MOVEMENT OF DATA BETWEEN TWO LINEAR REFERENCING SYSTEMS OF DIFFERENT RESOLUTION

by

SUFAL KUMAR BISWAS

ANDREW J. GRAETTINGER, COMMITTEE CHAIR
S. ROCKY DURRANS
RANDY K. SMITH

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Civil, Construction, and Environmental Engineering
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2014

ABSTRACT

Two independent liner referencing systems are being used simultaneously by Wisconsin Department of Transportation (WisDOT) for business data management and analysis in transportation sector. The State Trunk Network (STN) representing only state route is used for analysis and planning purposes. Wisconsin Information for Local Roads (WISLR) representing all roads (State and local roads) of Wisconsin and having comparatively lower resolution than STN is used in Incident Location Tools for storing crash data. A functional merge link_link table was developed to define relationship between these two systems.

The link_link table updating is required yearly not only to reflect the changes in line work and attributes of STN and WISLR but also to make updated crash map and to translate data between STN to WISLR. Data translation from STN to WISLR is acceptably well while data translation in reverse direction is ambiguous where the resolution is dissimilar between STN and WISLR. To control these ambiguities some rules and guidelines were proposed. In this research those rules and guidelines were implemented to control all ambiguities. Research was also done to use link_link table to provide name to the ramps dynamically.

The thesis consists of link_link table update methodology, QA/QC procedure of link_link table, data translation between STN and WSLR in both directions, application of link_link table for giving name to ramps and recommendations for future work

DEDICATION

This thesis is dedicated to my family, friends and coworkers who helped to assist me with this research work.

ACKNOWLEDGMENTS

I am thankful to so many for the opportunity to do this research work. I would like to thank my family, friends, and colleagues for their help and encouragement throughout this research work and thesis writing.

I am very much grateful to Dr. Andrew J. Graettinger for helping me to guide not only in research but also in other aspects of my life. I would also like to thank Dr. Steven Parker at the University of Wisconsin-Madison for his help throughout this project. I would like to thank all of the members of GIS lab for helping me in various ways during working in this project as well as being awesome office-mates. I am forever grateful to all of these people.

Finally I would like to thank to the University of Alabama to provide me the opportunity to pursue my study as well as research.

TABLE OF CONTENTS

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

LIST OF TABLES ix

LIST OF FIGURES x

INTRODUCTION 1

1.1 Introduction 1

1.2 Thesis Organization 3

LITERATURE REVIEW AND BACKGROUND 4

2.1 Introduction 4

2.2 NCHRP 20-27(2) conceptual data model 5

2.3 Linear Referencing Systems used by WisDOT : STN and WISLR 7

2.4 link_link Table 9

2.5 link_link table coding process 11

2.6 Data movement from STN to WISLR 15

METHODOLOGY 17

3.1 Introduction 17

3.2 Updating procedure of link_link table 18

3.2.1 Data preparation for link_link table updating 18

3.2.2 Identifying Changes in LRS data 19
3.2.3 Updating link_link Table Records 21
3.2.4 Population of Start_Valid and End_Valid Date columns 21
3.3 QA/QC Procedure 22
3.3.1 County wide QA/QC 22
3.3.1.1 STN Link Check 23
3.3.1.2 WISLR Link Check 23
3.3.1.3 Gore Point Check 24
3.3.1.4 XY Connector Line Check 24
3.3.2 State wide QA/QC 26
3.3.2.1 Duplicity Check 27
3.3.2.2 STN Link Check 27
3.3.2.3 WISLR Link Check 28
3.3.2.4 STNstart and STNend check 28
3.3.2.5 STN continuity check 28
3.3.2.6 WISLRstart and WISLRend Check 29
3.3.2.7 WISLR continuity check 29
3.3.2.8 Date columns check 30
3.4 Data Translation 30
3.4.1 Data translation ambiguity on median crossover/intersection: 31
3.4.2 Data translation ambiguity at turn lanes 33
3.4.3 Data translation ambiguity at wayside 34
3.4.4 Data translation rules and options 35

3.4.4.1 Median cross over rules 36
3.4.4.2 Turn lane rules 37
3.4.4.3 Wayside Rules 38
3.4.4.4 Problem Rules: 39
3.5 Implementation of data translation rules and options 40
3.6 Conclusion 40
APPLICATION OF the link_link TABLE 41
4.1 Introduction 41
4.2 Methodology 42
4.3 Ramp Identification 45
4.4 Name of ramp identified through STN 45
4.5 Name of ramp identified through WISLR 49
4.6 Ramp name QA/QC 52
4.7 Results 53
4.8 Conclusion 54
RESULTS 55
5.1 Introduction 55
5.2 link_link table update results 55
5.3 Crash data translation from STN to WISLR results 55
5.4 Data translation from WISLR to STN results 56
5.5 Conclusion 58
CONCLUSION AND FUTURE WORK 59
6.1 Conclusion 59

6.2 Future Work 60

REFERENCES 61

APPENDIX A 63

APPENDIX B 68

APPENDIX C 74

APPENDIX D 102

LIST OF TABLES

Table 2.1 Names and description of main six columns of link_link table (Ryals, 2011; Morison, 2012) 10

Table 2.2 Names and description of five flag columns of link_link table (Ryals, 2011; Morison, 2012) 10

Table 2.3 Names and description of four date columns of link_link table (Ryals, 2011; Morison, 2012) 10

Table 4.1 Ramp table format 44

Table 4.2 List of tables and associated columns used form naming ramp identified using STN 45

Table 4.3 An example of ramp name generation 47

Table 4.4 List of tables and associated columns used form naming ramp identified using WISLR 49

LIST OF FIGURES

Figure 2.1 NCHRP 20-27(2) conceptual data model..... 6

Figure 2.2 Expression of conceptual LRS model (Scarponcini, 2002)..... 7

Figure 2.3 STN and WISLR system of Eau Claire county of Wisconsin..... 9

Figure 2.4. Example of link-link table coding process: (a) shows basic attribute information for STN link, (b) shows basic attribute information for WISLR links, (c) shows basic roadway representation in WISLR, (d) and (e) show basic roadway representation of STN, (f) presents the link _link table associated with the shown STN and WISLR links, (g) presents updated link _link table associated with the shown STN and WISLR links. 12

Figure 2.5 Inconsistency during data movement between STN and WISLR. (a) Data movement from STN to WISLR, (b) Data movement from WISLR to STN..... 15

Figure 3.1 (a) XY connector (black lines) of data points from STN (red lines) to WISLE (blue lines) for perfect relationship; (b) XY connector (black lines) of data points from STN (red lines) to WISLE (blue lines) for inconsistent relationship 26

Figure 3.2 Data point translation between STN and WISLR at median cross over: (a) aerial image of an intersection (b) moving point from higher resolution STN to lower resolution WISLR, four data point (1-4) move to one single point; (c) moving point from WISLR to STN, each of four data point move to every node of STN links in intersection. 32

Figure 3.3 Example of data translation in turn lane (a) aerial image of a turn lane marked, (b) data movement from STN to WISLR and there is no ambiguity, (c) each point on WISLR has two places to land on STN and ambiguity initiates 34

Figure 3.4 Example of data translation at wayside, (a) aerial image of a wayside, (b) data point 1-3 movement from STN to WISLR, (c) data movement 1-3 from WISLR to STN and every point 1-3 moves to every STN sites and ambiguity occurs..... 35

Figure 3.5 User interface of point moving program(WISLR to STN) before running..... 36

Figure 3.6 Data movement (WISLR to STN) options for median crossover 37

Figure 3.7 Data movement (WISLR to STN) options for turn lane 38

Figure 3.8 Data movement (WISLR to STN) options for way side 39

Figure 3.9 Data movement (WISLR to STN) options for problem flaged records in link_link table..... 39

Figure 4.1 Example of ramp name format..... 42

Figure 4.2 Ramp representation (a) Aerial image of ramps and surrounding area, (b) STN representation of ramps and names, (c) WISLR representation of ramps and names 43

Figure 4.3 Example of generating ramp name..... 46

Figure 4.4 Flow chart of naming procedure of ramp identified using STN links..... 48

Figure 4.5 Algorithm of naming procedure of ramp identified using WISLR links 51

Figure 4.6 Ramp with names in WISLR. Black lines represent WISLR links, yellow lines represent ramps and red lines represent STN links. Black dots are WISLR sites and green squares are STN sites. 52

Figure 4.7 Ramp identification pie chart (Total ramps : 3733) 53

Figure 4.8 Ramp naming pie chart (Total ramps : 3733)..... 54

Figure 5.1 Point moving options from WISLR to STN and report after running the program using every hundredth mile point on STN of Dane county, WI 57

Figure 5.2 Point moving options from WISLR to STN and report after running the program using RP crash data provided by WisDOT..... 58

CHAPTER 1

INTRODUCTION

1.1 Introduction

A linear referencing system consists of a set of line features, on which an event is localized by referencing the associated line and measurement from a known point along the associated line. Usually measurement is taken from the starting point of the associated line. Data about linear features like roads, pipelines, rivers, electric cables etc. is often collected, stored, displayed and analyzed using linear referencing system (LRS). LRS can follow cartographic representation or not and different LRS can provide different information.

Linear referencing system (LRS) used for data collection and displaying are often not the same linear referencing system (LRS) used for analysis. The difference can be with representation, resolution and attributes. The Wisconsin Department of Transportation (WisDOT) maintains two independent linear referencing systems of different resolution for state routes and local roads. The State Trunk Network (STN) was developed in early 1990s for interstates and state routes in Wisconsin. About 10 years latter a new system called Wisconsin Information System for Local Road (WISLR), which mostly follows cartographic orientation of road network, was developed for local roads of Wisconsin. As these two systems were developed independently, there are significant differences in expressing information. For example intersections are expressed in more detail in STN using multiple straight lines and points where in WISLR only one point is used to represent an intersection. Approximately 12,000 miles of state road and 100,000 miles of local road are included in STN and WISLR

respectively. All roads in STN are represented as part of the WISLR system. WisDOT is using WISLR in police officer cars as part of the Incident Locator Tool (ILT). WISLR is used to locate and store crash data, because it is a cartographic representation of the road network but DoT analysis tools use the STN network and are not compatible with WISLR (Graettinger, et al., 2013). Therefore a methodology is necessary to move data from one network to another for the overlapping 12,000 miles state road network.

A functional merge called link_link table was developed to define a relationship between STN and WISLR (Graettinger, et al., 2009; Ryals, 2011; Graettinger, et al., 2013). Using the link_link table, data can be translated from STN to WISLR with the help of a previously developed 'point moving program' and a procedure was developed to update the link_link table yearly to reflect the changes in network year to year (Morrison, 2012). Data movement from STN to WISLR works well though movement in the reverse direction, from WISLR to STN, is still challenging. Some rules were proposed by Graettinger, et al., 2013 to manage the data movement ambiguity.

WisDOT is currently using a Reference Point (RP) system to record crash location data (Graettinger, et al., 2013). In this system, a crash point is expressed with reference point number (Unique ID), link id, and an offset (positive distance from the start point of a link). The incident location tool (ILT) is based on cartographic road network in WISLR and is being used by law enforcement officers to record crash data. The crash data is moved manually by assigning an STN link id and offset for each crash data using the existing RP coding method. Resolution difference between WISLR and STN does not allow moving these crash data from WISLR to STN without ambiguity.

This research is done to apply a functional merge (link_link table) to move data between the LRSs at WisDOT. Data such as crash locations and other business data are moved in this research. This research develops an algorithm that allows data movement from WISLR to STN and controls the ambiguities. The research also applies the link_link table to automatically name to the ramps in WISLR based on STN data. Improving link_link table updating procedure as well as formalizing quality assurance/quality control procedure of the link_link table are also described herein.

1.2 Thesis Organization

The thesis consists of six chapters. Chapter 2, Literature Review and Background, details conceptual model of Linear Referencing Systems, data structure of STN and WISLR, and previous works related to data movement between these two systems. Chapter 3, Methodology, presents updating procedure of link_link table, improved quality assurance/ quality control procedure and basically an algorithm for translating data between STN and WISLR in both directions. Chapter 4, Application of the link_link table details the procedure to give names to ramps in WISLR dynamically. Chapter 5, Results, discusses the efficiency of improved updating procedure and quality assurance/ quality control (QA/QC) procedure, and workability of the algorithm to translate data including ramp names. Chapter 6, Conclusion and Future Work, presents a discussion about the results of this research and future work

CHAPTER 2

LITERATURE REVIEW AND BACKGROUND

2.1 Introduction

Linear referencing is a system, where measurements along a linear element are used to express the location of an event or a feature (point) for any network of linear features like roads, pipelines, electric lines, etc. The measurement is taken from a known point usually the start point of a linear element. An event is referenced with a link ID and an offset from a known point on that link. A link is a linear segment of a network and offset is the measurement from the starting point of a link to the point event. The advantage of using Linear Referencing System (LRS) is that it does not need spatial reference as cartographic data set does. Moreover attribute table integrated in LRS can be used to define geographic data specifying the link ID and offset. Complexities arise during transferring data between different LRSs or combining multiple LRSs.

A generic data model for linear referencing system was introduced by Alan Vonderohe in August, 1994 and referred as the NCHRP 20-27(2) data model (Vonderohe, 1995). The model is multilevel linear referencing system. A five level conceptual linear referencing model system from NCHRP 20-27(2) is shown in Figure 2.1, and the following section details the model concept.

2.2 NCHRP 20-27(2) conceptual data model

In NCHRP 20-27(2) data model, the levels are named as cartographic representation, linear datum, network, linear referencing method and event. Cartographic representations are referred as sources. Sources are spatial representation of physical real life network for example a road way network. This is the level from where all data is collected and also where data is mapped after analysis. Numerous cartographic representations could be mapped onto the linear datum as well as multiple networks can be part of linear datum. Various linear referencing methods (LRMs) can be mapped on each network and Events can be expressed with any one of the linear referencing methods (LRMs).

The linear datum represents the network and consists of anchor points and anchor sections. Anchor points express well defined point locations like intersection of two streets. An anchor section is the connector of a pair of anchor points and represents the street segments between two associated anchor points. Anchor sections are allowed to cross each other without intervening anchor point.

Networks represent allowable transportation paths through a set of roadways. The networks consist of nodes and links. Nodes are the turn points and links are the connector of two nodes. In a link-node system, links are directional and represents the direction of traffic flow. A node is expressed with anchor section id and a positive distance (offset) from the start point of that anchor section. It is not necessary to put a node at every anchor point. Nodes should be where the flow direction can change.

The level linear referencing methods (LRMs) states methods to describe a location. WisDOT is uses a link-offset method for the STN and WISLR systems (Graettinger, et al., 2009; Ryals, 2011). In these systems an event is described with the link id and distance from the start

point of a link. WISLR also uses an on-at offset linear referencing method (LRM) to locate events.

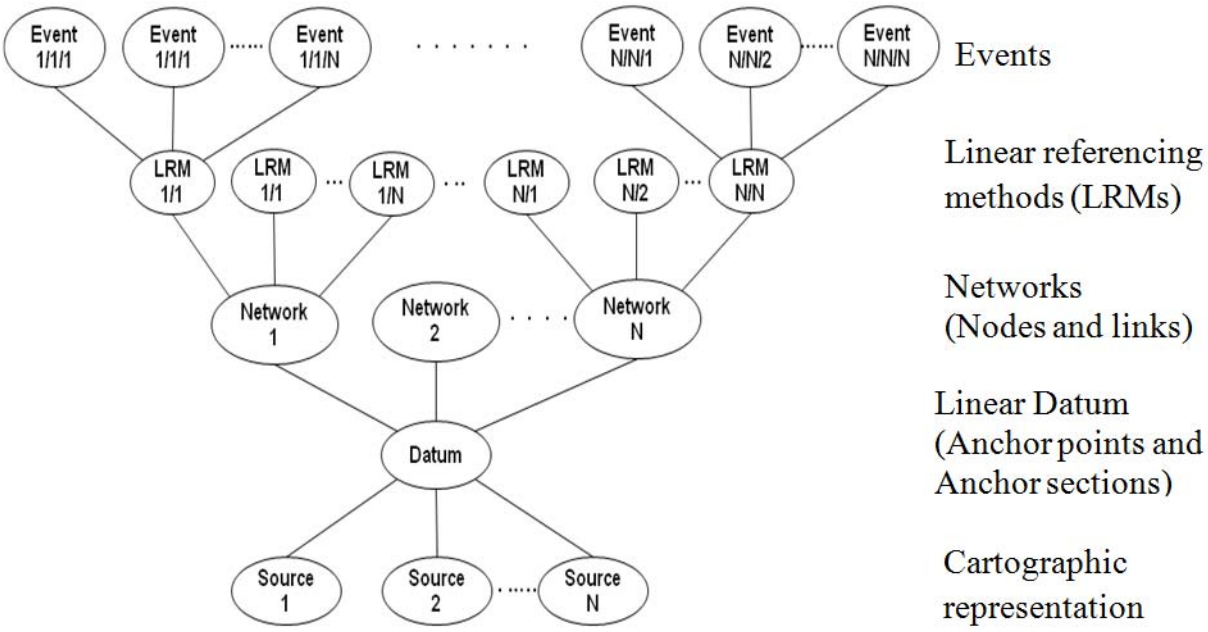


Figure 2.1 NCHRP 20-27(2) conceptual data model

Expressing an event on the datum involves describing the event with a LRM, then on a network link, and then on an anchor section (datum). Figure 2.2 illustrates how an event can be expressed with LRS. In the cartographic representation, like on a GIS map consisting of lines, there are no constraints on where lines start or end, where lines break, etc. Break points are not necessarily at anchor points, network nodes, or reference point locations. Under these circumstances, LRS approach allows for events to be on lines (cartographic representation) through the LRM, network, and datum level.

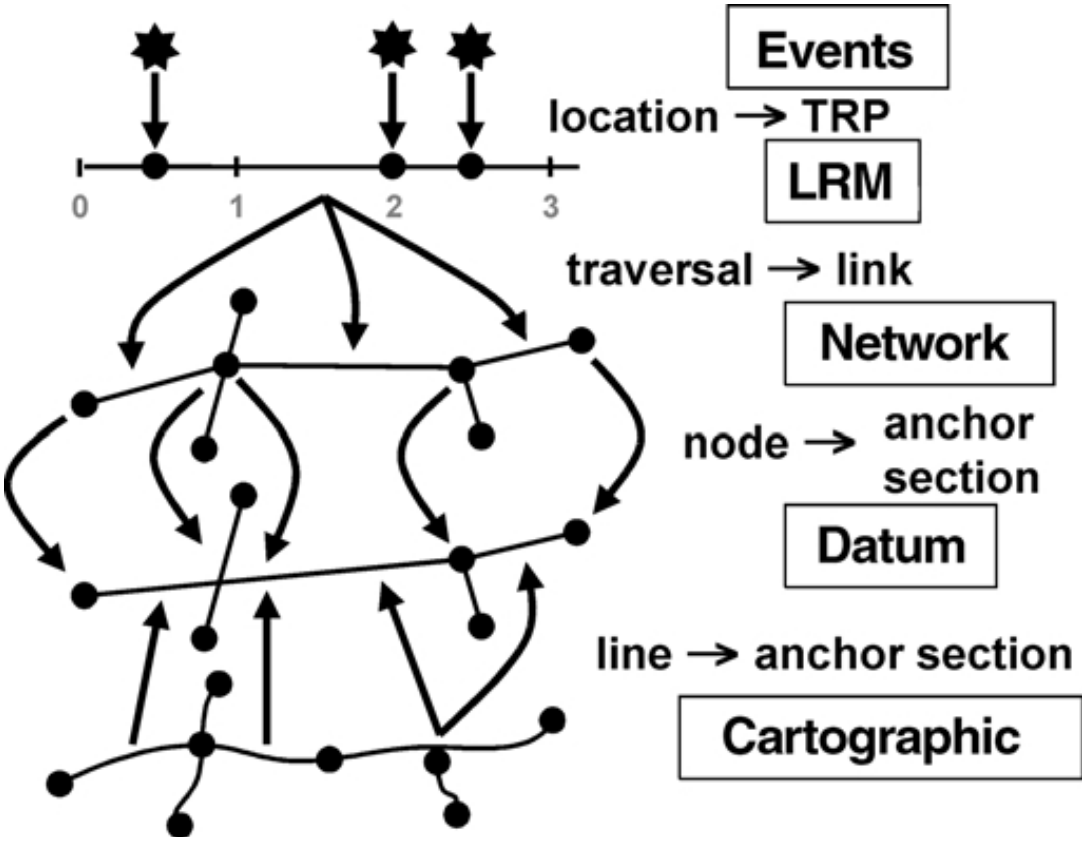


Figure 2.2 Expression of conceptual LRS model (Scarponcini, 2002)

2.3 Linear Referencing Systems used by WisDOT : STN and WISLR

The State Trunk Network (STN) and Wisconsin Information System for Local Roads (WISLR) are two different linear referencing systems that are currently used by WisDOT simultaneously. Both systems follow the basic rules of a linear referencing system (LRS). The State Trunk Network (STN) was developed to manage transportation information on state highways and interstates in Wisconsin. STN consists of state controlled roads only. In order to have more detail and accurate transportation data along with maintenance and safety analysis, WisDOT introduced Wisconsin Information System for Local Road (WISLR). These two LRSs

were developed separately and independently though all roads in STN system are included in WISLR system. The STN system (thick red lines) that is overlaid with the WISLR system (thin black lines) in Eau Claire county of Wisconsin is shown in Figure 2.3.

The STN system consists of links and sites (nodes) with cartographic chains. Each link has an ID, length, from-site and to-site. The ID is a unique link identifier; length is the driven measured length of the link; and the from-site and to-site provide the directional information of link. The link runs from the from-site to the to-site point. In the cases where two links run one over another, the from-site and to-sites are used to identify the direction of the link. STN link length is measure in thousandth of a mile and the accuracy is hundredth of a mile. STN is not a cartographic representation of the roadway, rather, STN links are connector straight lines between two sites (nodes). Sites are placed where a driver can turn from one state road to another state road.

The WISLR system also consists of links and sites (nodes) as STN does. The attributes of WISLR are the same as STN except the length of link in WISLR is measured to the nearest foot. The other difference between STN and WISLR is that WISLR links provide cartographic representation of actual transportation features.

STN was fully developed and functioning when WISLR was developed. Though WISLR contains all state route roadways, abandoning STN in favor of WISLR was not practical as STN contains information that WISLR does not (Graettinger et al., 2008; Graettinger et al., 2009; Ryals 2011). Moreover STN and WISLR are using two different types of datum (Ryals, 2011). In STN, the datum relates the LRS to features in the network to define the theoretical location of the features rather than expressing spatial location directly. On the other hand, WISLR uses a datum which expresses the spatial location directly (Ryals, 2011). Though both STN and

WISLR have the datum integrated in the respective network, both lack a distinct datum (Ryals, 2011). Besides these STN and WISLR are using different LRM (Graettiner et al., 2008; Graettinger et al., 2009). As WisDOT has to maintain two LRS simultaneously, accurate data translation demands continuous updating of the two systems.

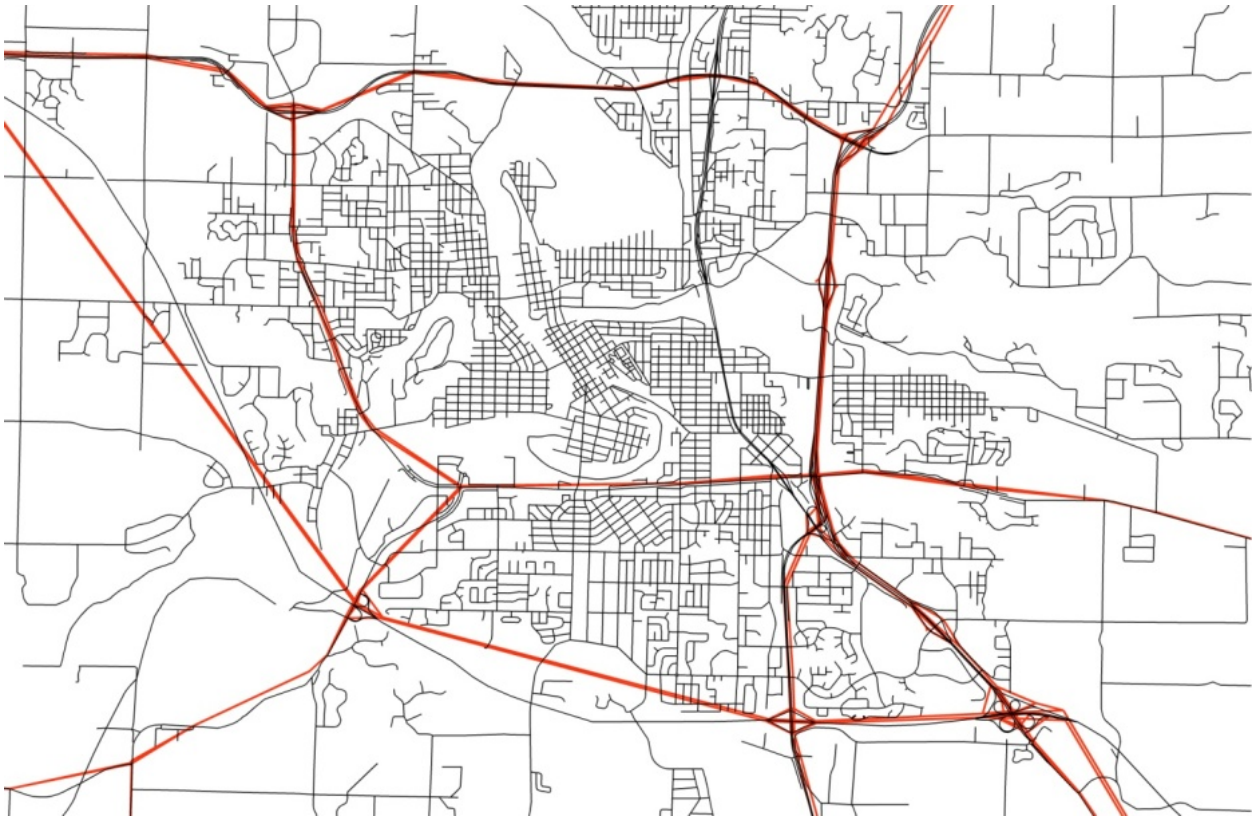


Figure 2.3 STN and WISLR system of Eau Claire county of Wisconsin

2.4 link_link Table

link_link table is a functional merge which defines a relationship between the STN and WISLR systems in order to relate one system to another as simply as possible without compromising functionality (Ryals, 2011). The table has six main columns, five flag columns, four date columns, two comments columns and a county name column. The table was coded county by county to distribute the coding effort among multiple coders.

