic Network Analysis Division
Scott AFB, IL 62225
Scott.gardner.2@scott.af.mil, DSN 779-6794

10. SPONSOR/MONITOR'S ACRONYM(S)
AFCA/ENAN

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Skip graphs are an application layer-based distributed routing data structure that can be used in a sensor network to facilitate user queries of data collected by the sensor nodes. This research investigates the impact of a proposed modification to the skip graph proposed by Aspnes and Shah.

Nodes contained in a standard skip graph are sorted by their key value into successively smaller groups based on random membership vectors computed locally at each node. The proposed modification inverts the node key and membership vector roles, where group membership is computed deterministically and node keys are computed randomly.

Both skip graph types are modeled in Java. Range query and node mobility simulations are performed. The number of skip graph levels, total node count, and query precision are varied for query simulations; number of levels and total node count is varied for the mobility simulation. Query performance is measured by the number of skip graph messages used to execute the query while mobility performance is measured by the number of messages transmitted to maintain skip graph coherence.

When the number of levels is limited and query precision is low, or when query precision is matched by the number of levels in the skip graph and total network node counts are increased, the modified skip graph transmits fewer messages to execute the query. Furthermore, fewer update messages are needed to fix lost node references due to mobile nodes.

15. SUBJECT TERMS

distributed sensor network (DSN), skip list, skip graph, multi-dimensional, range query, distributed query, space-filling curve.

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Barry E. Mullins, Ph.D. (ENG)
U	**U**	**U**	**UU**	136	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-3636, ext 7979 (Barry.Mullins@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18