Table 2.1 to Table 2.3 provide the name and description of columns of link_link table. Ryals (2011) detailed the main columns of the link_link table and the flag columns along with comments columns. Morrison (2012) detailed date columns.

Table 2.1 Names and description of main six columns of link_link table (Ryals, 2011; Morison, 2012)

STNid	STNstart	STNend	WISLRid	WISLRstart	WISLRend
Unique STN link identifier	Start point measure of STN link segment	End point measure of STN link segment	Unique WISLR link identifier	Start point measure of WISLR link segment	End point measure of WISLR link segment

Table 2.2 Names and description of five flag columns of link_link table (Ryals, 2011; Morison, 2012)

T	M	G	W	P
Turn-Lanes Represents turning traffic lane. Usually maintained in STN	Median crossover Represents the pavement segment of intersection. Usually maintained in STN	Gore points At the acute-angle pavement intersection	Weight-stations Weight-stations, Park & Rides	Problem Where the inconsistency could not be defined by other flag columns

Table 2.3 Names and description of four date columns of link_link table (Ryals, 2011; Morison, 2012)

Record_Created	Record_Historic	Start_Valid	End_Valid
Date when the record is entered into the link_link table	Date when the record is considered invalid in link_link table	Date when the relationship between STN and WISLR segments starts	Date when the relationship between STN and WISLR segments expires

The comments columns provide a location to store information which is necessary to understand the discrepancies or reasons for changing of a particular record. The county column provides the county where the link exists. Each row represents a segment of road in the real world what is in both LRSs maintained by WisDOT.

2.5 link_link table coding process

The link_link table provides all current relationships between STN links and WISLR links along with historic relationships. A relationship could be one to one or one to many depending on the orientation and arrangement of STN and WISLR links. A basic one directional one to many relationship example of coding link_link table is shown in Figure 2.4. Figure 2.4 (a) and Figure 2.4 (b) shows the attributes of STN and WISLR. The WISLR links associated with STN link S1 are W1, W2 and W3 which are shown in Figure 2.4 (c). Figure 2.4 (d) shows basic one directional STN link having STNid S1. The direction for all links is from left to right that can also be seen in attribute table data (Figure 2.4 (a, b)). Figure 2.4(e) shows a new STN link of STNid S2 when STN S1 becomes historic.

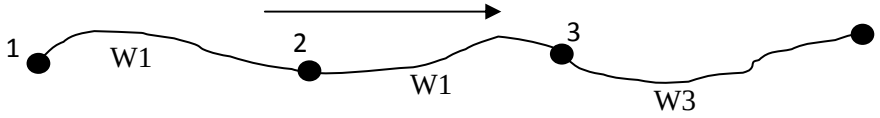
(a)

STN Link id	From site	To site	Distance	Start Valid	End Valid
S1	5	6	80	8/15/2002	2/2/2012
S2	7	8	66	2/2/2012	

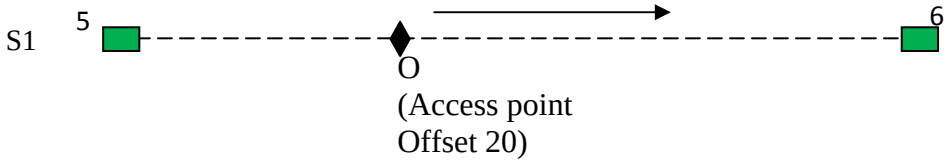
(b)

WISLR Link id	From site	To site	Distance	Start Valid	End Valid
W1	1	2	100	7/15/2002	
W2	2	3	50	9/28/2005	
W3	3	4	200	7/15/2002	

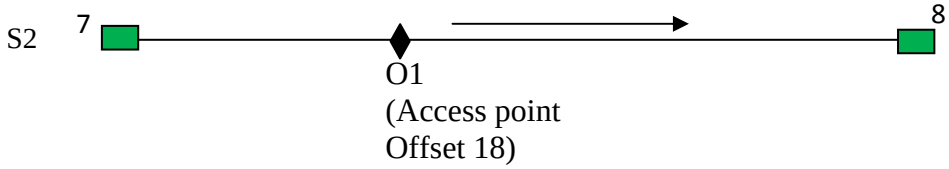
(c)



(d)



(e)



(f)

STNid	STNstart	STNend	WISLRid	WISLRstart	WISLRend	Record_Created	Record_Historic	Start_Valid	End_Valid
S1	0	20	W1	0	100	5/20/2011		8/15/2002	
S1	20	32	W2	0	50	5/20/2011		9/28/2005	
S1	32	80	W3	0	200	5/20/2011		8/15/2002	

(g)

STNid	STNstart	STNend	WISLRid	WISLRstart	WISLRend	Record_Created	Record_Historic	Start_Valid	End_Valid
S1	0	20	W1	0	100	5/20/2011	7/09/2012	8/15/2002	2/2/2012
S1	20	30	W2	0	50	5/20/2011	7/09/2012	9/28/2005	2/2/2012
S1	30	70	W3	0	200	5/20/2011	7/09/2012	8/15/2002	2/2/2012
S2	0	18	W1	0	100	7/09/2012		2/2/2012	
S2	18	30	W2	0	50	7/09/2012		2/2/2012	
S2	30	80	W3	0	200	7/09/2012		2/2/2012	

Figure 2.4. Example of link-link table coding process: (a) shows basic attribute information for STN link, (b) shows basic attribute information for WISLR links, (c) shows basic roadway representation in WISLR, (d) and (e) show basic roadway representation of STN, (f) presents the link _link table associated with the shown STN and WISLR links, (g) presents updated link _link table associated with the shown STN and WISLR links.

Figure 2.4 (f) shows an example of coded link_link table for brand new records using the given information in the attribute tables along with STN and WISLR line work (Figure 2.4 (a, b, c, d)). The first row of Figure 2.4(f) was populated with STNid (S1), STNstart(0), WILRid (W1), WISLRstart (0) and WISLRend (100). Start_Valid date of S1 link, according STN link attribute table, is 8/15/2002 (Figure 2.4(a)) and Start_Valid date of W1 link, according to WISLR link attribute table, is 7/15/2002 (Figure 2.4(b)). The Start_Valid date in link_link table (Figure 2.4(f)) should be the most recent date of these two dates (8/15/2002 and 7/15/2002) and is stored in the Start_Valid date column. This date expresses the date from when the relationship between STN link (S1) and the associated WISLR link (W1) starts. For the second row, STNid (S1), WISLRid (W2), WISLRstart (0), WISLRend(50), Record_Created (5/20/2011) and Start_Valid (9/28/2005) columns were populated using available data in the attribute tables. The third row of the link_link example table (Figure 2.4(f)) was populated like the second row, except the STNend column. This was the very last segment of S1 link and the STNend should be the full length of the S1 link (80) which is entered in the STNend column. To populate the missing values in the STNstart and STNend columns two approaches were used. There is an access point (O) on S1 link (Figure 2.4(d)). The access point expresses the equivalent point for end point of the W1 link, as well as start point of W2 link. Therefore, the STNend for second row, and STNstart of third row of Figure 2.4(f), were populated with the offset value of the access point O (20). The remaining part of S1 link was divided with the ratio method to express the W2 and W3 WISLR links. The length of the remaining part of the STN link is 60 (80-20 = 60) and total length of W2 and W3 links is 250 (50 + 200 = 250). The length of STN link segment for the second row of Figure 2.4(d) is 12 (60 * 50/250). Therefore, STNend of second

row, and STNstart of third row, of Figure 2.4(f) was 32 (20 + 12 = 32). All links were valid during the time of coding, and therefore, Record_Historic and End_Valid column are blank.

Figure 2.4(g) shows an example of an updated link_link table using the given information shown in Figure 2.4 (a, b, c, d, e, f). On 7/09/2012, when the table was updated, it was seen that STN link S1 was no longer a valid link and a new STN link (S2) took the place of S1. All WISLR links remained valid and the records associated with STN link S1 was made historic with the date 7/09/2012 entered in Record_Historic column for first three rows of link_link table (Figure 2.4(f)). At the same time, new records for the new STN link, S2, and associated WISLR links (W1, W2, W3) were entered into the table (Figure 2.4(f)). The fields for new records were populated as described earlier. From the attribute tables, it can be seen that the End_Valid date of the STN link S1 was 2/2/2012, which means the relationship between STN link S1 and the associated WISLR links (W1, W2, W3) was invalid from 2/2/2012. That date is stored in End_Valid column associated with STN link S1 (Figure 2.4(f)). If both the STN and the associated WISLR links are invalid, End_Valid date in the link_link table should be the oldest date between them (End_Valid date of STN and End_Valid date of WISLR).

This example describes a basic coding and updating procedure of the link_link table. There are various discrepancies between STN and WISLR that are identified and captured during the coding and updating of the link_link table. Five flag columns are used to accommodate these discrepancies (Table 2.2). Detail procedure of coding can be found in reference [3] (Ryals, 2011). Detail updating procedures are described in Chapter 3.

2.6 Data movement from STN to WISLR

The link_link table functions well when translating data from STN to WISLR. Ambiguities occur when the data is moved in reverse direction. Most of the ambiguities are of inconsistency and duplicity. In the figure 2.5 the red lines are STN links, black stars are event on STN and the green square under the black stars are STN sites, blue lines are WISLR links, black triangle is event on WISLR after moving from STN, brown circle under the black triangle is WISLR sites and the black dashed lines are indicating the event movement direction. The Figure 2.5 shows that STN, the higher resolution representation of road network, four links represent an intersection of two divided roads. On the other hand the same intersection in WISLR is represented with only a point indicating the intersection of two lines. Figure 2.5(a) shows that multiple events (black stars) on intersection on STN translate to only one point (black triangle) on WISLR. In Figure 2.5(b) it is shown that there are multiple options for an event to be landed on STN during translating from WISLR to STN.

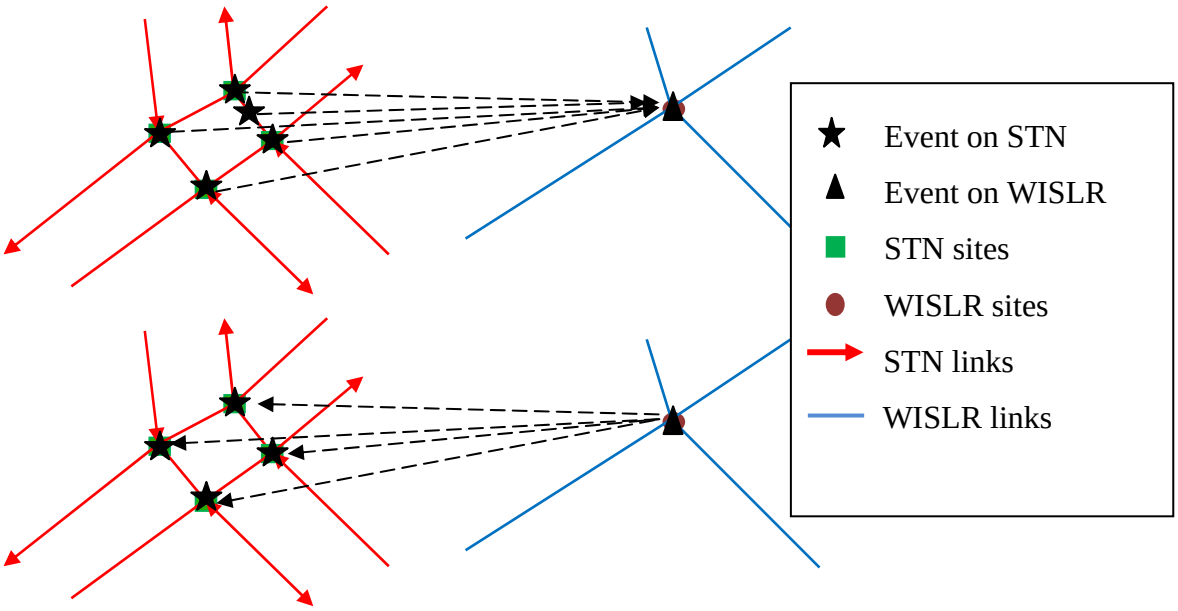


Figure 2.5 Inconsistency during data movement between STN and WISLR. (a) Data movement from STN to WISLR, (b) Data movement from WISLR to STN

Currently, crash data is being collected based on the WISLR network and then manually move to the STN network. Introducing an automated system to move data from either in STN to WISLR or WISLR to STN would take less time and save resources. To eliminate ambiguities, or to keep track of ambiguities, some rules were proposed by Graettinger, et al (2012) and efforts were given to apply them. This research created and tested data transition tool. The tool applies various rules to map data in a controlled way, without duplicity and ambiguity, as well as produce a report of data that need manual attention.

CHAPTER 3

METHODOLOGY

3.1 Introduction

This research is performed to manage crash data in the state of Wisconsin, to update STN-WISLR functional merge (link_link) table, and to show the capability of link_link table. The focus of this research can be divided into four categories: (1) to develop a tool to translate geographical data, in a user controlled way, between two linear referencing systems, State Trunk Network (STN) and Wisconsin Information System of Local Roads (WISLR); (2) to improve updating procedure of a previously developed functional merge (link_link table) (Morison, 2012) between two LRSs (STN and WISLR), (3) to incorporate a QA/QC procedure used during updating of the link_link table; and (4) to use the functional merge (link_link table) dynamically to provide names to ramps in WISLR.

The link_link table was first created based on 2009 line work provided by WisDOT (Graettinger, et al., 2009, Ryals, 2011). The link_link table was then updated based on 2010 data and four date columns were incorporated to keep the track of temporal changes (Morison, 2012). In this research, the latest link_link table is updated based on 2011 line work and attributes provided by WisDOT. Steps were taken to make updating and QA/QC procedure more programmatic rather than manual. Attention was paid translating crash data between STN and WISLR system controlling the ambiguities between these two systems by following user defined rules and options where ambiguity occurs. Finally initiatives were taken to use the link_link

table to provide names to ramps in WISLR dynamically based on connecting roads, and to accommodate ramp name changes if any occurs in future.

3.2 Updating procedure of link_link table

Updating link_link table involves including new STN-WISLR relationship and making historic outdated STN-WISLR relationship in the link_link table of previous year. In this way link_link table provide the most updated relationship that exists between STN and WISLR. The link_link table updating starts with, obtaining current data; then identifying changes compared with past year data; then entering the new records in link_link table; and making historic records outdated in link_link table. Five different types of data are necessary to update link_link table: (1) STN links, the STN line works with directional information and validity date; (2) STN sites, a STN link is the connector of two STN sites; (3) WISLR links, WISLR line works with directional information and validity date; (4) WISLR sites, a WISLR link is connector of two WISLR sites; (5) Access points, equivalent points on STN, that indicate intersection or other point of interest on WISLR.

3.2.1 Data preparation for link_link table updating

To update the link_link table current line work of STN and WISLR must be obtained and any changes with previous data sets must be identified. Updated shapefiles (line works) and attribute tables/databases are provided by WisDOT. The route-link table in the database provides information on whether a link is current or historic. There are two date columns in the route-link table for current date (DT_RTE_LINK_CURR) and historic date (LCM_DT_HSTL). Current date tells the date from when the route-link relationship starts, and the historic date tells the date from when a route-link relationship is invalid. The route-link table is filtered by selecting records whose route-link current date is earlier than last day of the year and route-link

historic date was null or newer than first day of the year. The attribute table of shapefile of a LRS (STN or WISLR) provided by WisDOT is then joined with the list of current links based on the link ID. The joined records are selected and the shapefile is exported. The exported shapefile provides the current records for the year being updaed. This step is done for both STN and WISLR separately to obtain the current link shapefiles along with attribute tables of STN and WISLR respectively.

Current sites or nodes are obtained by selecting sites whose historic date is null or newer than the first day of the year for STN and WISLR separately. In this case, WisDOT supplies reference site table (DT_REF_SITE) which is used to list the nodes.

Access points are the equivalent point of WISLR on STN. These points indicate the point of interest of WISLR LRS, on STN LRS with STNid and offset. Access points are belong to STN. In access point table there is no date information. The table provides an STN link ID, offset and street name for every point. The access point table is updated by WisDOT by deleting outdated access points and including new access point. Therefore, the current access point table is checked when updating the link_link table.

3.2.2 Identifying Changes in LRS data

There are five basic changes that are considered during updating of the link_link table. These changes are: addition of new STN links; addition of new WISLR links; deletion of STN links; deletion of WISLR links; and changes of access points.

Identifying these changes is done with ArcMap 10 join command. New links are identified by joining old line work (previous year) shapefile attribute table with current line work (created as described in section 3.2.1) attribute table based on link ID keeping all data. In the joined attribute table, non-joined records (‘link ID’ column from current line work is null) are

new links as they were not included in previous year data. This step is done both for STN and WISLR separately to obtain new STN and WISLR links respectively.

Deleted links are identified by joining current line work (created as described in section 3.2.1) shapefile attribute table with old line work (previous year) shape file attribute table based on link ID keeping all data. In the joined table, non-joined records (link ID column from old line work is null) are deleted links as they were in old data table but not in current data table. This step was done both for STN and WISLR separately to obtain deleted STN and deleted WISRL links respectively.

New, deleted or changed access points were identified by following a different approaches. In the access point table, each access point has a link ID (RWLK_ID), intersecting road name or point of interest name (ASCI_INTS) and offset value (ASCI_PT1). Identifying access point changes needs to consider all three of these attributes. To do this, a new column iss created in the access point attribute table and populated that column with concatenated link ID, intersecting road name, and offset value separated by underscore (_). The concatenation is done for both current and old access point shapefile attribute tables. The old access point shapefile attribute table is joined to the current access point shapefile attribute table based on new concatenated column. The non-joined records shows the new/changed access point data. The joining process is done again in the reverse direction (current to old) to obtain deleted or changed access point data. The combination of these two access point data tables (new and deleted or changed access points) provides the complete list of affected access points table for the current year.

3.2.3 Updating link_link Table Records

The updating procedure for the link_link table is similar to link_link table coding process as described in Chapter 2 (section 2.5). The updating is done for new and deleted links (both in STN and WISLR) as well as links having access point changes. In most of the cases, a deleted link is replaced with a new link. Moreover, WISLR link changes often occur at the same location as changes in STN links. As a result, updating of WILR links is done with the updating the associated STN links. The updating procedure is done in order of STN, WISLR, and access point changes. Updating is also done county by county. County specific data is separated from the entire state data set with the help of ArcGIS 10 "select by location" tool. The updating procedure is manually done for every column of the link_link table except Start_Valid and End_Valid date columns. These two columns are populated programmatically once link_link table for a county is updated.

3.2.4 Population of Start_Valid and End_Valid Date columns

The Start_Valid date and End_Valid dates come from the route-link database table where every route-link relationship for every link and route in Wisconsin is stored. The current date in the route-link table for a link is the first date on which a relationship between a link and a route starts. Similarly, route-link historic date for link is the latest date after which there is no relationship between a link and a state route.

A query is used to make date table having link ID, Start_Valid (MIN_DT_RTE_LINK_CURR) and End_Valid (MAX_LCM_DT_HSTL) for STN and WISLR separately. The Start_Valid column of the link_link table for a record is populated with the latest Start_Valid date between Start_Valid date for STN (MIN_DT_RTE_LINK_CURR in STN date table) and Start_Valid date for WISLR (MIN_DT_RTE_LINK_CURR in WISLR date table).

Similarly, the End_Valid column of link_link table is populated with the oldest End_Valid date between that of STN (MAX_LCM_DT_HSTL column in STN date table) and that of WISLR (MAX_LCM_DT_HSTL column in WISLR date table). If any STN link has an End_Valid date in the STN date table but the associated WISLR link does not have an End_Valid date in WISLR date table, the End_Valid date in the link_link table is the End_Valid date of the associated STN link. The End_Valid date of a particular record in the link_link table should be the End_Valid date of the associated WILSR link if the corresponding STN link does not have an End_Valid date. A comparison between two End_Valid dates is done only when the End_Valid date exists for both STN and the associated WISLR link. A custom Python script (Appendix A) was written to populate Start_Valid and End_Valid columns of link_link table from two date tables.

3.3 QA/QC Procedure

The QA/QC procedure is done in two steps. First, QA/QC is done on each county wise. A second QA/QC procedure is done after appending all county data into one state wide table (link_link table for whole state). County wise QA/QC procedure is mostly similar to the QA/QC procedure used during the original link_link coding (Ryals, 2011), and QA/QC procedure of updating link_link table (Lane, 2012) though there are some differences which are briefly discussed in the following sections.

3.3.1 County wide QA/QC

There are four basic steps associated with a county link_link table QA/QC. The steps are STN link check, WISLR link check, Gore point check, and XY connector line check. All these checks were introduced during first link_link table coding (Ryals, 2011). All these checks are done only for current records. So, a new table having only current records (Record_Historic is null) is produced from the link_link table.

3.3.1.1 STN Link Check

All current STN links must be included in the in link_link table. To check this requirement, attribute table of STN shapefile for a county is joined to the link_link table based on the link ID keeping all record. Any un-joined records (link) in the joined attribute table need to be included in the link_link table. In order to make sure that all current links are in link_link table, an STN shapefile having only current STN links (obtained previously) is joined to the link_link table containing current records (Record_Historic is Null) based on link ID. Any un-joined links are manually investigated.

3.3.1.2 WISLR Link Check

WISLR link checking involves joining the link_link table for a county to the WISLR shapefile attribute table for that county based on WISLR link ID. If there are any un-joined records, the associated WISLR link does not exist in WISLR shapefile. So the link-link record is made historic. The second WISLR check involves joining the link_link table having current records of a county to the WISLR shapefile having current WISLR links. Un-joined records are looked at and edited appropriately. The next step in WISLR link checking procedure is visual checking which involves joining the WISLR shapefile of a county to the link_link table of current records for that county. After joining, different color symbols are assigned for joined and un-joined records. Joined records are then checked visually with a map to assure continuous WISLR link connectivity

3.3.1.3 Gore Point Check

Gore points are seen where roads merge or split at an acute angled intersection. A gore point is flagged in the link_link table because measuring methods are different in WISLR and STN system at gore points. For example, in WISLR system, the ramp length is measured along the centerline of the ramp and main road; whereas in the STN system, the measurement of the ramp is taken along the grass line from the gore point. Therefore, the ratio of WISLR link length and associated STN link(s) length is not close to the standard ratio of WISLR and STN length (5.28). The gore point check is done by joining the link_link table having current records in a county with STN shapefile based on link ID. Then links are given different color symbol based on gore column. Records are coded as: to (T), from (F) or both (B). A visual check is done at gore points to ensure that gore points are correctly coded in link_link table.

3.3.1.4 XY Connector Line Check

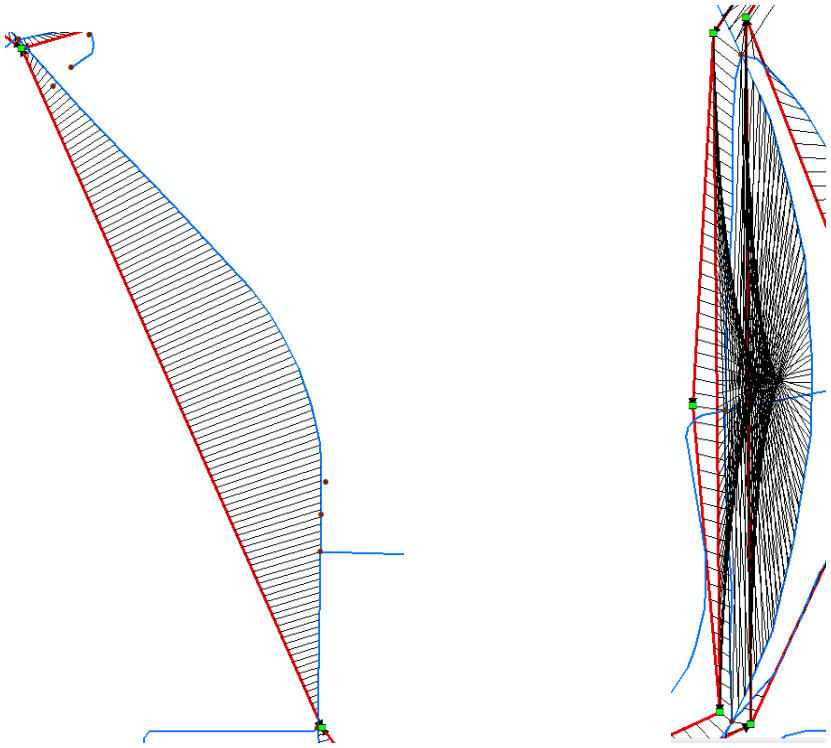
This check is done to check the spatial relationship between STN and WISLR links, which is the basis of link_link table. This check is done for only current records of link_link table. The very first step in doing this check is to generate points along all STN links in a county at hundredth of a mile (10 units in STN) interval. Generation of points is done with a previously created "STN Points Generator" program (Ryals, 2011). The input for this program is a spreadsheet file containing the STNid, full length of respective STN link, and the associated county name. This program generates a table named STN_Points in the database that stores the link_link table. STN_Points table contains a unique identifier (UNIQUE_ID) column, an STN link ID (Link_ID) column, and an STN offset (Link_Offset) column. Previously developed "Point moving program from STN to WISLR" (Ryals, 2011) is used to translate points that are coded to STN and move those points to WISLR links. The input of the "Point moving program

from STN to WISLR" is a database file containing the link_link table and STN_Points. The output of "Point moving program from STN to WISLR" is a table called "WISLR_final". WISLR_final table has three columns: the unique identifier of a point, the WISLR link, and WISLR offset.

After generating points for both STN and WISLR, STN_Points and WISLR_final tables are imported into GIS (ArcGIS 10). Two route files are created for STN and WISLR with "Create Route" tool integrated in ArcGIS 10. Then STN points are mapped using "display route event" tool in ArcGIS 10. A new event file showing STN points on STN route is created and is exported as a shapefile named STN_Points. The STN_Points shapefile is then added to the ArcMap and two new columns (STN_X and STN_Y) are added to the attribute table. STN_X and STN_Y columns were then populated with the latitude and longitude of the points respectively using the "calculate geometry" function. The data frame coordinate system should be the coordinate system of original WISLR system. Same procedure is performed for WISLR to get latitude and longitude of points on WISLR links. Then, the two attribute tables (STN_Points and WISLR_Points) are joined based on the unique identifier (UNIQUE_ID) and exported as a .dbf file (XY_Connector_Table.dbf).

The XY_Connector_Table.dbf is then used to create XY connector lines by applying "XY to line" tool in ArcGIS 10. The input for this is the XY_Connector_Table.dbf. The start point of the connector line is the point on the STN link and end point of the line is the associated point on WISLR link. Every line is a connector between two associated points; one on STN and one on WISLR.

The XY connector lines appear almost parallel for appropriately coded relationships between STN and WISLR links as shown in Figure 3.1 (a). If the relationship is not coded correctly, or the attribute table provides inconsistent information or cartographic representation of links was inconsistent, the XY connector lines cross each other as shown in Figure 3.1 (b).



(a) (b)

Figure 3.1 (a) XY connector (black lines) of data points from STN (red lines) to WISLE (blue lines) for perfect relationship; (b) XY connector (black lines) of data points from STN (red lines) to WISLE (blue lines) for inconsistent relationship

3.3.2 State wide QA/QC

After appending all county link_link tables into one complete statewide link_link table, state wide QA/QC is performed. A custom Python script (Appendix B) was written to perform some steps of this QA/QC procedure programmatically. A brief description of the checks that

are done in this procedure is given in the following sections. The input of the code is the entire link_link table, STN link attribute table and the WISLR link attribute table. STN and WISLR attribute tables are imported into link_link table database and the database with these three tables serve as input for statewide QA/QC.

3.3.2.1 Duplicity Check

There can be incidents when a link is incorrectly coded multiple times. A convention is followed for links crossing the boundary line of a county during coding of the link_link table. If a link crosses the east or south boundary of a county, that link is coded under that county. But this convention is not sufficient to avoid all duplicity. For example, a link could be coded multiple times under different county names when a link just cuts across a corner of a county and possibly touches three counties.

To do a duplicity check, all records in link_link table are considered. If every column of two records are exactly same, one record is deleted. The second step of this check is done for current records only. If STNid, STNstart, STNend, WISLRid, WISLRstart, WISLRend and Record_Created columns of two records are same, one of these two records is deleted.

3.3.2.2 STN Link Check

All STN links in a shapefile should be in the link_link table. Though this check is done at the county level QA/QC, this check is also perform one more time for the entire link_link table. In this case, the STN shapefile attribute table is joined to the link_link table based on the STN link ID. Any un-joined links are manually checked and added to the link_link table.

3.3.2.3 WISLR Link Check

This check is done by joining the link_link table to the attribute table of the WISLR shapefile based on the WISLR link id. The un-joined records in this operation show the links that are in link_link table, but are not in WISLR link shapefile attribute table. The un-joined records are manually evaluated and necessary modifications were made.

3.3.2.4 STNstart and STNend check

In the link_link table, all STN links should start at 0 (zero) and end at the full length of that link. This means that STNstart of the first segment (record in link_link table) of an STN link should be 0 (zero) and the STNend of the last segment (record in link_link table) of an STN link should be the full length of that STN link. This check is done only on current records.

This check involves separating the current records (Record_Historic is Null) from the entire link_link table, classifying the records based on STNid, and checking the lowest STNstart as well as highest STNend of each link. If the lowest STNstart of a link is nonzero, the STNstart is wrong. If the highest STNend of a link is not the full length of that STN link, STNend is wrong. Two different tables (STNstart wrong and STNend wrong) are created programmatically (APPENDIX B) for records having wrong STNstart and wrong STNend respectively. After identifying the records with inconsistencies (STNstart wrong and STNend wrong table) necessary changes were made.

3.3.2.5 STN continuity check

Every link in the link_link table should be continuous, which means that there should not be any gap between two segments along a link. A segment of a STN link either starts from zero or the end point of the previous segment of associated STN link.

The first step in the STN continuity check involves sorting the current records in the link_link table based on link ID and then sorted each STN link based on the STNstart. The second step is to identify any gaps in the link measurement. The logic used to find discontinuous links is that, the start point along an STN link is not equal to the end point of previous record for that STN link. This step is done for every segment of a STN link, but the first segment of that STN link. An STN link having only one segment cannot be discontinuous. The third step is to create a table (named STN discontinuous) with any identified discontinuous STN links. Having the list of discontinuous STN links, necessary corrections are made manually.

3.3.2.6 WISLRstart and WISLRend Check

WISLRstart and WISLRend checking is done in the same way as STNstart and STNend checking are done. In this case, the WISLR attribute table is used. In very few cases, nonzero WISLRstart and less-than-full-length of link WISLRend are acceptable. This happens when an STN link does not represent the full length of a WISLR link.

3.3.2.7 WISLR continuity check

The procedure involved sorting the records based on WISLRid and then sorting each link based on WISLRstart. Continuity of a WISLR link exists when the WISLRstart of a segment is the same as the WISLRend of the previous segment. A WISLR link having only one segment cannot be discontinuous. The very first segment of WISLR link should be zero (except few special cases), that were manually checked.

The WISLR continuity check does not work for turn lanes, median crossovers and waysides. In turn lanes, the same segment of WISLR link is used to represent two STN links. In this case, the WISLRstart and WISLRend are the same for two records in the link_link table, but

the record is marked with a T for furn lane. So if the records are found discontinuous, and any flag column is marked, those links are manually checked.

3.3.2.8 Date columns check

The Start_Valid and End_Valid date columns are populated programmatically and there could be some inconsistencies such as Start_Valid is older than End_Valid or Record_Historic is younger than Start_Valid. Checks are done to identify these types of inconsistency. Four queries that are used to identify records having potentially wrong dates. These are: (1) selecting records having End_Valid younger than Start_Valid; (2) selecting records having Record_Historic younger than Start_Valid; (3) selecting records having End_Valid, but not having Record_Historic; and (4) selecting records having Record_Created before Start_Valid. Any record selected from these four queries is manually inspected for errors.

3.4 Data Translation

Crash data are collected based on WISLR network but not all analysis tools used by WisDOT is using STN and yet to compatible with WISLR. Therefore field data collected on WISLR need to be translated on STN. At present this translation is done manually by assigning an STNid and offset for each crash. After analyzed, to put the data on cartographic presentation data need to be moved back on WISLR. WisDOT provided crash data in 2011 with reference to STN links and offset. In this research data on STN links are translated to WISLR and then moved data on WISLR are translated back to STN. Moved data is checked if the data lands on reasonable positions. Data translation is acceptable from higher resolution STN to lower resolution WISLR with the help of Point moving program developed through the early phases of the research. Data translation in reverse direction (from WISLR to STN) using same point moving program works except for those places where resolution differences occur. In these

locations points can land to multiple locations in STN, instead of one location when points are translate from WISLR to STN. Some rules were proposed by Graettinger et. al ,2013 to handle the ambiguity in data movement from WISLR to STN. Following these rules, and incorporating new options to make the data translation flexible to accommodate ambiguities, a new "Point moving program (WISLR to STN)" was developed. The places where the data movement ambiguity occurs are: 1) Median crossover, 2) Turn lane, 3) Wayside and 4) links having inconsistency in cartographic alignment.

3.4.1 Data translation ambiguity on median crossover/intersection:

Resolution difference between STN and WISLR representation is the cause of data translation ambiguity at intersections. An example is shown in Figure 3.2. The aerial image of the intersection is shown in Figure 3.2 (a). Here an intersection of two divided highways with four STN sites and four STN link (red lines). On the other hand this intersection is expressed by only one WISLR site (node) in WISLR network (blue lines). Four events 1-4 are shown on four STN links, which are moved to the WISLR network (Figure 3.2(b)). All four event points land on the one single intersecting point in WISLR. In the next step, when four events in WISLR are moved back to STN, each point has four potential places to land. In addition, the final STN point locations do not match the original point locations (Figure 3.2 (c)). According to the link_link table, any events at the intersecting point in WISLR network can move to any of the STN sites due to ambiguity in the resolution between these two systems..

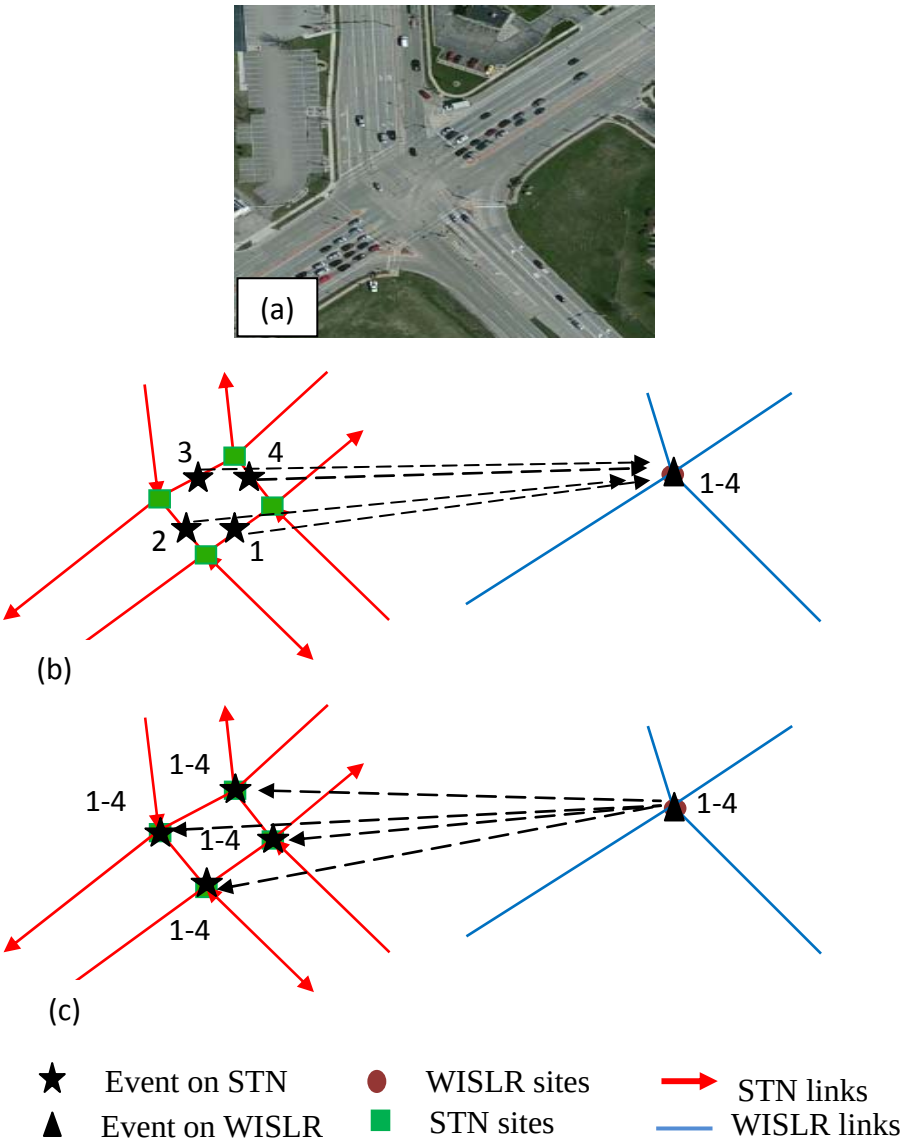


Figure 3.2 Data point translation between STN and WISLR at median cross over: (a) aerial image of an intersection (b) moving point from higher resolution STN to lower resolution WISLR, four data point (1-4) move to one single point; (c) moving point from WISLR to STN, each of four data point move to every node of STN links in intersection.

3.4.2 Data translation ambiguity at turn lanes

Data translation ambiguity at turn lanes is also due to resolution difference between STN and WISLR. An example of turn lane data transition ambiguity is shown in Figure 3.3. The aerial image of a turn lane is shown in Figure 3.3(a). In Figure 3.3, two separate STN links (red lines) are used to express a turn lane. On the other hand, only one WISLR link (blue line) expresses the same turn lane. Two points 1-2 on two separate STN links are moved to the associate single WISLR link when the points are moved from STN to WILSR as shown in Figure 3.3(b). Then, when points 1-2 are moved back from WISLR to STN, each point has two places to land on different STN link as shown in Figure 3.3(c). Though the points on WISLR are moved to their original locations in STN, additional possible locations are available which create ambiguity as there is no way to find the true location for the STN point on the WISLR network. The only information in link_link table about turn lane is that the record has a flag in T column, and that two STN links land on the same section of a WISLR link.

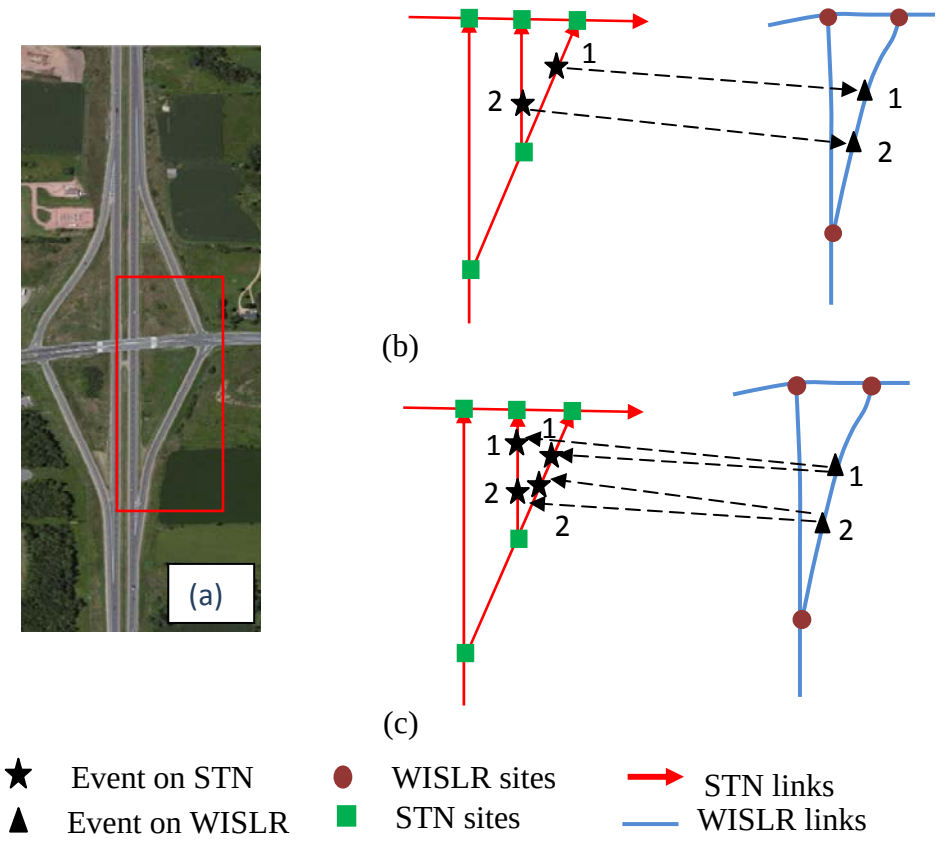


Figure 3.3 Example of data translation in turn lane (a) aerial image of a turn lane marked, (b) data movement from STN to WISLR and there is no ambiguity, (c) each point on WISLR has two places to land on STN and ambiguity initiates

3.4.3 Data translation ambiguity at wayside

Waysides are another place where ambiguity occurs during data movement between STN and WISLR. Multiple STN links are used to express the wayside (i,e weight station, rest area, etc); whereas only one point on WISLR indicates the wayside. Aerial photo of a wayside is shown in Figure 3.4(a). Three STN links (red lines) are used to express the wayside, whereas only one point on WISLR represents the wayside. Three point events, 1-3 on STN links, are moved to one point on WISLR according to the link_link table relationship which is shown in Figure 3.4(b). Every point on the wayside has the option to land at any of four end points on the

STN links when the points are moved from WISLR to STN network which is shown in Figure 3.4(c). As a result events in WISLR do not have a single location in STN which causes ambiguities. This location inconsistency during movement of data from WISLR to STN is due to resolution difference between these two LRSs at ways

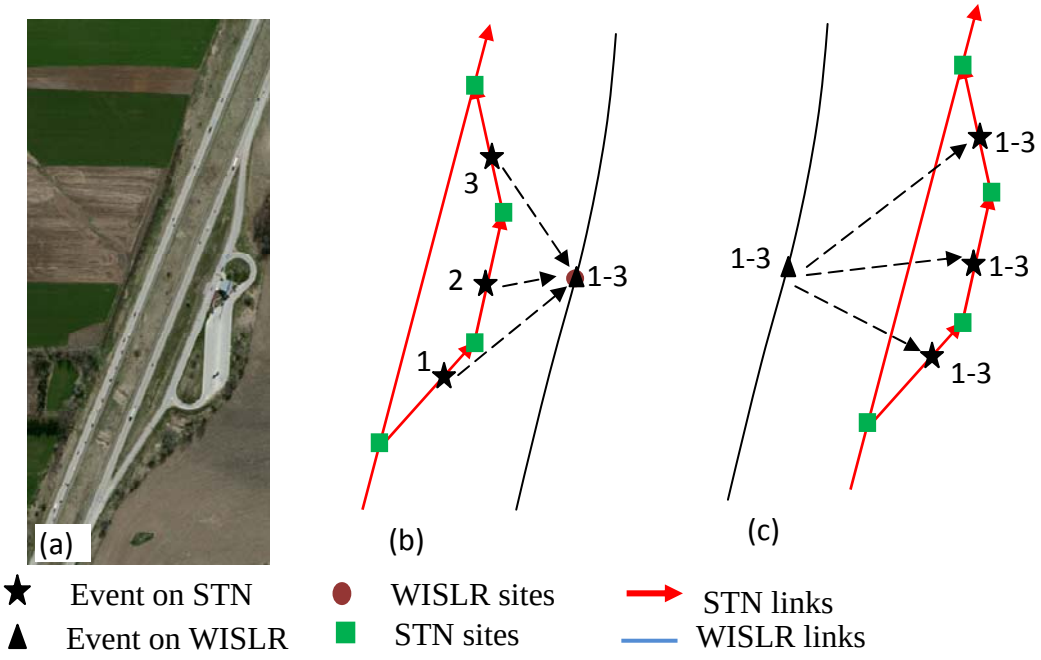


Figure 3.4 Example of data translation at wayside, (a) aerial image of a wayside, (b) data point 1-3 movement from STN to WISLR, (c) data movement 1-3 from WISLR to STN and every point 1-3 moves to every STN sites and ambiguity occurs.

3.4.4 Data translation rules and options

Improving the resolution in WISLR, up to the resolution of STN, is the most effective way to resolve the problem of data translation ambiguity, though extensive effort and time would be required. On the other hand, regular updating of the relationship between the two LRSs is also required to translate data in an acceptable manner. Instead of upgrading the resolution level of WISLR, two approaches are presented that help resolve the data translation issues. Firstly improve the information during data collection. Firstly the crash reporting form, crash reporting

method, etc. can be modified to collect more information about the position of crash. Implementation of this modification is prohibited due to time, resource and institutional constraints (Graettinger et.al., 2013).

In order to avoid the institutional format changes, some rules and options were developed and tested the flag column in the link_link table to reduce the data translation ambiguity, or at a minimum mark the data points having ambiguous transition. A tool was developed to implement the rules and options in user controlled way. The script of the tool, which is called 'Point Moving Program (WISLR to STN)' is included in APPENDIX C. The user interface of the program is shown in Figure 3.5

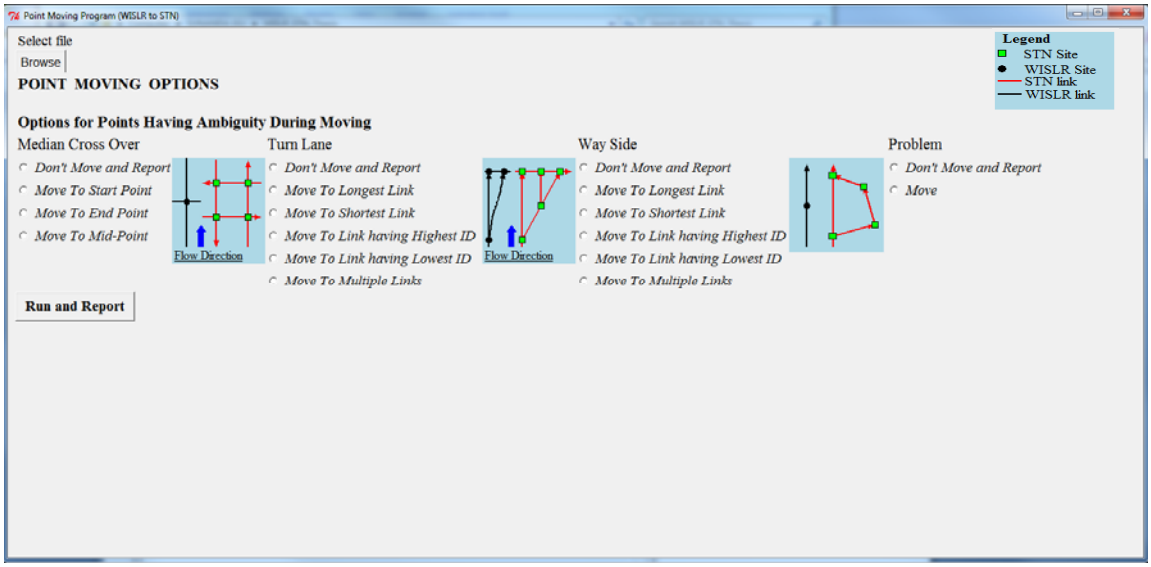


Figure 3.5 User interface of point moving program(WISLR to STN) before running

3.4.4.1 Median cross over rules

Ambiguity in data translation at median crossover happens when data is moved from lower resolution (WISLR) to higher resolution (STN) system. Four rules are applied to control ambiguity: (1) Data on WISLR would not be moved from WISLR to STN but be reported,

(2) Data on WILSR would be moved to the start point of the associated STN, (3) Data on WILSR would be moved to the end point of the associated STN, (4) Data on WISLR would be moved to the midpoint of the associated STN link. Figure 3.6 shows the options in 'Point Moving program (WISLR to STN) for median cross over data movement.

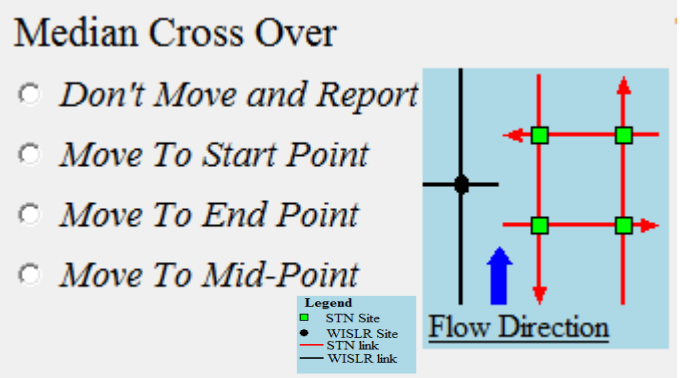


Figure 3.6 Data movement (WISLR to STN) options for median crossover

3.4.4.2 Turn lane rules

At turn lanes one point on WISLR could move to two places on two STN links which causes ambiguities. Six options are incorporated to control data movement ambiguity: (1) Data on WISLR would not be moved and be reported, (2) Data on WISLR would be moved to the associated STN link having longest length, (3) Data on WISLR would be moved to the associated STN link having shortest length, (4) Data on WISLR would be moved to the associated STN link having highest STNid, (5) Data on WISLR would be moved to the associated STN link having lowest STNid, (6) Data on WISLR would be moved to all possible places on the associated STN links. In the options 2 and 3, data would be moved to the STN link having lowest STNid if the associated STN links (2 links) are of equal length. The options in 'Point Moving program (WISLR to STN) for median crossover data movement is shown in Figure 3.7.

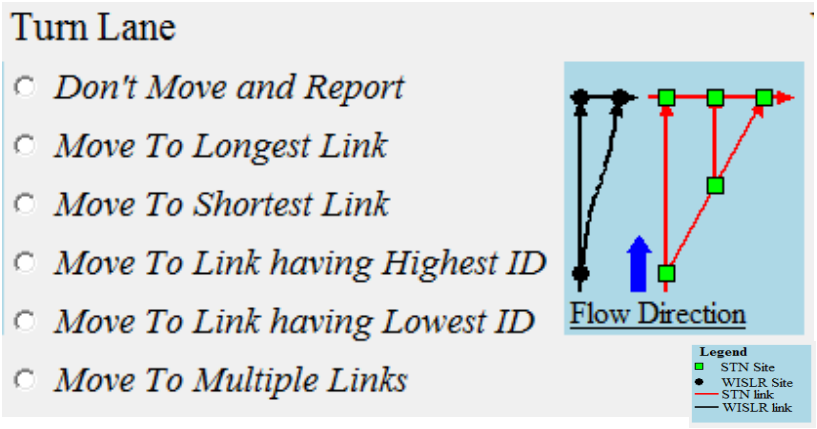


Figure 3.7 Data movement (WISLR to STN) options for turn lane

3.4.4.3 Wayside Rules

At way side, one point on WISLR can move to multiple places on multiple STN links; therefore, ambiguities occur. Six options are incorporated to control data movement and ambiguity (1) Data on WISLR would not be moved but would be reported, (2) Data on WISLR would be moved to the associated STN link having longest length, (3) Data on WISLR would be moved to the associated STN link having shortest length, (4) Data on WISLR would be moved to the associated STN link having highest STNid, (5) Data on WISLR would be moved to the associated STN link having lowest STNid, (6) Data on WISLR would be moved to all possible places on the associated STN links. In the options 2 and 3, data would be moved to the STN link having lowest STNid if the associated STN links are equal in length. The options in 'Point Moving program (WISLR to STN)' for way side data movement are shown in Figure 3.8.

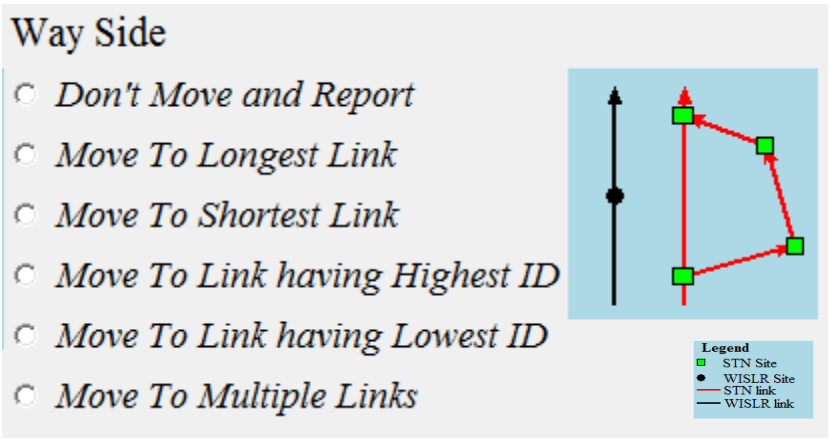


Figure 3.8 Data movement (WISLR to STN) options for way side

3.4.4.4 Problem Rules:

Links having inconsistency in cartographic representation or dissimilarity between line work and database records or any other inconsistency not expressible by a flag column are included in the link_link table with a flag in Problem (P) column. It is expected that points/events on these links would not transfer properly. There are two options for data on links having a flag in the P column in link_link table – (1) Data on WISLR would not be moved, but reported or (2) Data on WISLR would be moved in accordance with the ratio to the possible place on STN. The data movement option where STN-WISLR relationship inconsistency exists are shown in Figure 3.9.

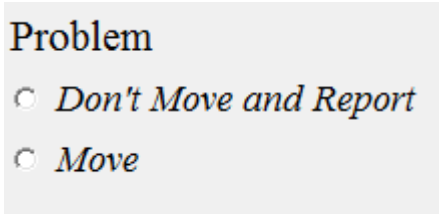


Figure 3.9 Data movement (WISLR to STN) options for problem flaged records in link_link table

3.5 Implementation of data translation rules and options

Two sets of data were used to test the WISLR to STN program. The first set of data was every hundredth mile point on every STN link in Dane County. These data were generated with the previously developed 'STN point generator' program (Ryals, 2011). The data points were moved from STN to WISLR and then moved back to STN. The returned data on STN was then compared with the original data on STN.

The second data set was crash points from 2011 provided by WisDOT. The crash data was moved from STN to WISLR and then back to STN. A report was generated containing the percentage of data that move to median crossovers, turn lanes, waysides and problem links.

3.6 Conclusion

The QA/QC approaches developed, programmed and tested in this research save time and impove accuracy ot the link_link table. Implementation of proposed rules to move data from WISLR to STN controls ambiguities during data translation from lower resolution network (WISLR) to higher resolution network (STN).

CHAPTER 4

APPLICATION OF the link_link TABLE

4.1 Introduction

The link_link table was developed as a functional merge between theSTN and WISLR systems to establish a relationship that allows data to translate from one system to the another. Beside this, link_link table has the capability for additional DOT activities. This chapter details how the link_link table can be used to provide names to the all ramps along of Wisconsin state routes. This improves map readability and crash data collection because ramp fro to names are understandable to the offices in the field. All ramps are called ‘ramp’ in WISLR. WISLR is used in car map by police officers in Wisconsin. WISLR link ids are employed behind the scene to read data, but street names assist officers. Under these circumstances, if a crash happens on a ramp it is difficult for the police officer to identify the exact ramp in the car map because all ramps have common name ‘ramp’ rather than specific name for a specific ramp. To improve the situation WisDOT seeks to find a way to provide names for ramps, and the ramp name should be updated when changes to the connecting road names occur. Capability as well as flexibility of link_link table provides the ability to name the ramps and also update the ramp names with the updated link_link table in the future. In this research an algorithm is developed to provide names to ramps and accommodate future road name changes. A script following the algorithm was written in Python. The input to this script is the attribute table of WISLR links, the attribute table for STN links, the STN route link table, and the link_link table. The output is a table

containing the WISLRid of each Ramp, WISLRids of connection roads of ramp, and ramp name in a specific format.

4.2 Methodology

Ramp name is produced from the two connecting roads. The format of ramp name is Ramp_<from road name>_to_<to road name>. The 'from road name' and 'to road name' indicate the roads from where a ramp starts and to where a ramp ends, respectively. Figure 4.1 shows an example of ramp name format.

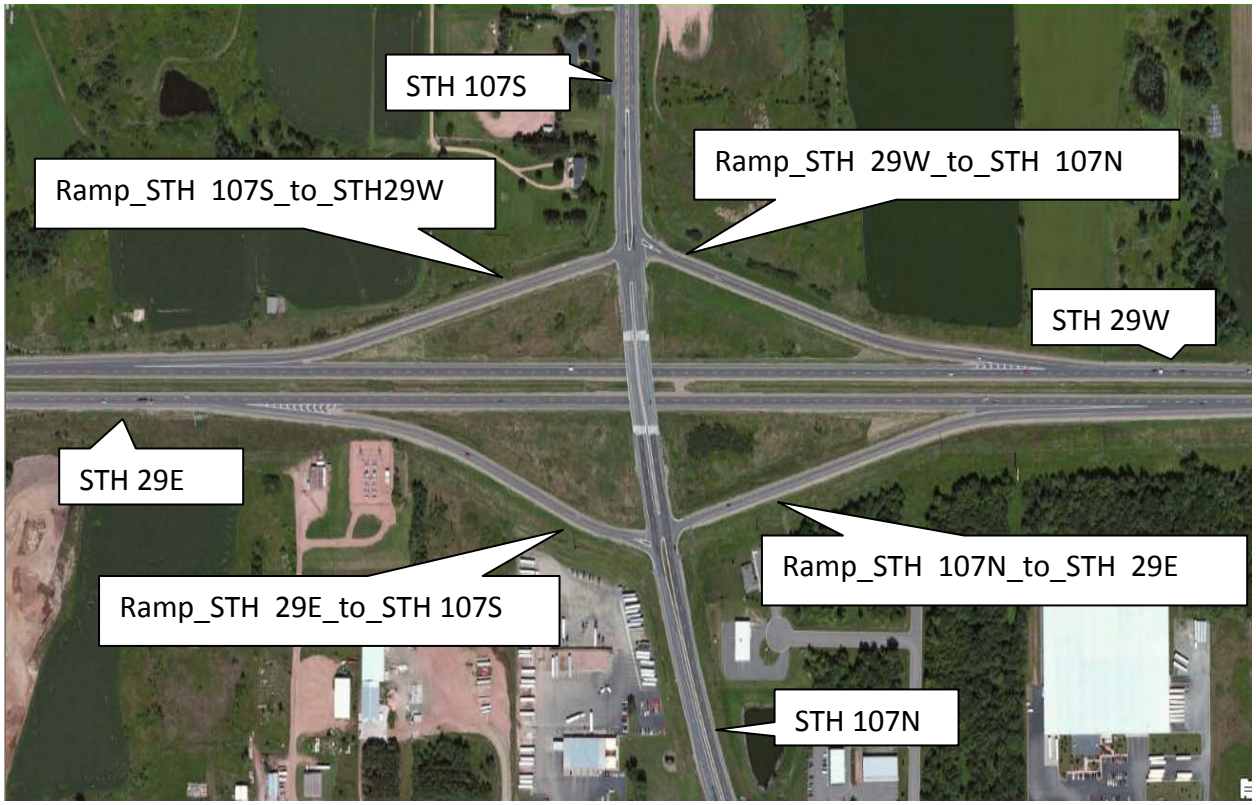


Figure 4.1 Example of ramp name format

The 'from road' and 'to road' names match the STN name if available, otherwise the names are taken from WISLR. STN names are given preference over WISLR names because the STN network provides the most updated and specific names associated with state routes. For

example, in WISLR, the road name does not reflect the directional orientation of the road. Two WISLR links running on top of each another in opposite directions have the same name even though the associated STN links have two different names following the directional initials (i,e E, W, N, S). Figure 4.2 shows an aerial image of on and off ramps. STN representation of the ramps and the WISLR representation of ramps are shown in Figure 4.2(b) and Figure 4.2(c) respectively. Ramps are called 'OFF' in STN while ramps are called 'Ramp' in WISLR.

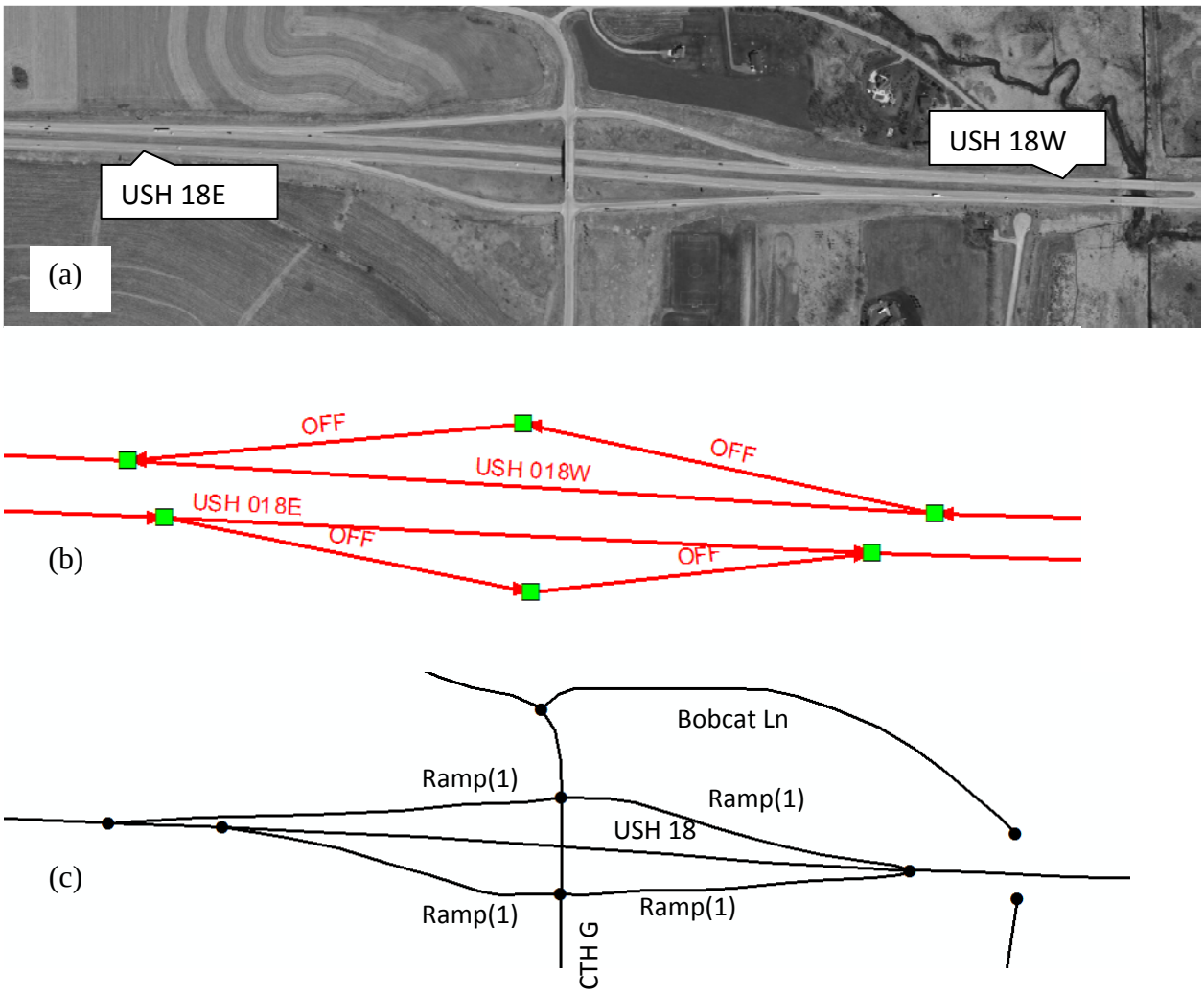


Figure 4.2 Ramp representation (a) Aerial image of ramps and surrounding area, (b) STN representation of ramps and names, (c) WISLR representation of ramps and names

Two sides of the highway in two directions, are represented with two STN links in Figure 4.2(b) and are called USH 018E and USH 018W respectively. Figure 4.2(c) shows the WISLR representation of same roads. In Figure 4.2(c) the two WISLR links representing the two highways (USH 18E and USH 18W) in opposite directions are running on top of each another in opposite directions. As shown in Figure 4.2(c) ramps are called 'Ramp(1)', and the name of highway is USH 18 regardless of the direction.

The ramp name table produced as a part of this work contains five columns. Table 4.1 shows the layout of ramp table. There is a flag column to indicate inconsistency in the associated ramp name process. These inconsistencies may be related to STN-WISLR relationships, road names, databases etc.

Table 4.1 Ramp table format

WISLRid	**WISLRid_F**	**WISLRid_T**	**RampName**	**Flag**
Unique WISLRid of Ramps	WISLRid of the link from where the WISLR link of ramp starts	WISLRid of the link to where the WISLR link of ramp ends	Ramp_<from road>_to_<to road>	1 in Flag column indicates inconsistency.

In the ramp name procedure initially list of links that represent ramps must be identified in STN and WISLR. After identifying the ramp a 'ramp name' needs to be produced. Finally a QA/QC procedure is performed.

4.3 Ramp Identification

A list of ramps was identified using STN links, and a table of ramp STNids as well as associated WISLR link ids was created. This table was called Ramp_STN as STN is used as a primary source to identify these ramp links. Another list of WISLRids representing ramps was obtained using WISLR data is created. This table was called Ramp_WISLR.

Different approaches were followed to produce names for ramp depending on if the link was found with STN data (STN ramp) or WISLR data (WISLR ramp). For Ramp_STN links, STN data and link_link table were used. For Ramp_WISLR links, only WISLR data was used.

4.4 Name of ramp identified through STN

Ramp link identified from STN data were named through a systematic process. Table 4.2 shows the data base tables and columns associated with tables that were utilized to identify ramps and to provide name to the ramps.

Table 4.2 List of tables and associated columns used form naming ramp identified using STN

Tables	**Used Columns and Description**				
DT_RDWY_LINK (STN link table)	RDWY_LINK_ID (Unique STN link id)		REF_SITE_FROM (Unique start site id)	REF_SITE_TO (Unique end site id)	LCM_STUS (Status of the link)
DT_RDWY_RTE_LINK (STN route link table)	RDWY_LINK_ID (Unique STN link id)		SYS_DSGT_TY (Primary road name)	RTE_TY_NB_DIR (Directional road name)	LCM_STUS (Status of the route link relationship)
link_link (functional merge between STN and WISLR)	STNid (Unique STN id)	WISLRid (Unique WISLR id)	STNstart (Start point of a STN link/link segment)	STNend (End point of a STN link/link segment)	Record_Historic (Last date when STN-WISLR relationship valid)

DT_RDWY_LINK table provides information about STN links. In DT_RDWY_LINK table RDWY_LINK_ID column provides the unique id for every STN link which is called

STNid; REF_SITE_FROM columns provide the STN node/ site id from where a STN link starts; REF_SITE_TO column provides the STN node/site id where a STN link ends; and LCM_STUS provides the date from when a link becomes historic. For current link LCM_STUS column is null. DT_RDWY_RTE_LINK table provide the route information and relationship between routes and STN links. DT_RDWY_RTE_LINK table, RDWY_LINK_ID is the STNid; SYS_DSGT_TY column provides the primary road name (i.e. STH 30, IH 20 etc); RTE_TY_NB_DIR column provides the directional information (i.e. E, W, N, S) and LCM_STUS tells the date from when an STN link- route relationship becomes historic. For link_link table the columns are as described in chapter 2, section 2.4.

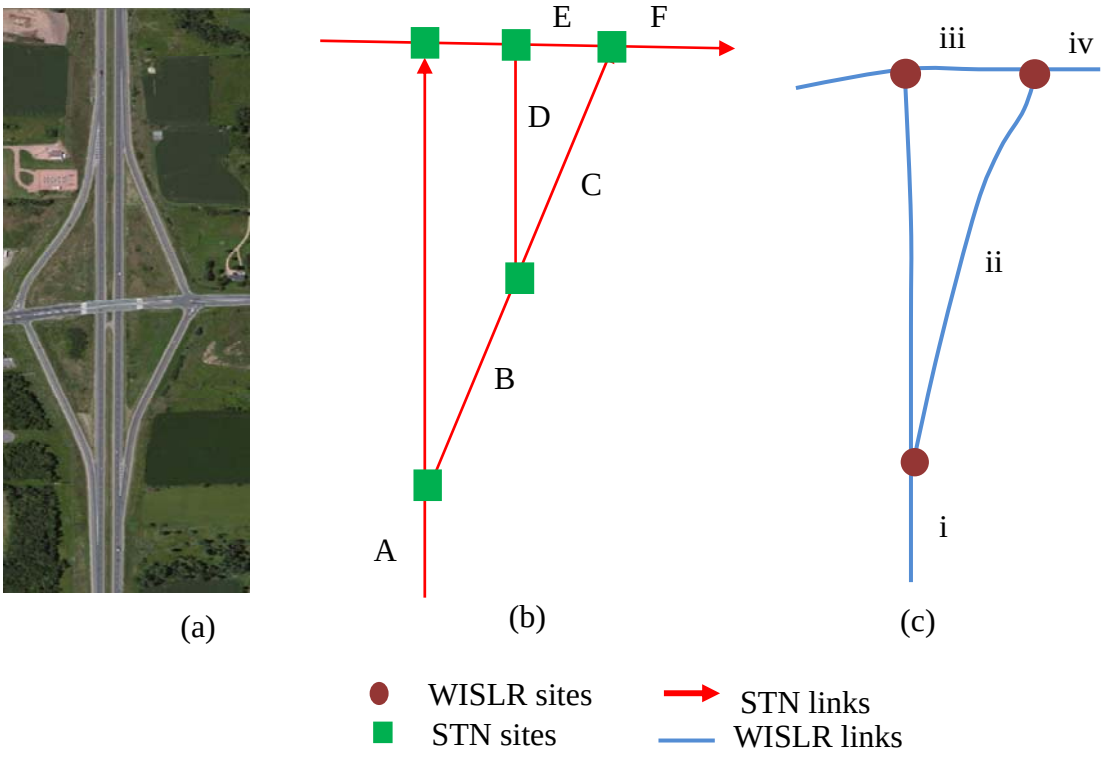


Figure 4.3 Example of generating ramp name

An example of generating ramp name is shown in Figure 4.3. An aerial image of an intersection is shown in Figure 4.3(a). The ramp in the rectangle in Figure 4.3(a), is represented

with STN and WISLR in Figure 4.3(b) and Figure 4.3 (c) respectively. The red lines are STN links and the blue lines are WISLR links. The green squares are STN sites and the violet dots are WISLR sites. STN links B, C and D represent the ramp. The start sites of STN link B and the end site of STN link A is same. So, the 'from road' of STN link B is STN link A and associated WISLR id is 'i' which is shown as WISLRid_F in ramp table. Now the end site of STN link B is the same as the start site of STN link C or D. But STN link C and D are the part of the ramp. Therefore STN link C or D can not be the to road of link B. The 'to road' for B link is the 'to road' of C or D link. So the 'to road's for B, C, and D are STN link E and F. The 'to road' WISLRids are iii and iv, which are the associated WISLRid of STN link E and F respectively. In ramp table smallest WISLRid is considered in case of multiple 'form road' and 'to road' WISLRids. Therefore, in ramp table WISLR_T is iii which is shown in Tbale 4.3. The name of STN link E and F is same in most of the cases in STN link. Now using link_link table the WISLR link associated with the ramp shown in Figure 4.3 is 'ii'. So the ramp table looks like the Table 4.3 for the ramp shown in Figure 4.3.

Table 4.3 An example of ramp name generation

WISLRid	WISLRid_F	WISLRid_T	RampName	Flag
ii	i	iii	Ramp_<Name of STN A>_to_<Name of STN E>	

A flow chart of the STN based ramp naming is shown in Figure 4.4. The algorithm starts with STN route link table. If the primary name of a route is 'OFF', the STN link associated with that route is a ramp. Ramp WISLRids associated with the Ramp STN ids are obtained using link_link table. STN table provides the STN link starts and ends information.

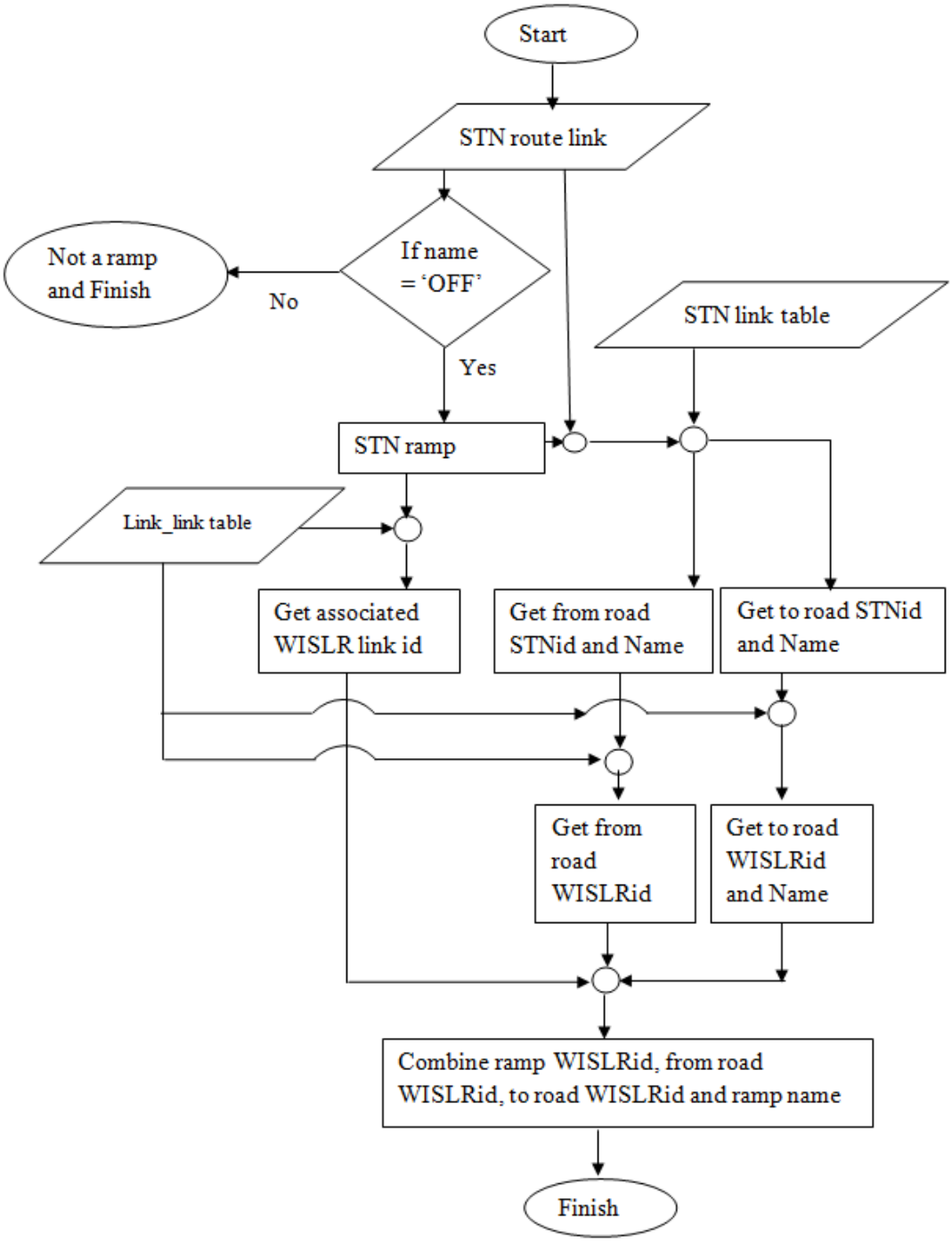


Figure 4.4 Flow chart of naming procedure of ramp identified using STN links

No

Using STN ramp list, STN link table and STN route link table ‘from road’ name and STNid as well as ‘to road’ name and STNid are obtained. The logic behind getting ‘from STN link’ is that, ‘from STN link’ of a ramp ends at the STN site from where the associated ramp starts. The logic behind getting ‘to STN link’ is that, ‘to STN link’ starts from the site where the associated ramp STN link ends. Using link_link table ‘from road’ and ‘to road’ WISLRid are obtained from ‘from road’ STNid and ‘to road’ STNid respectively. A ramp table is generated in expected format by combining ramp WISLRid, ‘form road’ WISLRid, ‘to road’ WISLRid and ramp name. Detail queries are in the Python script in APPENDIX D.

4.5 Name of ramp identified through WISLR

The data base tables and columns associated with ramp naming using WISLR links are shown in Table 4.4. WISLR link information can be obtained from DT_RDWY_LINK table. In DT_RDWY_LINK table RDWY_LINK_ID column provide the unique WISLR link ids;

Table 4.4 List of tables and associated columns used form naming ramp identified using WISLR

Tables	**Used Columns and Description**			
DT_RDWY_LINK (WISLR link table)	RDWY_LINK_ID (Unique STN link id)	REF_SITE_FROM_ID (Unique start site id)	REF_SITE_TO_ID (Unique end site id)	LCM_STUS_TYCD (Status of the link)
DT_ST_PRTE_OVLY_LINE (WISLR name table)	RDWY_LINK_ID (Unique WISLR link id)	ST_PRMY_SYMB_TY (Primary road name)	ST_LABL_NM (Local road name)	

REF_SITE_FROM_ID column provides the WISLR site/node ids from where a WISLR link starts; REF_SITE_TO_ID column provides the WISLR site/node ids where a WISLR link ends; and LCM_STUS_TYCD tells the date from when a WISLR link becomes historic. In the table DT_ST_PRTE_OVLY_LINE, RDWY_LINK_ID column tells the unique WISLR ids of

WISLR links; ST_PRMY_SYMB_TY column tells the type of name (i.e. STH, IH, USH); and ST_LABL_NM column tells state route number (i.e. 20, 66) or the local road name (i.e. Millar road).

A flow chart of the WISLR based ramp naming is shown in Figure 4.5. The procedure starts with WISLR name table (DT_ST_PRTE_OVLY_LINE). If the primary road name of a WISLR link starts with 'Ramp', that link is considered as ramp. Now if a WISLR link whose name is started with 'Ramp' is identified and if this WISLR link is not in the ramp list, which was created, using STN links as primary source (section 4.4); the WISLR link is considered to name using WISLR links as a primary source. Utilizing WISLR link table (DT_RDWY_LINK) and WISLR name table 'from road' WISLRid and name as well as 'to road' WISLRid and name are generated. The logic behind getting 'from road' information is that, 'from road' WISLR link ends at the WISLR site, from where the associated WISLR ramp starts. The logic behind getting 'to road' information is that, 'to road' WISLR link starts from the WISLR site, where the associated WISLR ramp ends. Finally ramp WISLRid, from road WISLRid, to road WISLR id and ramp name are combined into a ramp name table in desired format. Details queries can be found in APPENDIX D.

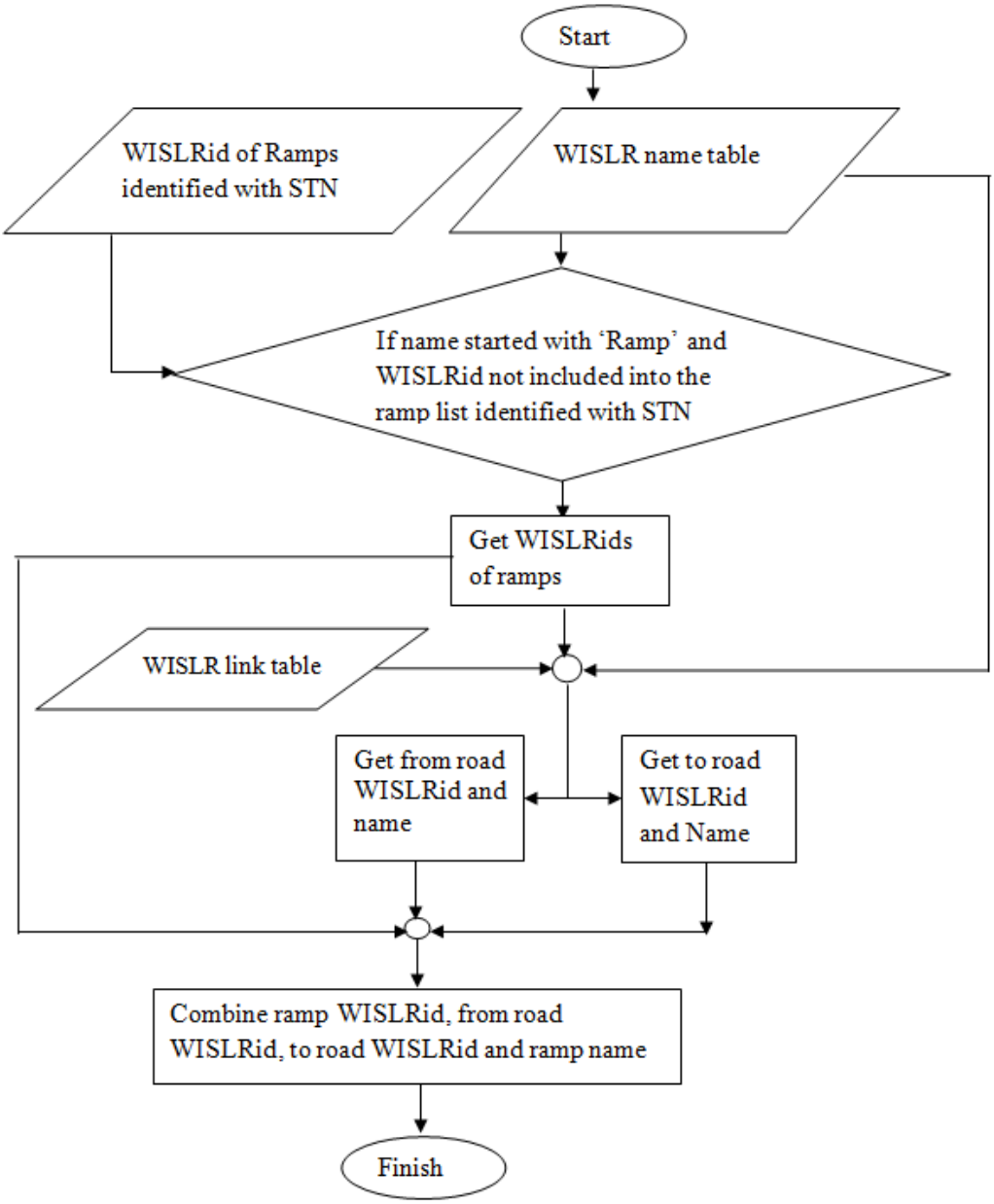


Figure 4.5 Algorithm of naming procedure of ramp identified using WISLR links

4.6 Ramp name QA/QC

The ramp name QA/QC was done county by county and started by visual checking from one end of a state route to the other end of that route. The QA/QC procedure was a manual process. The ramp name table was joined to the WISLR shapefile and the joined links were exported as a different shapefile shows only WISLR ramp links that were found following the algorithms stated in the previous sections. Then, STN links, STN sites, WISLR links, WISLR sites, and the ramp shapefile for a particular county are extracted. The ramp shapefile is given different symbology/color from the WISLR links to facilitate visual inspection. The STN shapefile is also displayed over the WISLR shapefile to show the STN names of road. The base map helps to improve visual inspection facility. An example is shown in figure 4.6.

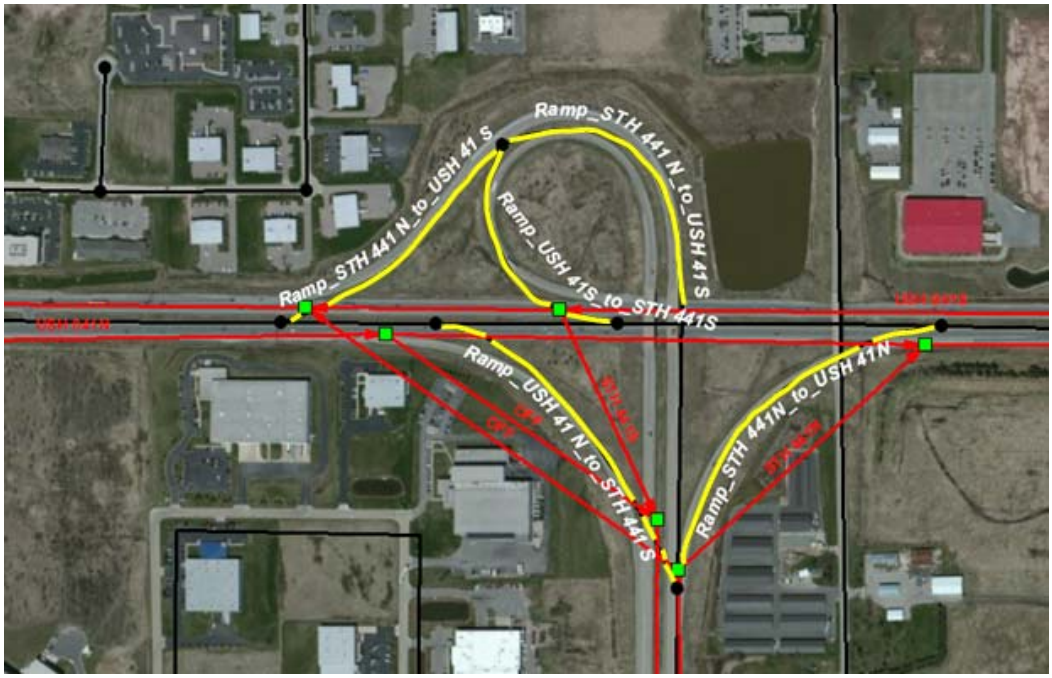


Figure 4.6 Ramp with names in WISLR. Black lines represent WISLR links, yellow lines represent ramps and red lines represent STN links. Black dots are WISLR sites and green squares are STN sites.

A visual checking is then performed from one end of a state route to the other end of that state route. During checking three things were done: (1) non ramp WISLRids are deleted from ramp list; (2) the "from name/WISLRid" and "to name/WISLRid" of a ramp are changed if needed; and (3) ramps that were not automatically included are manually added in the ramp list.

4.7 Results

The accuracy of ramp naming largely depends on ramp identification. Approximately 80% of the ramps were identified through STN link filtering, 19% of the ramps were identified using WISLR link filtering, and 1% of the ramps were identified during visual checking. The breakdown of ramp identification statistics is shown in Figure 4.7.

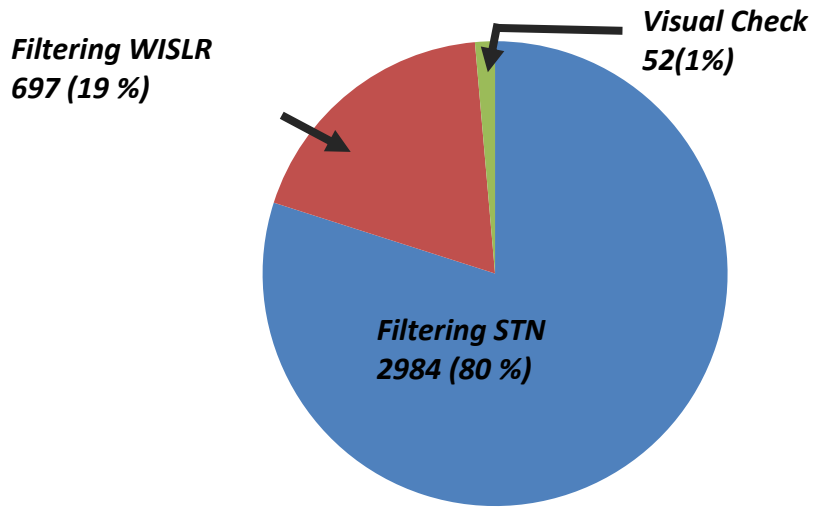


Figure 4.7 Ramp identification pie chart (Total ramps : 3733)

Approximately 50% of the ramps were given a ramp name using only STN data, and the link_link table; 38% of the ramps were given names using STN, WISLR, and the link_link table

data; and only 12% of the ramps were given name using only WISLR data. A pie chart for the giving of ramp names is shown in Figure 4.8.

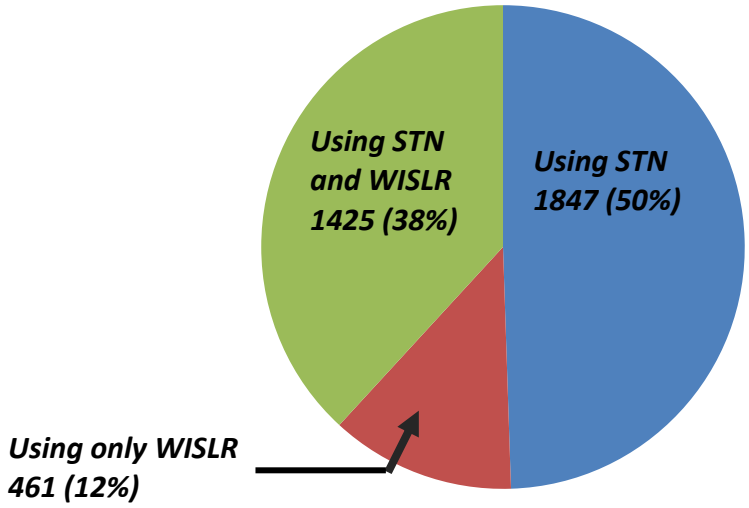


Figure 4.8 Ramp naming pie chart (Total ramps : 3733)

4.8 Conclusion

Naming of ramp in WISLR required extensive use of the link_link table as well as both LRSs used at WisDOT. Updating ramp names to reflect any future name changes is automatically included in this process. This algorithm is capable of updating ramp names to accommodate the road name changes dynamically. Providing ramp names to WISLR links increase the readability of the in car map, as names can be used to distinguish one ramp from another in the field.

CHAPTER 5

RESULTS

5.1 Introduction

This chapter provides the results of this research with regards to the link_link table updating and crash data location moving from STN to WISLR system. Results after implementing the new rules and options for data translation from WISLR to STN are also discussed in this chapter.

5.2 link_link table update results

The link_link table was updated with 2011 data from the previous 2010 data. The 2010 link_link table contains 71,290 records while the 2011 link_link table contains 77,072 records. In 2011, 5,398 records which were current in 2010 link_link table became historic, and 5,782 new records were entered. Changes were observed in 2,234 STN links as well as in 5,332 WISLR links. Usually a deleted/historic link is replaced by a new link.

5.3 Crash data translation from STN to WISLR results

There were 53,449 crash points on STN links in 2011 in Wisconsin state. All crash points were moved from STN to WISLR. Among the 53,449 crash points, 52,604 (98.41%) crash points were moved to current WISLR links and 845 (1.59%) crash points moved to historic WISLR links. Current WISLR links associated with current STN links have a valid relationship. Because STN and WISLR are updated independently at WisDOT, a current STN link may be

associated with a historic WISLR link. Because the WISLR link has not yet been updated, this would cause a point to move to a historic WISLR link.

5.4 Data translation from WISLR to STN results

Two sets of data were used to test the new 'Point Moving Program (WISLR to STN)'. The first data set were the points on every hundredth mile along STN links in Dane county. The second data set were the crash points on Wisconsin state routes provided by WisDOT. The crash data set was provided with reference to STNid and offset.

Points on STN links in Dane county were chosen because Dane county has the large number of STN-WISLR relationships and a cross section of inconsistency and flags. All points on STN links were moved to WISLR. Then the points on WISLR were moved back to STN. A comparison was made between the moved points on STN with the original points on STN. The 86,082 points on STN moved acceptably to WISLR with previously developed point moving program (Ryals, 2011).

Using the new 'Point Moving Program (WISLR to STN)', options can be chosen to control the ambiguous point movement. In the case of moving the same point multiple times to the same location but on different STN links (end of one STN link and start of another STN link might be same), the smaller STN id is chosen as a default. Using this program, every point on WISLR that does not move back to its original starting position on STN, (in this case 10.89% of total points), could be controlled and reported. The report shown in Figure 5.1 is for the 100^{th} mile points that were moved from STN to WILSR and then back to STN.

According to the report, 100% of the data points on WISLR were moved; among which 0.29 % were associated with median crossover, 0.4% were associated with turn lanes and 0.08%

were associated with waysides. As a result, ambiguity occurred for only 0.77% of total points that were moved. The remaining 99.23% of the points, including points on problem links, moved to their original position in STN.

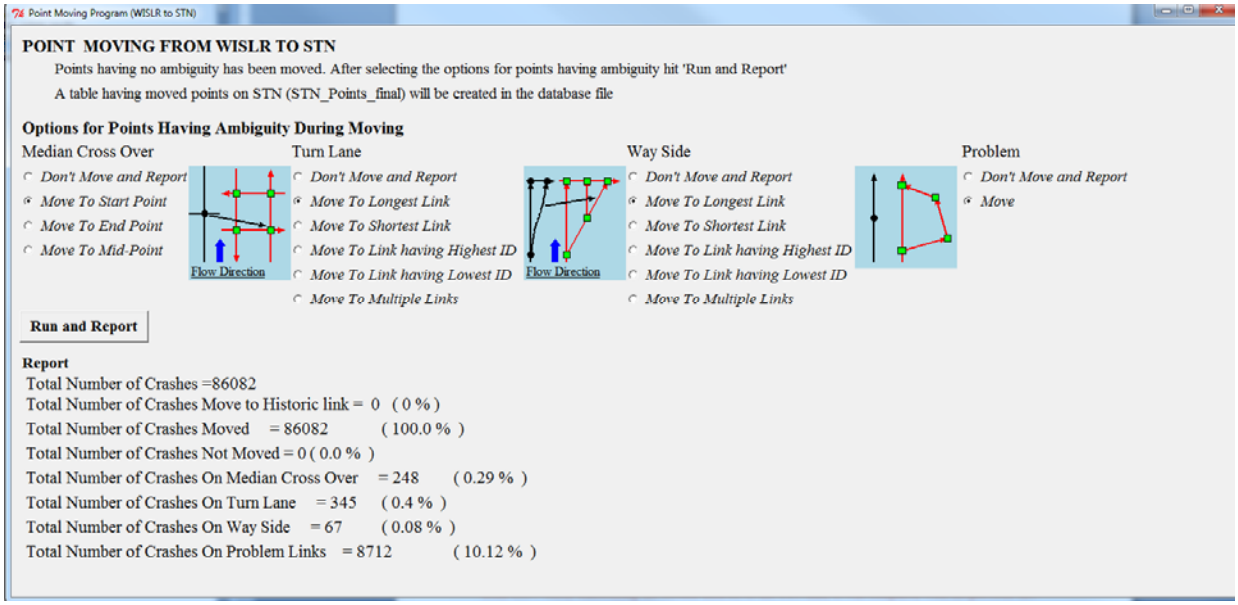


Figure 5.1 Point moving options from WISLR to STN and report after running the program using every hundredth mile point on STN of Dane county, WI

A total of 53,449 state route crashes were moved from STN to WISLR. When moving these crashes back from WISLR to STN, using the new 'Point Moving Program (WISLR to STN)', 3.88% of crash points moved to a potentially ambiguous location, but the location was controlled by the program. The results and report of the test using crash data for 2011 is shown in Figure 5.2.

From the report it is shown that 96.125% crash points, including crash points on problem links, landed in their original position in STN. Only 3.875 % of the crash points occurred at ambiguous location when moving from WISLR to STN. Thus locations are controlled by user input and also reported in error log files.

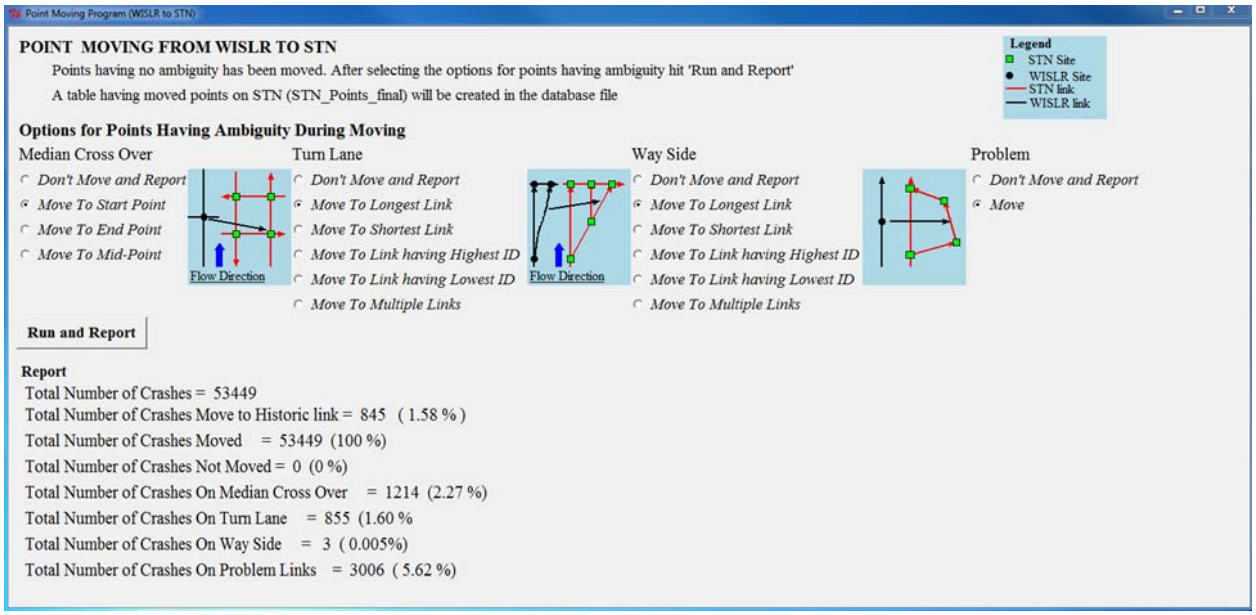


Figure 5.2 Point moving options from WISLR to STN and report after running the program using RP crash data provided by WisDOT

5.5 Conclusion

In this research, the link_link table was successfully updated and all state route crash points on STN were successfully translated to WISLR. Improving the resolution or modifying the data collection format would help data translation from WISLR to STN by reducing ambiguity. Both of the options are time as well as resource consuming. Moreover the result shows that following the methodology described in this thesis, data can be translated from WISLR to STN and ambiguities can be controlled and reported.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

This research was done to: (1) improve the updating and QA/QC procedure of the link_link table, (2) move 2011 state route crashes from STN to WISLR, (3) implement rules and options for controlling ambiguities during data translation from lower resolution WISLR to higher resolution STN, and (4) provide ramp names on state routes jin WISLR.

Procedures were established and tested for updating the link_link table. Two columns of link_link table data, Record_Created and Record_Historic were populated programmatically. County wide QA/QC procedure were developed, program, and successfully tested. Additional statewide QA/QC procedures were incorporated in the process to remove duplicate records and to check STN/WISLR continuity as well as the dates in link_link table. An additional tool was developed to facilitate statewide QA/QC automatically. An updated link_link table, along with state route crash data on WISLR links was delivered to WisDOT. All crashes from 2011 were moved successfully from STN to WISLR. This research also investigated methods to move data from lower resolution WISLR network to higher resolution STN network. A new point moving program, which is flexible enough to accommodate ambiguities in a user controlled fashion during data translation from lower resolution WISLR to higher resolution STN was developed and tested successfully.

An algorithm was also successfully developed that generates names to the all ramps in Wisconsin. A Python script was written that uses STN as well as WISLR line work to generate

ramp name. Visual checking was done on ramp names to determine that the procedure was successful.

6.2 Future Work

The link_link table requires yearly updating to accommodate changes in the STN and WISLR networks. Many aspects of the updating procedure are manual operations and prone to errors. Any automation in the link_link table updating procedure is expected to save time and resources.

Data translation at gore points is acceptable, though not perfect, as gore points in STN and gore points in WISLR do not represent the same location. To accommodate this inconsistency the gore point column (G) in link_link table is marked. Efforts should be made to improve the data translation at gore points.

Leveraging the link_link table for additional data needs at WisDOT should be investigated. In this research, the link_link table was used to provide names to ramps in addition to crash data translation. The flexibility of the link_link table allows this table to be utilized where STN and WISLR need to be used together to improve analysis and display.

REFERENCES

Graettinger, A., Morrison, A. L., Parker, S., Forde, S., & Qin, X. (2013). Translating Transportation Data Between Dissimilar-Resolution Linear Referencing Systems. *Transportation Research Board 92nd Annual Meeting (No. 13-2817).*

Graettinger AJ, Qin X, Spear G, Parker ST, and Forde S. Combining State Route and Local Road Linear Referencing System Information. *Journal of the Transportation Research Record. 2009;2121:152-159*

Ryals ZT. A Technique for Merging State and Non-State Linear Referencing Systems. 2011.

Morrison, A. L. (2012). Updating and Expanding a Multi Resolution Linear Referencing System Functional Merge

NCHRP. Highway Location Reference Methods, Systhesis of Highway Practice 21, *National Academy of Sciences, Washington, DC, 1974*

Viggen Corporation. Location Referencing Systems: *Analysis of Current Methods Applied ot IVHS User Services, Boston, MA, 1994*

P. Okunieff, D. Siegel, and Q. Miao. " Location referencing methods for intelligent transportation systems (ITS) user services recommended approach," *1995 GIS-T Proceedings, 57-75, AASHTO, Washington, DC, 1995*

A.P. Vonderohe, Results of a Workshop on A Generic Data Model for Linear Referencing Systems, *Department of Civil and Environmental Engineering, University of Wisconsin, Madison, 1995*

P Scarponcini, Generalized Model for Linear Referencing in Transportation, *GeoInformatica 6:1, 35-55, 2002, 2002 Kluwer Academic Publishers.*

Graettinger, A. J., Qin, X., Spear, G., Parker, S. T., & Forde, S. (2008). State and non-state network mapping integration. *Proceedings of the 2008 Mid-Continent Transportation Research Forum, Madison, Wisconsin.*

APPENDIX A

```
import time
import pyodbc

print 'TABLE CREATION'

DBfile = 'State.mdb'
conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};DBQ='+DBfile)
cur = conn.cursor()

if cur.tables(table = 'Date_Table').fetchone():
    string_d1 = "DROP TABLE Date_Table"
    cur.execute(string_d1)

string_1 = "CREATE TABLE Date_Table(STNid integer, STNstart integer, STNend
integer,\
    WISLRid integer, WISLRstart integer, WISLRend integer,\
    T string, M string, G string, W string, P string,Coder string,\
    Record_Created string, Record_Historic string, Start_Valid string, S_S
string, S_W string,End_Valid string, \
     E_S string, E_W string, Comments string, Reviewer_Comments string,
Date_Reviewed string, County string)"
cur.execute(string_1)
conn.commit()

if cur.tables(table = 'Date_Table_update').fetchone():
    string_d1 = "DROP TABLE Date_Table_update"
    cur.execute(string_d1)

string_2 = "CREATE TABLE Date_Table_update(STNid integer, STNstart integer,
STNend integer,\
    WISLRid integer, WISLRstart integer, WISLRend integer,\
    T string, M string, G string, W string, P string,Coder string,\
    Record_Created string, Record_Historic string, Start_Valid string,
End_Valid string, \
    Comments string, Reviewer_Comments string, Date_Reviewed string, County
string)"


cur.execute(string_2)
conn.commit()

#================================================================================
================================================
# STN DATE POPULATION
#================================================================================
================================================
sql_s = 'SELECT link_link.STNid, link_link.WISLRid,
STN_Date.MIN_DT_RTE_LINK_CURR, STN_Date.MAX_LCM_DT_HSTL\
```

```
        FROM link_link LEFT JOIN STN_Date  ON link_link.STNid =
STN_Date.RDWY_LINK_ID ORDER BY link_link.STNid, link_link.WISLRid'
IDs = cur.execute(sql_s)

X=[]
for r in IDs:
    X.append(r)


i =0
while i < len(X):
    if X[i][2] == None:
        X[i][2] = ''
    if X[i][3] == None:
        X[i][3] = ''
    i = i + 1


#=============================================================================
=================================================
#WISLR DATE POPULATION
#=============================================================================
=================================================
sql_w = 'SELECT link_link.STNid, link_link.WISLRid,
WISLR_Date.MIN_LCM_CURR_DT, WISLR_Date.MAX_LCM_HSTL_DT\
        FROM link_link LEFT JOIN WISLR_Date ON link_link.WISLRid =
WISLR_Date.RDWY_LINK_ID ORDER BY link_link.STNid, link_link.WISLRid'
IDw = cur.execute(sql_w)
Y = []
for r in IDw:
    Y.append(r)

i =0
while i < len(Y):
    if Y[i][2] == None:
        Y[i][2] = ''
    if Y[i][3] == None:
        Y[i][3] = ''
    i = i + 1

# Inserting Date into Date_Table having all date comes from STN and WISLR
date table
#=============================================================================
=================================================
string = 'SELECT * FROM link_link ORDER BY STNid, WISLRid'
IDS = cur.execute(string)
A =[]
for r in IDS:
    A.append(r)

i =0
while i < len(A):
    if A[i][16] == None:    # Comments
        A[i][16] = ''

    A[i][14] = ''
    A[i][15] = ''
```

```
    i = i + 1
#print 'A=', A

i= 0
while i <len (A):

    sql_all = " INSERT INTO Date_Table(STNid, STNstart, STNend, WISLRid,
WISLRstart, WISLRend,\
                T, M, G, W, P, Coder, Record_Created, Record_Historic,
Start_Valid, S_S, S_W,End_Valid,\
                  E_S, E_W, Comments, Reviewer_Comments, Date_Reviewed,
County)\
           VALUES ('%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s',
'%s', '%s', '%s', '%s', '%s',\
                    '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s',
'%s')"\
            % (A[i][0], A[i][1], A[i][2], A[i][3], A[i][4], A[i][5],
A[i][6], A[i][7], A[i][8], A[i][9], A[i][10],\
               A[i][11], A[i][12], A[i][13], A[i][14],  X[i][2], Y[i][2],
A[i][15], X[i][3], Y[i][3], A[i][16], A[i][17],  A[i][18],  A[i][19] )



    cur.execute (sql_all)
    conn.commit()
    i = i + 1

sql1 = "ALTER TABLE Date_Table ALTER COLUMN S_S date  "
sql2 = "ALTER TABLE Date_Table ALTER COLUMN S_W date"
sql3 = "ALTER TABLE Date_Table ALTER COLUMN Start_Valid date"
sql4 = "ALTER TABLE Date_Table ALTER COLUMN E_S date  "
sql5 = "ALTER TABLE Date_Table ALTER COLUMN E_W date"
sql6 = "ALTER TABLE Date_Table ALTER COLUMN End_Valid date"
cur.execute(sql1)
cur.execute(sql2)
cur.execute(sql3)
cur.execute(sql4)
cur.execute(sql5)
cur.execute(sql6)
conn.commit()

# Population of Start_Valid and End_Valid
#=============================================================================
=============================================================
string_current = 'SELECT * FROM Date_Table'
Current = cur.execute(string_current)
#conn.commit()
C = []
for r in Current:
    C.append(r)
C = [list (member) for member in C]

i = 0
while i < len(C):
    if C[i][15] == None and C[i][16] != None:
        C[i][14] = C[i][16]
```

```
    if C[i][15] != None and C[i][16] == None:
        C[i][14] = C[i][15]
    if C[i][15] != None and C[i][16] != None:
        if C[i][15] > C[i][16]:
            C[i][14] = C[i][15]
        else:
            C[i][14] = C[i][16]
    #print 'C[i][14] = ',C[i][0], ' = ', C[i][14]
    if C[i][14] == None:
        C[i][14] = ''
    #print 'C[i][14] = ',C[i][0], ' = ', C[i][14]
#-------------------------------------------------------------------------------
-------------------------------------------------
#For End_Valid Population
    if C[i][18] == None and C[i][19] != None:
        C[i][17] = C[i][19]
    if C[i][18] != None and C[i][19] == None:
        C[i][17] = C[i][18]
    if C[i][18] != None and C[i][19] != None:
        if C[i][18] > C[i][19]:
            C[i][17] = C[i][19]
        else:
            C[i][17] = C[i][18]
    #print 'C[i][17] = ',C[i][0], ' = ', C[i][17]
    if C[i][17] == None or C[i][17] =='None':
        C[i][17] = ''
    #if C[i][17] == 'None':
        #C[i][17] = ''
    if C[i][13] == 'None' or  C[i][13] == None:
        C[i][13]=''
    if C[i][12] =='None' or C[i][12] ==None:
        C[i][12]=''
    #print 'C[i][17] = ',C[i][0], '=', C[i][3] ,' = ', C[i][17], C[i][21]

    sql_all = "INSERT INTO Date_Table_update(STNid, STNstart, STNend,
WISLRid, WISLRstart, WISLRend,\
                T, M, G, W, P, Coder, Record_Created, Record_Historic,
Start_Valid, End_Valid,\
                Comments, Reviewer_Comments, Date_Reviewed, County)\
             VALUES ('%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s',
'%s', '%s', '%s', '%s', '%s','\
                      '%s', '%s', '%s', '%s', '%s', '%s')"\
              % (C[i][0], C[i][1], C[i][2], C[i][3], C[i][4], C[i][5],
C[i][6], C[i][7], C[i][8], C[i][9], C[i][10],\
                C[i][11], C[i][12], C[i][13], C[i][14], C[i][17], C[i][20],
C[i][21],  C[i][22],  C[i][23] )

    cur.execute (sql_all)
    conn.commit()
    i = i + 1
sql_c = "ALTER TABLE Date_Table_update ALTER COLUMN Start_Valid date  "
sql_h = "ALTER TABLE Date_Table_update ALTER COLUMN End_Valid date  "
cur.execute(sql_c)
cur.execute(sql_h)
conn.commit()
```

```
print 'Step 4.  All dates are populated into link_link table and named as
Date_Table_Updated'
cur.close()
conn.close()
```

APPENDIX B

```
import time
import pyodbc

print 'TABLE CREATION'
print '=============================================================================='
# Connect to database file
#=============================================================================
#DBfile = raw_input ('Enter the DB File (.mdb file)'
DBfile = raw_input ('Enter Link Link .mdb file (table name must be link_link)
>>>   ')
conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};DBQ='+DBfile)
cursor = conn.cursor()

#=============================================================================
==============================
# Table Creation
#=============================================================================
==============================
# 1 # Current Link Table

if cursor.tables(table = 'Current_Link_Table').fetchone():
    sql_d1 =    "DROP TABLE Current_Link_Table"
    cursor.execute(sql_d1)

sql_1 = "   CREATE TABLE    Current_Link_Table (STNid integer, STNstart
integer, STNend integer,\
                            WISLRid integer, WISLRstart integer, WISLRend
integer,\
                            T string,M string, G string, W string, P
string,Coder string,\
                            Record_Created Date, Record_Historic string,
Start_Valid Date, End_Valid string,\
                            Comments string, Reviewer_Comments string,
Date_Reviewed string, County string)"

cursor.execute(sql_1)
conn.commit()
print ('1. Current_Link_Table created')
#-----------------------------------------------------------------------------
------------------------------
# 2 # Historic Link Table

if cursor.tables(table = 'Historic_Link_Table').fetchone():
    sql_d2 =    "DROP TABLE Historic_Link_Table"
    cursor.execute(sql_d2)

sql_2 = "   CREATE TABLE    Historic_Link_Table (STNid integer, STNstart
integer, STNend integer,\
```

```
                            WISLRid integer, WISLRstart integer, WISLRend
integer,\
                            T string,M string, G string, W string, P
string,Coder string,\
                            Record_Created Date, Record_Historic string,
Start_Valid Date, End_Valid string,\
                            Comments string, Reviewer_Comments string,
Date_Reviewed string, County string)"

cursor.execute(sql_2)
conn.commit()
print ('2. Historic_Link_Table created')

# 3  # STN_Wrong_Start_Table table

if cursor.tables(table = 'STN_Wrong_Start_Table').fetchone():
    #raw_input ('STN_Wrong_Start_Table is existed.  For deleting press
enter')
    sql_d3 = "DROP TABLE STN_Wrong_Start_Table"
    cursor.execute(sql_d3)

sql_3 = "   CREATE TABLE    STN_Wrong_Start_Table (STNid integer, STNstart
integer, STNend integer,\
                            WISLRid integer, WISLRstart integer, WISLRend
integer,\
                            T string,M string, G string, W string, P
string,Coder string,\
                            Record_Created Date, Record_Historic string,
Start_Valid Date, End_Valid string,\
                            Comments string, Reviewer_Comments string,
Date_Reviewed string, County string)"

cursor.execute(sql_3)
conn.commit()
print ('3. STN_Wrong_Start_Table Table created')

#-------------------------------------------------------------------------
# 4 # STN_Discontinuity_Table table

if cursor.tables(table = 'STN_Discontinuity_Table').fetchone():
    #raw_input ('STN_Wrong_Start_Table is existed.  For deleting press
enter')
    cursor.execute ('DROP TABLE STN_Discontinuity_Table')


cursor.execute ("   CREATE TABLE    STN_Discontinuity_Table (STNid integer,
STNstart integer, STNend integer,\
                                    WISLRid integer, WISLRstart integer,
WISLRend integer,\
                                    T string,M string, G string, W string, P
string,Coder string,\
                                    Record_Created Date, Record_Historic
string, Start_Valid Date, End_Valid string,\
                                    Comments string, Reviewer_Comments
string, Date_Reviewed string, County string)")

#cursor.execute(sql_3)
```

```
conn.commit()
print ('4. STN_Discontinuity_Table Table created')
#
#================================================================================
#Insert values into the tables
#================================================================================
print 'TABLE POPULATION'

# 1  # Populate Current_Link_Table table

Insert_1 = 'INSERT INTO Current_Link_Table(STNid, STNstart, STNend,WISLRid,
WISLRstart, WISLRend,\
                T , M, G, W, P,Coder,Record_Created, Record_Historic,
Start_Valid, End_Valid,\
                Comments, Reviewer_Comments, Date_Reviewed, County)\
            SELECT * FROM link_link\
            WHERE Record_Historic is NULL\
            ORDER BY STNid, STNstart, STNend'

cursor.execute(Insert_1)
conn.commit()
print '1. Current_Link_Table has been populated'
#--------------------------------------------------------------------------------

# 2  # Populate Historic_Link_Table

Insert_2 = 'INSERT INTO Historic_Link_Table(STNid, STNstart, STNend,WISLRid,
WISLRstart, WISLRend,\
                T , M, G, W, P,Coder,Record_Created, Record_Historic,
Start_Valid, End_Valid,\
                Comments, Reviewer_Comments, Date_Reviewed, County)\
            SELECT * FROM link_link\
            WHERE Record_Historic is NOT NULL\
            ORDER BY STNid, STNstart, STNend'

cursor.execute(Insert_2)
conn.commit()
print '2. Historic_Link_Table has been populated'
#--------------------------------------------------------------------------------
# 3  # Populate STN_Wrong_Start_Table table

import_3    =   'SELECT * FROM link_link WHERE Record_Historic Is Null ORDER
BY STNid, STNstart, STNend, WISLRid '
IDS         =   cursor.execute(import_3)
A           =   [r for r in IDS]

# Condition
i = 0
if A[i][1] != 0 :
    Insert_3_0 =    "INSERT INTO  STN_Wrong_Start_Table(STNid, STNstart,
STNend, WISLRid, WISLRstart, WISLRend,\
                                T, M, G, W, P, Coder, Record_Created,
Record_Historic, Start_Valid, End_Valid,\
                                Comments, Reviewer_Comments, Date_Reviewed,
County)\
                    VALUES      ('%s', '%s', '%s', '%s', '%s', '%s', '%s',
'%s', '%s', '%s', '%s', '%s', '%s', '%s',\
```

```
                                '%s', '%s', '%s', '%s', '%s', '%s')"\
                                 % (A[i][0], A[i][1], A[i][2], A[i][3],
A[i][4], A[i][5], A[i][6] if A[i][6] else '',\
                                    A[i][7]  if A[i][7]  else '', A[i][8]  if
A[i][8] else '', A[i][9] if A[i][9] else '',\
                                    A[i][10] if A[i][10] else '', A[i][11] if
A[i][11] else '', A[i][12] if A[i][12] else '',\
                                    A[i][13] if A[i][13] else '', A[i][14] if
A[i][14] else '',  A[i][15] if A[i][15] else '' ,\
                                    A[i][16] if A[i][16] else '', A[i][17] if
A[i][17] else '',  A[i][18] if A[i][18] else '',\
                                    A[i][19] if A[i][19] else '' )

    cursor.execute(Insert_3_0)
    conn.commit()
i = 1
while i < len(A):
    if (A[i-1][0]) != (A[i][0]):
        if A[i][1]!=0:
            Insert_3 = "INSERT INTO  STN_Wrong_Start_Table(STNid, STNstart,
STNend, WISLRid, WISLRstart, WISLRend,\
                               T, M, G, W, P, Coder, Record_Created,
Record_Historic, Start_Valid, End_Valid,\
                               Comments, Reviewer_Comments, Date_Reviewed,
County)\
                        VALUES  ('%s', '%s', '%s', '%s', '%s', '%s', '%s',
'%s', '%s', '%s', '%s', '%s', '%s', '%s',\
                                '%s', '%s', '%s', '%s', '%s', '%s')"\
                                 % (A[i][0], A[i][1], A[i][2], A[i][3],
A[i][4], A[i][5], A[i][6] if A[i][6] else '',\
                                    A[i][7]  if A[i][7]  else '', A[i][8]  if
A[i][8] else '', A[i][9] if A[i][9] else '',\
                                    A[i][10] if A[i][10] else '', A[i][11] if
A[i][11] else '', A[i][12] if A[i][12] else '',\
                                    A[i][13] if A[i][13] else '', A[i][14] if
A[i][14] else '',  A[i][15] if A[i][15] else '' ,\
                                    A[i][16] if A[i][16] else '', A[i][17] if
A[i][17] else '',  A[i][18] if A[i][18] else '',\
                                    A[i][19] if A[i][19] else '' )

            cursor.execute (Insert_3)
            conn.commit()
    i = i + 1


sql_3_1 = "ALTER TABLE STN_Wrong_Start_Table ALTER COLUMN Record_Historic
date"
sql_3_2 = "ALTER TABLE STN_Wrong_Start_Table ALTER COLUMN End_Valid date"
cursor.execute(sql_3_1)
cursor.execute(sql_3_2)
conn.commit
print '3. STN_Wrong_Start_Table table is populated'
#-------------------------------------------------------------------------
# 4  # Populate STN_Discontinuity_table table
record = cursor.execute ( 'SELECT * FROM link_link WHERE Record_Historic Is
Null ORDER BY STNid, STNstart, STNend, WISLRid')
B =[r for r in record]
```

```
i = 1
while i < len(B):
    if B[i-1][0] == B[i][0]:
        if B[i][1]!=B[i-1][2]:

            '''cursor.execute('INSERT INTO STN_Discontinuity_Table(STNid,
STNstart, STNend, WISLRid, WISLRstart, WISLRend,\
                                        T, M, G, W, P, Coder, Record_Created,
Record_Historic, Start_Valid, End_Valid,\
                                        Comments, Reviewer_Comments,
Date_Reviewed, County)\
                                VALUES      ("%s", "%s", "%s", "%s", "%s", "%s",
"%s", "%s", "%s", "%s", "%s", "%s", "%s", "%s",\
                                        "%s", "%s", "%s", "%s", "%s", "%s")'\
                                    % (B[i][0], B[i][1], B[i][2], B[i][3],
B[i][4], B[i][5], B[i][6] if B[i][6] else '',\
                                        B[i][7]  if B[i][7]  else '', B[i][8]
if B[i][8] else '', B[i][9] if B[i][9] else '',\
                                        B[i][10] if B[i][10] else '',
B[i][11] if B[i][11] else '', B[i][12] if B[i][12] else '',\
                                        B[i][13] if B[i][13] else '',
B[i][14] if B[i][14] else '', B[i][15] if B[i][15] else '' ,\
                                        B[i][16] if B[i][16] else '',
B[i][17] if B[i][17] else '',  B[i][18] if B[i][18] else '',\
                                        B[i][19] if B[i][19] else '' ))'''
            cursor.execute("INSERT INTO STN_Discontinuity_Table(STNid,
STNstart, STNend, WISLRid, WISLRstart, WISLRend,\
                                        T, M, G, W, P, Coder, Record_Created,
Record_Historic, Start_Valid, End_Valid,\
                                        Comments, Reviewer_Comments,
Date_Reviewed, County)\
                                VALUES      ('%s', '%s', '%s', '%s', '%s', '%s',
'%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s',\
                                        '%s', '%s', '%s', '%s', '%s', '%s')"\
                                    % (B[i][0], B[i][1], B[i][2], B[i][3],
B[i][4], B[i][5], B[i][6] if B[i][6] else '',\
                                        B[i][7]  if B[i][7]  else '', B[i][8]
if B[i][8] else '', B[i][9] if B[i][9] else '',\
                                        B[i][10] if B[i][10] else '',
B[i][11] if B[i][11] else '', B[i][12] if B[i][12] else '',\
                                        B[i][13] if B[i][13] else '',
B[i][14] if B[i][14] else '',  B[i][15] if B[i][15] else '' ,\
                                        B[i][16] if B[i][16] else '',
B[i][17] if B[i][17] else '',  B[i][18] if B[i][18] else '',\
                                        B[i][19] if B[i][19] else '' ))
            conn.commit()


    i = i + 1


sql1 = "ALTER TABLE STN_Discontinuity_Table ALTER COLUMN Record_Historic
date"
sql2 = "ALTER TABLE STN_Discontinuity_Table ALTER COLUMN End_Valid date"
cursor.execute(sql1)
cursor.execute(sql2)
```

```
conn.commit()
print '4. STN_Discontinuity_Table table is populated'

#-----------------------------------------
cursor.close()
conn.close()
```

APPENDIX C

```
import sys, math, pyodbc, Tkinter, tkMessageBox, tkColorChooser, tkFileDialog
from itertools import groupby
from operator import itemgetter
from Tkinter import *

class InterFace:
    def __init__(self, r):
        self.r = r
        r.title("Point Moving Program (WISLR to STN)")
        r.geometry ("1500x700")

        self.t1 = Label(self.r,text = 'Select file', font = ('Times', '14')
).place (x = 10, y = 5)
        self.b1 = Button( self.r, text = 'Browse', command = self.browse,
font = 14 ).place( x = 10, y = 30)
        self.options = Label(self.r,text = 'POINT  MOVING  OPTIONS', font =
('Times', '16', 'bold') ).place (x = 10, y = 60)
        self.run = Button(self.r, text = " Run and Report ",  command =
self.main, font = ('Times', '14', 'bold')).place(x = 10, y = 350)

        self.medianOptions()
        self.turnLaneOptions()
        self.waySideOptions()
        self.probOptions()
        self.r.mainloop()

    def browse(self):
        self.filepath = tkFileDialog.askopenfilename()

    def medianOptions(self):
        self.b = StringVar()
        self.b.set(None)
        Ambiguity_Option    = Label(self.r,text = 'Options for Points Having
Ambiguity During Moving', font = ('Times', '16', 'bold') ).place (x = 10, y =
110)
        Median_Option       = Label(self.r, text = "Median Cross Over", font
= ('Times', '16')).place(x = 10, y = 140)
        Median_No    = Radiobutton(self.r, text = "Don't Move and Report",
command = self.medianNo,    font = ('Times', '14', 'italic'), variable =
self.b, value = 3).place(x = 10, y = 170)
        Median_Start = Radiobutton(self.r, text = 'Move To Start Point',
command = self.medianStart, font = ('Times', '14', 'italic'), variable =
self.b, value = 4).place(x = 10, y = 200)
        Median_END   = Radiobutton(self.r, text = 'Move To End Point',
command = self.medianEnd,   font = ('Times', '14', 'italic'), variable =
self.b, value = 5).place(x = 10, y = 230)
        Median_MID   = Radiobutton(self.r, text = 'Move To Mid-Point',
command = self.medianMid,   font = ('Times', '14', 'italic'), variable =
self.b, value = 6).place(x = 10, y = 260)
```

```
        self.C_M = Canvas(self.r, height = 140, width = 125, background =
'lightblue')
        self.C_M.create_line(2,60,40,60, fill = 'black', width = 2)
        self.C_M.create_line(21,2,21,120, fill = 'black', width = 2)
        self.C_M.create_oval(17,55,25,64, fill = 'black')
        self.C_M.create_line(120,35,42,35, fill = 'red', width = 2, arrow =
'last')
        self.C_M.create_line(42,80,120,80, fill = 'red', width = 2, arrow =
'last')
        self.C_M.create_line(60,5,60,120, fill = 'red', width = 2, arrow =
'last')
        self.C_M.create_line(102,120,102,5, fill = 'red', width = 2, arrow =
'last')
        self.C_M.create_rectangle(56,76,64,84, fill = 'green')
        self.C_M.create_rectangle(56,31,64,39, fill = 'green')
        self.C_M.create_rectangle(98,31,106,39, fill = 'green')
        self.C_M.create_rectangle(98,76,106,84, fill = 'green')
        self.C_M.create_text(50,130, font = 'Times 12 underline', text =
'Flow Direction')
        self.C_M.create_line(40,120,40,90, fill = 'blue', width = 8, arrow =
'last')
        self.C_M.place (x=215, y=170)


    def medianNo (self):
        self.flag = 0
    def medianStart (self):
        self.C_M.create_line(26,62,94,74, fill = 'black', width = 2, arrow =
'last')
        self.flag = 1

    def medianEnd (self):
        self.B = self.C_M.create_line(26,56,94,38, fill = 'black', width = 2,
arrow = 'last')
        self.flag = 2

    def medianMid (self):
        self.C = self.C_M.create_line(26,56,94,56, fill = 'black', width = 2,
arrow = 'last')
        self.flag = 3
    def turnLaneOptions(self):
        self.c = StringVar()
        self.c.set(None)
        self.Turn_Lane_Option    = Label(self.r, text = "Turn Lane",  font =
('Times', '16')).place(x = 340, y = 140)
        self.Turn_Lane_No        = Radiobutton(self.r, text = "Don't Move and
Report",            command = self.turnNo,     font = ('Times', '14',
'italic'), variable = self.c, value = 7).place(x = 340, y = 170)
        self.button_T_Lane_Long  = Radiobutton(self.r, text = 'Move To
Longest Link',             command = self.turnLong,   font = ('Times', '14',
'italic'), variable = self.c, value = 8).place(x = 340, y = 200)
        self.button_T_Lane_Short = Radiobutton(self.r, text = 'Move To
Shortest Link',            command = self.turnShort,  font = ('Times', '14',
'italic'), variable = self.c, value = 9).place(x = 340, y = 230)
```

```
        self.button_T_Lane_High  = Radiobutton(self.r, text = 'Move To Link
having Highest ID',   command = self.turnHighid, font = ('Times', '14',
'italic'), variable = self.c, value = 10).place(x = 340, y = 260)
        self.button_T_Lane_Low   = Radiobutton(self.r, text = 'Move To Link
having Lowest ID',    command = self.turnLowid,  font = ('Times', '14',
'italic'), variable = self.c, value = 11).place(x = 340, y = 290)
        self.button_T_Lane_Multi = Radiobutton(self.r, text = 'Move To
Multiple Links',           command = self.turnMulti,  font = ('Times', '14',
'italic'), variable = self.c, value = 12).place(x = 340, y = 320)

        self.C_T = Canvas(self.r, height = 140, width = 125, background =
'lightblue')
        self.C_T.create_line(10,120,10,20,   fill = 'black', width = 2, arrow
= 'last')
        self.C_T.create_line(5,20,40,20,     fill = 'black', width = 2, arrow
= 'last')
        self.C_T.create_line(10,110, 15, 80, 16, 75, 25, 50, 30,20, smooth =
'true', fill = 'black', width = 2, arrow = 'last')

        self.C_T.create_oval(6,16,14,24, fill = 'black')
        self.C_T.create_oval(6,106,14,114, fill = 'black')
        self.C_T.create_oval(26,16,34,24, fill = 'black')

        self.C_T.create_line(45,20,120,20, fill = 'red', width = 2, arrow =
'last')
        self.C_T.create_line(54,120,54,20, fill = 'red', width = 2, arrow =
'last')
        self.C_T.create_line(54,110,104,20, fill = 'red', width = 2, arrow =
'last')
        self.C_T.create_line(79,65,79,16, fill = 'red', width = 2, arrow =
'last')

        self.C_T.create_rectangle(50,16,58,24, fill = 'green')
        self.C_T.create_rectangle(75,16,83,24, fill = 'green')
        self.C_T.create_rectangle(100,16,108,24, fill = 'green')
        self.C_T.create_rectangle(75,61,83,69, fill = 'green')
        self.C_T.create_rectangle(50,106,58,114, fill = 'green')

        self.C_T.create_text(50,130, font = 'Times 12 underline', text =
'Flow Direction')
        self.C_T.create_line(40,120,40,90, fill = 'blue', width = 8, arrow =
'last')

        self.C_T.place (x=625, y=170)

    def turnNo (self):
        self.T = 0

    def turnLong (self):
        self.C_T.create_line(28, 50,90,40 , fill = 'black', width = 2, arrow
= 'last')
        self.T = 1

    def turnShort (self):
        self.C_T.create_line(28, 50,77,40 , fill = 'black', width = 2, arrow
= 'last')
        self.T = 2
```

```
        #print self.flag

    def turnHighid (self):
        self.C_T.create_line(28, 50,90,40 , fill = 'black', width = 2, arrow
= 'last')
        self.T = 3

    def turnLowid (self):
        self.C_T.create_line(28, 50,77,40 , fill = 'black', width = 2, arrow
= 'last')
        self.T = 4

    def turnMulti (self):
        self.C_T.create_line(28, 50,90,40 , fill = 'black', width = 2, arrow
= 'last')
        self.C_T.create_line(28, 50,77,40 , fill = 'black', width = 2, arrow
= 'last')
        self.T = 5
    def waySideOptions(self):

        self.d = StringVar()
        self.d.set(None)
        Way_Side_Option     = Label(self.r, text = "Way Side", font =
('Times', '16')).place(x = 750, y = 140)
        Way_Side_No         = Radiobutton(self.r, text = "Don't Move and
Report",            command = self.wayNo,    font = ('Times', '14',
'italic'), variable = self.d, value = 13).place(x = 750, y = 170)
        button_W_Long       = Radiobutton(self.r, text = 'Move To Longest
Link',             command = self.wayLong,  font = ('Times', '14', 'italic'),
variable = self.d, value = 14).place(x = 750, y = 200)
        button_W_Short      = Radiobutton(self.r, text = 'Move To Shortest
Link',            command = self.wayShort, font = ('Times', '14', 'italic'),
variable = self.d, value = 15).place(x = 750, y = 230)
        button_W_High       = Radiobutton(self.r, text = 'Move To Link having
Highest ID',   command = self.wayHighid,font = ('Times', '14', 'italic'),
variable = self.d, value = 16).place(x = 750, y = 260)
        button_W_Long       = Radiobutton(self.r, text = 'Move To Link having
Lowest ID',    command = self.wayLowid, font = ('Times', '14', 'italic'),
variable = self.d, value = 17).place(x = 750, y = 290)
        button_W_Multi      = Radiobutton(self.r, text = 'Move To Multiple
Links',           command = self.wayMulti, font = ('Times', '14', 'italic'),
variable = self.d, value = 18).place(x = 750, y = 320)

        self.C_W = Canvas(self.r, height = 125, width = 125, background =
'lightblue')

        self.C_W.create_line(25,120, 25, 10,fill = 'black', width = 2, arrow
= 'last')
        self.C_W.create_oval(21,61,29,69, fill = 'black')

        self.C_W.create_line(60,120,60,10, fill = 'red', width = 2, arrow =
'last')
        self.C_W.create_line(60,105,115,90, fill = 'red', width = 2, arrow =
'last')
        self.C_W.create_line(115,90,100,40, fill = 'red', width = 2, arrow =
'last')
```

```
        self.C_W.create_line(100,40,60,25, fill = 'red', width = 2, arrow =
'last')

        self.C_W.create_rectangle(54,101,64,109, fill = 'green')
        self.C_W.create_rectangle(111,86,119,94, fill = 'green')
        self.C_W.create_rectangle(96,36,104,44, fill = 'green')
        self.C_W.create_rectangle(54,21,64,29, fill = 'green')

        self.C_W.place (x=1030, y=170)

    def wayNo (self):
        self.W = 0

    def wayLong (self):
        self.W = 1

    def wayShort (self):
        self.W = 2

    def wayHighid (self):
        self.W = 3

    def wayLowid (self):
        self.W = 4

    def wayMulti (self):
        self.W = 5

    def probOptions(self):
        self.e = StringVar()
        self.e.set(None)
        Problem_Option  = Label(self.r, text = "Problem", font = ('Times',
'16')).place(x = 1160, y = 140)
        Problem_No      = Radiobutton(self.r, text = "Don't Move and
Report",command = self.probNo,    font = ('Times', '14', 'italic'), variable
= self.e, value = 19).place(x = 1160, y = 170)
        button_Problem  = Radiobutton(self.r, text = 'Move', command =
self.prob,  font = ('Times', '14', 'italic'), variable = self.e, value =
20).place(x = 1160, y = 200)

    def probNo (self):
        self.P = 0

    def prob (self):
        self.P = 1

    #---------------User Interface End-------------------------------------
    def required(self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'WISLR_Having_Crash').fetchone():
            cur.execute ('DROP TABLE WISLR_Having_Crash')

        cur.execute ('CREATE TABLE WISLR_Having_Crash ( Unique_ID integer,
WISLR_Offset double, STNid integer, STNstart integer, STNend integer,\
```

```
                                            WISLRid integer, WISLRstart
integer, WISLREnd integer,\
                                            T string, M string, W string, P
string,\
                                            Record_Historic string, County
string)')
        conn.commit()

        if cur.tables(table = 'WISLR_Having_Crash_Current').fetchone():
            cur.execute ('DROP TABLE WISLR_Having_Crash_Current')

        cur.execute ('CREATE TABLE WISLR_Having_Crash_Current ( Unique_ID
integer, WISLR_Offset double, STNid integer, STNstart integer, STNend
integer,\
                                            WISLRid integer, WISLRstart
integer, WISLREnd integer,\
                                            T string, M string, W string, P
string,\
                                            Record_Historic string, County
string)')
        conn.commit()

        if cur.tables(table = 'WISLR_Having_Crash_Historic').fetchone():
            cur.execute ('DROP TABLE WISLR_Having_Crash_Historic')

        cur.execute ('CREATE TABLE WISLR_Having_Crash_Historic ( Unique_ID
integer, WISLR_Offset double, STNid integer, STNstart integer, STNend
integer,\
                                            WISLRid integer, WISLRstart
integer, WISLREnd integer,\
                                            T string, M string, W string, P
string,\
                                            Record_Historic string, County
string)')
        conn.commit()

        if cur.tables(table = 'WISLR_Having_Crash_Historic_Only').fetchone():
            cur.execute ('DROP TABLE WISLR_Having_Crash_Historic_Only')

        cur.execute ('CREATE TABLE WISLR_Having_Crash_Historic_Only (
Unique_ID integer, WISLR_Offset double, STNid integer, STNstart integer,
STNend integer,\
                                            WISLRid integer, WISLRstart
integer, WISLREnd integer,\
                                            T string, M string, W string, P
string,\
                                            Record_Historic string, County
string)')
        conn.commit()

        if cur.tables(table = 'Crashes').fetchone():
            cur.execute ('DROP TABLE Crashes')

        cur.execute ('CREATE TABLE Crashes ( Unique_ID integer, WISLR_Offset
double, STNid integer, STNstart integer, STNend integer,\
                                            WISLRid integer, WISLRstart
integer, WISLREnd integer, T string, M string, W string, P string,\
```

```
                                                Record_Historic string, County
string)')
        conn.commit()
        if cur.tables(table = 'Crashes_No_F').fetchone():
            cur.execute ('DROP TABLE Crashes_No_F')

        cur.execute ('CREATE TABLE Crashes_No_F ( Unique_ID integer,
WISLR_Offset double, STNid integer, STNstart integer, STNend integer,\
                                                WISLRid integer, WISLRstart
integer, WISLREnd integer, T string, M string, W string, P string,\
                                                Record_Historic string, County
string)')
        conn.commit()
        print ('Crashes  created')


#=============================================================================
        if cur.tables(table = 'Crashes_Flag').fetchone():
            cur.execute ('DROP TABLE Crashes_Flag')

        cur.execute ('CREATE TABLE Crashes_Flag ( Unique_ID integer,
WISLR_Offset double, STNid integer, STNstart integer, STNend integer,\
                                                WISLRid integer, WISLRstart
integer, WISLREnd integer, T string, M string, W string, P string,\
                                                Record_Historic string, County
string)')
        conn.commit()
        print ('Crashes  created')
#=============================================================================
        if cur.tables(table = 'Crashes_No_Flag').fetchone():
            cur.execute ('DROP TABLE Crashes_No_Flag')

        cur.execute ('CREATE TABLE Crashes_No_Flag ( Unique_ID integer,
WISLR_Offset double, STNid integer, STNstart integer, STNend integer,\
                                                WISLRid integer, WISLRstart
integer, WISLREnd integer, T string, M string, W string, P string,\
                                                Record_Historic string, County
string)')
        conn.commit()
        print ('Crashes  created')

########################   TABLE POPULATION   ###############################
#=============================================================================
cur.execute('INSERT INTO    WISLR_Having_Crash ( Unique_ID, WISLR_Offset,
STNid, STNstart, STNend,\
                                                    WISLRid, WISLRstart,
WISLREnd, T, M, W, P,\
                                                    Record_Historic, County)\
            SELECT      WISLR_Points.Unique_ID, WISLR_Points.WISLR_Offset,
link_link.STNid, link_link.STNstart, link_link.STNend,\
                            link_link.WISLRid, link_link.WISLRstart,
link_link.WISLREnd,\
                            link_link.T, link_link.M, link_link.W,
link_link.P,\
                            link_link.Record_Historic, link_link.County\
            FROM        WISLR_Points\
            LEFT JOIN   link_link\
```

```
            ON          link_link.WISLRid = WISLR_Points.WISLRid\
            WHERE       (WISLR_Points.WISLR_Offset > link_link.WISLRstart AND
WISLR_Points.WISLR_Offset <= link_link.WISLRend)\
                        OR (WISLR_Points.WISLR_Offset = 0 AND
link_link.WISLRstart = 0)\
                        OR (WISLR_Points.WISLR_Offset = link_link.WISLRstart
AND link_link.W IS Not Null)')
        conn.commit()

#================================================================================
        cur.execute('INSERT INTO    WISLR_Having_Crash_Current ( Unique_ID,
WISLR_Offset, STNid, STNstart, STNend,\
                                                WISLRid, WISLRstart,
WISLRend, T, M, W, P,\
                                                Record_Historic, County)\
            SELECT      * FROM WISLR_Having_Crash\
            WHERE       WISLR_Having_Crash.Record_Historic IS NULL')
        conn.commit()
#================================================================================
cur.execute('INSERT INTO    WISLR_Having_Crash_Historic ( Unique_ID,
WISLR_Offset, STNid, STNstart, STNend,\
                                                WISLRid, WISLRstart,
WISLRend, T, M, W, P,\
                                                Record_Historic, County)\
            SELECT      * FROM WISLR_Having_Crash\
            WHERE       WISLR_Having_Crash.Record_Historic IS NOT NULL')
#--------------------------------------------------------------------------------
        cur.execute('INSERT INTO    WISLR_Having_Crash_Historic_Only (
Unique_ID, WISLR_Offset, STNid, STNstart, STNend,\
                                                WISLRid, WISLRstart,
WISLRend, T, M, W, P,\
                                                Record_Historic, County)\
            SELECT      WISLR_Having_Crash.Unique_ID,
WISLR_Having_Crash.WISLR_Offset, WISLR_Having_Crash.STNid,
WISLR_Having_Crash.STNstart, WISLR_Having_Crash.STNend,\
                            WISLR_Having_Crash.WISLRid,
WISLR_Having_Crash.WISLRstart, WISLR_Having_Crash.WISLRend,\
                            WISLR_Having_Crash.T, WISLR_Having_Crash.M,
WISLR_Having_Crash.W, WISLR_Having_Crash.P,\
                            WISLR_Having_Crash.Record_Historic,
WISLR_Having_Crash.County\
            FROM        WISLR_Having_Crash\
            LEFT JOIN   WISLR_Having_Crash_Current\
            ON          WISLR_Having_Crash.Unique_ID =
WISLR_Having_Crash_Current.Unique_ID\
            WHERE       WISLR_Having_Crash_Current.Unique_ID IS NULL')


        conn.commit()
#--------------------------------------------------------------------------------
        cur.execute('INSERT INTO    Crashes ( Unique_ID, WISLR_Offset, STNid,
STNstart, STNend,\
                                                WISLRid, WISLRstart,
WISLRend, T, M, W, P,\
                                                Record_Historic, County)\
            SELECT      * FROM WISLR_Having_Crash_Current\
            WHERE       WISLR_Having_Crash_Current.Unique_ID IS NOT NULL')
```

```
#-------------------------------------------------------------------------
        cur.execute('INSERT INTO    Crashes ( Unique_ID, WISLR_Offset, STNid,
STNstart, STNend,\
                                                WISLRid, WISLRstart,
WISLRend, T, M, W, P,\
                                                Record_Historic, County)\
            SELECT      * FROM WISLR_Having_Crash_Historic_Only\
            WHERE       WISLR_Having_Crash_Historic_Only.Unique_ID IS NOT
NULL')
#-------------------------------------------------------------------------
        cur.execute('INSERT INTO    Crashes_Flag ( Unique_ID, WISLR_Offset,
STNid, STNstart, STNend,\
                                                WISLRid, WISLRstart,
WISLRend, T, M, W, P,\
                                                Record_Historic, County)\
            SELECT      * FROM Crashes\
            WHERE       Crashes.M IS NOT NULL\
                        OR Crashes.T IS NOT NULL\
                        OR Crashes.W IS NOT NULL\
                        OR Crashes.P IS NOT NULL')
#-------------------------------------------------------------------------
        cur.execute('INSERT INTO    Crashes_No_F ( Unique_ID, WISLR_Offset,
STNid, STNstart, STNend,\
                                                WISLRid, WISLRstart,
WISLRend, T, M, W, P,\
                                                Record_Historic, County)\
            SELECT      * FROM Crashes\
            WHERE       Crashes.M IS NULL\
                        AND Crashes.T IS NULL\
                        AND Crashes.W IS NULL\
                        AND Crashes.P IS NULL')
#-------------------------------------------------------------------------
        cur.execute('INSERT INTO    Crashes_No_Flag ( Unique_ID,
WISLR_Offset, STNid, STNstart, STNend,\
                                                WISLRid, WISLRstart,
WISLRend, T, M, W, P,\
                                                Record_Historic, County)\
            SELECT      Crashes_No_F.Unique_ID, Crashes_No_F.WISLR_Offset,
Crashes_No_F.STNid, Crashes_No_F.STNstart, Crashes_No_F.STNend,\
                        Crashes_No_F.WISLRid, Crashes_No_F.WISLRstart,
Crashes_No_F.WISLRend,\
                        Crashes_No_F.T, Crashes_No_F.M, Crashes_No_F.W,
Crashes_No_F.P,\
                        Crashes_No_F.Record_Historic, Crashes_No_F.County\
            FROM        Crashes_No_F\
            LEFT JOIN   Crashes_Flag\
            ON          Crashes_No_F.Unique_ID = Crashes_Flag.Unique_ID\
            WHERE       Crashes_Flag.Unique_ID IS NULL')
        conn.commit()
#-------------------------------------------------------------------------
        print 'Done!'

        if cur.tables(table = 'WISLR_Having_Crash').fetchone():
            cur.execute ('DROP TABLE WISLR_Having_Crash')
        if cur.tables(table = 'WISLR_Having_Crash_Historic').fetchone():
            cur.execute ('DROP TABLE WISLR_Having_Crash_Historic')
        if cur.tables(table = 'WISLR_Having_Crash_Current').fetchone():
```

```
            cur.execute ('DROP TABLE WISLR_Having_Crash_Current')
        if cur.tables(table = 'WISLR_Having_Crash_Historic_Only').fetchone():
            cur.execute ('DROP TABLE WISLR_Having_Crash_Historic_Only')
        if cur.tables(table = 'Crashes').fetchone():
            cur.execute (' DROP TABLE Crashes')
        if cur.tables(table = 'Crashes_No_F').fetchone():
            cur.execute (' DROP TABLE Crashes_No_F')
        conn.commit()

#-------------------------------------------------------------------------------
        if cur.tables(table = 'STN_Points').fetchone():
            cur.execute ('DROP TABLE STN_Points')
        cur.execute ('CREATE TABLE STN_Points( Unique_ID string, Link_ID
string, Link_Offset string, Flag string)')

        conn.commit()

        print 'table created'

        if cur.tables(table = 'STN_Points_No_Flag').fetchone():
            cur.execute ('DROP TABLE STN_Points_No_Flag')
        cur.execute ('CREATE TABLE STN_Points_No_Flag( Unique_ID string,
WISLRid string, WISLR_Offset string, WISLRstart string, WISLRend string,\
                                        STNid string, STN_Offset string,
STNstart string, STNend string, Record_Historic string, County string)')

        cur.execute('INSERT INTO    STN_Points_No_Flag (Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLRend,\
                                             STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_No_Flag.Unique_ID,
Crashes_No_Flag.WISLRid, Crashes_No_Flag.WISLR_Offset,
Crashes_No_Flag.WISLRstart, Crashes_No_Flag.WISLRend,\
                                Crashes_No_Flag.STNid,
Crashes_No_Flag.STNstart, Crashes_No_Flag.STNend,
Crashes_No_Flag.Record_Historic, Crashes_No_Flag.County\
                FROM            Crashes_No_Flag\
                ORDER BY        Unique_ID')
        conn.commit()

        cur.execute('UPDATE     STN_Points_No_Flag\
            SET         STN_Offset = STNstart + (WISLR_Offset -
WISLRstart)*(STNend - STNstart)/(WISLRend - WISLRstart)')

        CNF = [r for r in cur.execute( ' SELECT STN_Points_No_Flag.Unique_ID,
STN_Points_No_Flag.STNid, STN_Points_No_Flag.STN_Offset\
                                    FROM STN_Points_No_Flag\
                                    ORDER BY STN_Points_No_Flag.Unique_ID' )]
        CNF_1 = [ max (g, key = itemgetter(0)) for Z, g in groupby (CNF,
itemgetter(0))]
        i = 0
        while i < len (CNF_1):
            cur.execute ( "INSERT INTO STN_Points (Unique_ID, Link_ID,
Link_Offset)\
                        VALUES ('%s', '%s', '%s')"\
                      % (CNF_1[i][0], CNF_1[i][1], CNF_1[i][2] ))
            i = i + 1
```

```
            conn.commit
        conn.commit()

        if cur.tables(table = 'Crashes_No_Flag').fetchone():
            cur.execute (' DROP TABLE Crashes_No_Flag')
        conn.commit()

        print 'DONE'

        #self.test()


        cur.close()
        conn.close()

#-------------------------------------------------------------------------

    def medStart (self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_M').fetchone():
            cur.execute ('DROP TABLE STN_Points_M')
        cur.execute ('CREATE TABLE STN_Points_M ( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLRend string,\
                                            STNid string, STN_Offset string,
STNstart string, STNend string, Record_Historic string, County string, Flag
string)')

        cur.execute('INSERT INTO    STN_Points_M (Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLRend,\
                                                STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLRend,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.M IS NOT NULL AND Crashes_Flag.P
IS NULL\
                ORDER BY        Unique_ID')
        cur.execute("UPDATE     STN_Points_M\
                SET         Flag = 'M'")

        conn.commit()
        cur.execute('UPDATE     STN_Points_M\
                SET         STN_Offset = STNstart')
        conn.commit()
        print 'medianStart selected'


    def medEnd (self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()
```

```
        if cur.tables(table = 'STN_Points_M').fetchone():
            cur.execute ('DROP TABLE STN_Points_M')
        cur.execute ('CREATE TABLE STN_Points_M ( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLREnd string,\
                                        STNid string, STN_Offset string,
STNstart string, STNend string, Record_Historic string, County string, Flag
string)')

        cur.execute('INSERT INTO    STN_Points_M (Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLREnd,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLREnd,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.M IS NOT NULL AND Crashes_Flag.P
IS NULL\
                ORDER BY        Unique_ID')
        cur.execute("UPDATE     STN_Points_M\
                SET         Flag = 'M'")

        conn.commit()

        cur.execute('UPDATE     STN_Points_M\
                SET         STN_Offset = STNend')
        conn.commit()
        print 'medEnd'


    def medMid (self):

        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_M').fetchone():
            cur.execute ('DROP TABLE STN_Points_M')
        cur.execute ('CREATE TABLE STN_Points_M ( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLREnd string,\
                                        STNid string, STN_Offset string,
STNstart string, STNend string, Record_Historic string, County string, Flag
string)')

        cur.execute('INSERT INTO    STN_Points_M (Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLREnd,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLREnd,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.M IS NOT NULL AND Crashes_Flag.P
IS NULL\
```

```
                ORDER BY        Unique_ID')
        cur.execute("UPDATE     STN_Points_M\
                SET         Flag = 'M'")

        conn.commit()
        cur.execute('UPDATE     STN_Points_M\
                SET         STN_Offset = (STNstart + STNend)/2')
        conn.commit()


    def turLon(self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_T').fetchone():
            cur.execute ('DROP TABLE STN_Points_T')
        cur.execute ('CREATE TABLE STN_Points_T ( Unique_ID string,STNid
string, STN_Offset string, Flag string)')

        if cur.tables(table = 'STN_Points_T_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_T_1')
        cur.execute ('CREATE TABLE STN_Points_T_1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLREnd string,\
                                            STNid string, STN_Length
string,STN_Offset string, STNstart string, STNend string, Record_Historic
string, County string, Flag string)')
        cur.execute('INSERT INTO    STN_Points_T_1(Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLREnd,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLREnd,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.T IS NOT NULL AND Crashes_Flag.M
IS NULL AND Crashes_Flag.P IS NULL\
                ORDER BY        Unique_ID')

        cur.execute("UPDATE     STN_Points_T_1\
                SET         STN_Length = (STNend - STNstart)")

        cur.execute('UPDATE     STN_Points_T_1\
            SET         STN_Offset = STNstart + (WISLR_Offset -
WISLRstart)*(STNend - STNstart)/(WISLREnd - WISLRstart)')
        conn.commit()

        TLL = [r for r in cur.execute('SELECT   STN_Points_T_1.Unique_ID,
STN_Points_T_1.STN_Length, STN_Points_T_1.STNid,STN_Points_T_1.STN_Offset\
                                    FROM STN_Points_T_1\
                                    ORDER BY STN_Points_T_1.Unique_ID,
STN_Points_T_1.STNid')]
        TLL_L = [max (g, key= itemgetter(1)) for z, g in groupby(TLL, key =
itemgetter(0))]
        i = 0
        while i < len (TLL_L):
```

```
            cur.execute ( "INSERT INTO STN_Points_T (Unique_ID, STNid,
STN_Offset)\
                          VALUES ('%s', '%s', '%s')"\
                        % (TLL_L[i][0], TLL_L[i][2], TLL_L[i][3]))
            i = i + 1

            conn.commit
        conn.commit()

        if cur.tables(table = 'STN_Points_T_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_T_1')

        cur.execute("UPDATE     STN_Points_T\
                SET         Flag = 'T'")

        conn.commit()
        print 'turnLong selected'

    def turSho(self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_T').fetchone():
            cur.execute ('DROP TABLE STN_Points_T')
        cur.execute ('CREATE TABLE STN_Points_T ( Unique_ID string,STNid
string, STN_Offset string, Flag string)')

        if cur.tables(table = 'STN_Points_T_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_T_1')
        cur.execute ('CREATE TABLE STN_Points_T_1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLREnd string,\
                                            STNid string, STN_Length
string,STN_Offset string, STNstart string, STNend string, Record_Historic
string, County string, Flag string)')
        cur.execute('INSERT INTO    STN_Points_T_1(Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLREnd,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLREnd,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.T IS NOT NULL AND Crashes_Flag.M
IS NULL AND Crashes_Flag.P IS NULL\
                ORDER BY        Unique_ID')

        cur.execute("UPDATE     STN_Points_T_1\
                SET         STN_Length = (STNend - STNstart)")

        cur.execute('UPDATE     STN_Points_T_1\
            SET         STN_Offset = STNstart + (WISLR_Offset -
WISLRstart)*(STNend - STNstart)/(WISLREnd - WISLRstart)')
        conn.commit()
```

```
        TLL = [r for r in cur.execute('SELECT   STN_Points_T_1.Unique_ID,
STN_Points_T_1.STN_Length, STN_Points_T_1.STNid,STN_Points_T_1.STN_Offset\
                                    FROM STN_Points_T_1\
                                    ORDER BY STN_Points_T_1.Unique_ID,
STN_Points_T_1.STNid')]
        TLL_S = [min (g, key= itemgetter(1)) for z, g in groupby(TLL, key =
itemgetter(0))]
        i = 0
        while i < len (TLL_S):
            cur.execute ( "INSERT INTO STN_Points_T (Unique_ID, STNid,
STN_Offset)\
                        VALUES ('%s', '%s', '%s')"\
                      % (TLL_S[i][0], TLL_S[i][2], TLL_S[i][3]))
            i = i + 1

            conn.commit
        conn.commit()

        if cur.tables(table = 'STN_Points_T_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_T_1')

        cur.execute("UPDATE     STN_Points_T\
                SET         Flag = 'T'")
        conn.commit()
        print 'turnShort selected'

    def turHid(self):

        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_T').fetchone():
            cur.execute ('DROP TABLE STN_Points_T')
        cur.execute ('CREATE TABLE STN_Points_T ( Unique_ID string,STNid
string, STN_Offset string, Flag string)')

        if cur.tables(table = 'STN_Points_T_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_T_1')
        cur.execute ('CREATE TABLE STN_Points_T_1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLRend string,\
                                            STNid string, STN_Length
string,STN_Offset string, STNstart string, STNend string, Record_Historic
string, County string, Flag string)')
        cur.execute('INSERT INTO    STN_Points_T_1(Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLRend,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLRend,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.T IS NOT NULL AND Crashes_Flag.M
IS NULL AND Crashes_Flag.P IS NULL\
                ORDER BY        Unique_ID')
```

```
        cur.execute("UPDATE     STN_Points_T_1\
                SET         STN_Length = (STNend - STNstart)")

        cur.execute('UPDATE     STN_Points_T_1\
            SET         STN_Offset = STNstart + (WISLR_Offset -
WISLRstart)*(STNend - STNstart)/(WISLRend - WISLRstart)')
        conn.commit()

        TLL = [r for r in cur.execute('SELECT   STN_Points_T_1.Unique_ID,
STN_Points_T_1.STN_Length, STN_Points_T_1.STNid,STN_Points_T_1.STN_Offset\
                                    FROM STN_Points_T_1\
                                    ORDER BY STN_Points_T_1.Unique_ID,
STN_Points_T_1.STNid')]
        TLL_H = [max (g, key= itemgetter(2)) for z, g in groupby(TLL, key =
itemgetter(0))]
        i = 0
        while i < len (TLL_H):
            cur.execute ( "INSERT INTO STN_Points_T (Unique_ID, STNid,
STN_Offset)\
                        VALUES ('%s', '%s', '%s')"\
                    % (TLL_H[i][0], TLL_H[i][2], TLL_H[i][3]))
            i = i + 1

            conn.commit
        conn.commit()

        if cur.tables(table = 'STN_Points_T_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_T_1')

        cur.execute("UPDATE     STN_Points_T\
                SET         Flag = 'T'")

        conn.commit()
        print 'turnLowid selected'



    def turLid(self):

        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_T').fetchone():
            cur.execute ('DROP TABLE STN_Points_T')
        cur.execute ('CREATE TABLE STN_Points_T ( Unique_ID string,STNid
string, STN_Offset string, Flag string)')

        if cur.tables(table = 'STN_Points_T_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_T_1')
        cur.execute ('CREATE TABLE STN_Points_T_1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLRend string,\
                                            STNid string, STN_Length
string,STN_Offset string, STNstart string, STNend string, Record_Historic
string, County string, Flag string)')
        cur.execute('INSERT INTO    STN_Points_T_1(Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLRend,\
```

```
                                        STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLRend,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.T IS NOT NULL AND Crashes_Flag.M
IS NULL AND Crashes_Flag.P IS NULL\
                ORDER BY        Unique_ID')

        cur.execute("UPDATE     STN_Points_T_1\
                SET         STN_Length = (STNend - STNstart)")

        cur.execute('UPDATE     STN_Points_T_1\
            SET         STN_Offset = STNstart + (WISLR_Offset -
WISLRstart)*(STNend - STNstart)/(WISLRend - WISLRstart)')
        conn.commit()

        TLL = [r for r in cur.execute('SELECT   STN_Points_T_1.Unique_ID,
STN_Points_T_1.STN_Length, STN_Points_T_1.STNid,STN_Points_T_1.STN_Offset\
                                    FROM STN_Points_T_1\
                                    ORDER BY STN_Points_T_1.Unique_ID,
STN_Points_T_1.STNid')]
        TLL_Lo = [max (g, key= itemgetter(2)) for z, g in groupby(TLL, key =
itemgetter(0))]
        i = 0
        while i < len (TLL_Lo):
            cur.execute ( "INSERT INTO STN_Points_T (Unique_ID, STNid,
STN_Offset)\
                        VALUES ('%s', '%s', '%s')"\
                      % (TLL_Lo[i][0], TLL_Lo[i][2], TLL_Lo[i][3]))
            i = i + 1

            conn.commit
        conn.commit()

        if cur.tables(table = 'STN_Points_T_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_T_1')

        cur.execute("UPDATE     STN_Points_T\
                SET         Flag = 'T'")

        conn.commit()
        print 'turnHighid selected'

    def turMul(self):

        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_T').fetchone():
            cur.execute ('DROP TABLE STN_Points_T')
        cur.execute ('CREATE TABLE STN_Points_T ( Unique_ID string,STNid
string, STN_Offset string, Flag string)')
```

```
        if cur.tables(table = 'STN_Points_T_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_T_1')
        cur.execute ('CREATE TABLE STN_Points_T_1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLRend string,\
                                                STNid string, STN_Length
string,STN_Offset string, STNstart string, STNend string, Record_Historic
string, County string, Flag string)')
        cur.execute('INSERT INTO    STN_Points_T_1(Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLRend,\
                                                STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLRend,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.T IS NOT NULL AND Crashes_Flag.M
IS NULL AND Crashes_Flag.P IS NULL\
                ORDER BY        Unique_ID')

        cur.execute("UPDATE     STN_Points_T_1\
                SET         STN_Length = (STNend - STNstart)")

        cur.execute('UPDATE     STN_Points_T_1\
            SET         STN_Offset = STNstart + (WISLR_Offset -
WISLRstart)*(STNend - STNstart)/(WISLRend - WISLRstart)')
        conn.commit()

        TLL = [r for r in cur.execute('SELECT   STN_Points_T_1.Unique_ID,
STN_Points_T_1.STN_Length, STN_Points_T_1.STNid,STN_Points_T_1.STN_Offset\
                                    FROM STN_Points_T_1\
                                    ORDER BY STN_Points_T_1.Unique_ID,
STN_Points_T_1.STNid')]
        #TLL_Lo = [max (g, key= itemgetter(2)) for z, g in groupby(TLL, key =
itemgetter(0))]
        i = 0
        while i < len (TLL):
            cur.execute ( "INSERT INTO STN_Points_T (Unique_ID, STNid,
STN_Offset)\
                        VALUES ('%s', '%s', '%s')"\
                    % (TLL[i][0], TLL[i][2], TLL[i][3]))
            i = i + 1

            conn.commit
        conn.commit()

        if cur.tables(table = 'STN_Points_T_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_T_1')

        cur.execute("UPDATE     STN_Points_T\
                SET         Flag = 'T'")

        conn.commit()
        print 'turnMulti selected'

    def wayLon(self):
```

```
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_W').fetchone():
            cur.execute ('DROP TABLE STN_Points_W')
        cur.execute ('CREATE TABLE STN_Points_W ( Unique_ID string,STNid
string, STN_Offset string, Flag string)')

        if cur.tables(table = 'STN_Points_W_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_W_1')
        cur.execute ('CREATE TABLE STN_Points_W_1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLREnd string,\
                                            STNid string, STN_Length
string,STN_Offset string, STNstart string, STNend string, Record_Historic
string, County string, Flag string)')
        cur.execute('INSERT INTO    STN_Points_W_1(Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLREnd,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLREnd,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.T IS NOT NULL AND Crashes_Flag.T
IS NULL AND Crashes_Flag.M IS NULL AND Crashes_Flag.P IS NULL\
                ORDER BY        Unique_ID')

        cur.execute("UPDATE     STN_Points_W_1\
                SET         STN_Length = (STNend - STNstart)")

        cur.execute('UPDATE     STN_Points_W_1\
            SET         STN_Offset = ((STNstart + STNend)/2)')
        conn.commit()

        W = [r for r in cur.execute('SELECT   STN_Points_W_1.Unique_ID,
STN_Points_W_1.STN_Length, STN_Points_W_1.STNid,STN_Points_W_1.STN_Offset\
                                    FROM STN_Points_W_1\
                                    ORDER BY STN_Points_W_1.Unique_ID,
STN_Points_W_1.STNid')]
        W_L = [max (g, key= itemgetter(1)) for z, g in groupby(W, key =
itemgetter(0))]
        i = 0
        while i < len (W_L):
            cur.execute ( "INSERT INTO STN_Points_W (Unique_ID, STNid,
STN_Offset)\
                        VALUES ('%s', '%s', '%s')"\
                      % (W_L[i][0], W_L[i][2], W_L[i][3]))
            i = i + 1

            conn.commit
        conn.commit()

        if cur.tables(table = 'STN_Points_W_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_W_1')
```

```
        cur.execute("UPDATE     STN_Points_W\
                SET         Flag = 'W'")

        conn.commit()
        print 'wayLong selected'

    def waySho(self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_W').fetchone():
            cur.execute ('DROP TABLE STN_Points_W')
        cur.execute ('CREATE TABLE STN_Points_W ( Unique_ID string,STNid
string, STN_Offset string, Flag string)')

        if cur.tables(table = 'STN_Points_W_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_W_1')
        cur.execute ('CREATE TABLE STN_Points_W_1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLREnd string,\
                                            STNid string, STN_Length
string,STN_Offset string, STNstart string, STNend string, Record_Historic
string, County string, Flag string)')
        cur.execute('INSERT INTO    STN_Points_W_1(Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLREnd,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLREnd,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.T IS NOT NULL AND Crashes_Flag.T
IS NULL AND Crashes_Flag.M IS NULL AND Crashes_Flag.P IS NULL\
                ORDER BY        Unique_ID')

        cur.execute("UPDATE     STN_Points_W_1\
                SET         STN_Length = (STNend - STNstart)")

        cur.execute('UPDATE     STN_Points_W_1\
            SET         STN_Offset = ((STNstart + STNend)/2)')
        conn.commit()

        W = [r for r in cur.execute('SELECT   STN_Points_W_1.Unique_ID,
STN_Points_W_1.STN_Length, STN_Points_W_1.STNid,STN_Points_W_1.STN_Offset\
                                    FROM STN_Points_W_1\
                                    ORDER BY STN_Points_W_1.Unique_ID,
STN_Points_W_1.STNid')]
        W_L = [min (g, key= itemgetter(1)) for z, g in groupby(W, key =
itemgetter(0))]
        i = 0
        while i < len (W_L):
            cur.execute ( "INSERT INTO STN_Points_W (Unique_ID, STNid,
STN_Offset)\
                        VALUES ('%s', '%s', '%s')"\
                      % (W_L[i][0], W_L[i][2], W_L[i][3]))
            i = i + 1
```

```
            conn.commit
        conn.commit()

        if cur.tables(table = 'STN_Points_W_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_W_1')

        cur.execute("UPDATE     STN_Points_W\
                SET         Flag = 'W'")

        conn.commit()
        print 'wayShort selected'

    def wayHid(self):

        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_W').fetchone():
            cur.execute ('DROP TABLE STN_Points_W')
        cur.execute ('CREATE TABLE STN_Points_W ( Unique_ID string,STNid
string, STN_Offset string, Flag string)')

        if cur.tables(table = 'STN_Points_W_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_W_1')
        cur.execute ('CREATE TABLE STN_Points_W_1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLREnd string,\
                                            STNid string, STN_Length
string,STN_Offset string, STNstart string, STNend string, Record_Historic
string, County string, Flag string)')
        cur.execute('INSERT INTO    STN_Points_W_1(Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLREnd,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLREnd,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.T IS NOT NULL AND Crashes_Flag.T
IS NULL AND Crashes_Flag.M IS NULL AND Crashes_Flag.P IS NULL\
                ORDER BY        Unique_ID')

        cur.execute("UPDATE     STN_Points_W_1\
                SET         STN_Length = (STNend - STNstart)")

        cur.execute('UPDATE     STN_Points_W_1\
            SET         STN_Offset = ((STNstart + STNend)/2)')
        conn.commit()

        W = [r for r in cur.execute('SELECT   STN_Points_W_1.Unique_ID,
STN_Points_W_1.STN_Length, STN_Points_W_1.STNid,STN_Points_W_1.STN_Offset\
                                    FROM STN_Points_W_1\
                                    ORDER BY STN_Points_W_1.Unique_ID,
STN_Points_W_1.STNid')]
```

```
        W_L = [max (g, key= itemgetter(2)) for z, g in groupby(W, key =
itemgetter(0))]
        i = 0
        while i < len (W_L):
            cur.execute ( "INSERT INTO STN_Points_W (Unique_ID, STNid,
STN_Offset)\
                        VALUES ('%s', '%s', '%s')"\
                      % (W_L[i][0], W_L[i][2], W_L[i][3]))
            i = i + 1

            conn.commit
        conn.commit()

        if cur.tables(table = 'STN_Points_W_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_W_1')

        cur.execute("UPDATE     STN_Points_W\
                SET         Flag = 'W'")

        conn.commit()
        print 'wayHighid selected'

    def wayLid(self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_W').fetchone():
            cur.execute ('DROP TABLE STN_Points_W')
        cur.execute ('CREATE TABLE STN_Points_W ( Unique_ID string,STNid
string, STN_Offset string, Flag string)')

        if cur.tables(table = 'STN_Points_W_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_W_1')
        cur.execute ('CREATE TABLE STN_Points_W_1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLREnd string,\
                                            STNid string, STN_Length
string,STN_Offset string, STNstart string, STNend string, Record_Historic
string, County string, Flag string)')
        cur.execute('INSERT INTO    STN_Points_W_1(Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLREnd,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLREnd,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.T IS NOT NULL AND Crashes_Flag.T
IS NULL AND Crashes_Flag.M IS NULL AND Crashes_Flag.P IS NULL\
                ORDER BY        Unique_ID')

        cur.execute("UPDATE     STN_Points_W_1\
                SET         STN_Length = (STNend - STNstart)")

        cur.execute('UPDATE     STN_Points_W_1\
            SET         STN_Offset = ((STNstart + STNend)/2)')
```

```
        conn.commit()

        W = [r for r in cur.execute('SELECT   STN_Points_W_1.Unique_ID,
STN_Points_W_1.STN_Length, STN_Points_W_1.STNid,STN_Points_W_1.STN_Offset\
                                    FROM STN_Points_W_1\
                                    ORDER BY STN_Points_W_1.Unique_ID,
STN_Points_W_1.STNid')]
        W_L = [min (g, key= itemgetter(2)) for z, g in groupby(W, key =
itemgetter(0))]
        i = 0
        while i < len (W_L):
            cur.execute ( "INSERT INTO STN_Points_W (Unique_ID, STNid,
STN_Offset)\
                        VALUES ('%s', '%s', '%s')"\
                    % (W_L[i][0], W_L[i][2], W_L[i][3]))
            i = i + 1

            conn.commit
        conn.commit()

        if cur.tables(table = 'STN_Points_W_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_W_1')

        cur.execute("UPDATE     STN_Points_W\
                SET         Flag = 'W'")

        conn.commit()

        print 'wayLowid selected'

    def wayMul(self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_W').fetchone():
            cur.execute ('DROP TABLE STN_Points_W')
        cur.execute ('CREATE TABLE STN_Points_W ( Unique_ID string,STNid
string, STN_Offset string, Flag string)')

        if cur.tables(table = 'STN_Points_W_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_W_1')
        cur.execute ('CREATE TABLE STN_Points_W_1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLREnd string,\
                                            STNid string, STN_Length
string,STN_Offset string, STNstart string, STNend string, Record_Historic
string, County string, Flag string)')
        cur.execute('INSERT INTO    STN_Points_W_1(Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLREnd,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLREnd,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
```

```
                WHERE           Crashes_Flag.T IS NOT NULL AND Crashes_Flag.T
IS NULL AND Crashes_Flag.M IS NULL AND Crashes_Flag.P IS NULL\
                ORDER BY        Unique_ID')

        cur.execute("UPDATE     STN_Points_W_1\
                SET         STN_Length = (STNend - STNstart)")

        cur.execute('UPDATE     STN_Points_W_1\
            SET         STN_Offset = ((STNstart + STNend)/2)')
        conn.commit()

        W = [r for r in cur.execute('SELECT   STN_Points_W_1.Unique_ID,
STN_Points_W_1.STN_Length, STN_Points_W_1.STNid,STN_Points_W_1.STN_Offset\
                                    FROM STN_Points_W_1\
                                    ORDER BY STN_Points_W_1.Unique_ID,
STN_Points_W_1.STNid')]
        #W_L = [min (g, key= itemgetter(2)) for z, g in groupby(W, key =
itemgetter(0))]
        i = 0
        while i < len (W):
            cur.execute ( "INSERT INTO STN_Points_W (Unique_ID, STNid,
STN_Offset)\
                        VALUES ('%s', '%s', '%s')"\
                    % (W[i][0], W[i][2], W[i][3]))
            i = i + 1

            conn.commit
        conn.commit()

        if cur.tables(table = 'STN_Points_W_1').fetchone():
            cur.execute ('DROP TABLE STN_Points_W_1')

        cur.execute("UPDATE     STN_Points_W\
                SET         Flag = 'W'")


        conn.commit()
        print 'wayMulti selected'

    def probMov(self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()

        if cur.tables(table = 'STN_Points_P').fetchone():
            cur.execute ('DROP TABLE STN_Points_P')
        cur.execute ('CREATE TABLE STN_Points_P( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLRend string,\
                                            STNid string, STN_Offset string,
STNstart string, STNend string, Record_Historic string, County string, Flag
string)')
        if cur.tables(table = 'STN_Points_P1').fetchone():
            cur.execute ('DROP TABLE STN_Points_P1')
        cur.execute ('CREATE TABLE STN_Points_P1( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLRend string,\
                                            STNid string, STN_Offset string,
STNstart string, STNend string, Record_Historic string, County string)')
```

```
        if cur.tables(table = 'STN_Points_P2').fetchone():
            cur.execute ('DROP TABLE STN_Points_P2')
        cur.execute ('CREATE TABLE STN_Points_P2( Unique_ID string, WISLRid
string, WISLR_Offset string, WISLRstart string, WISLRend string,\
                                            STNid string, STN_Offset string,
STNstart string, STNend string, Record_Historic string, County string)')

        conn.commit()

        cur.execute('INSERT INTO    STN_Points_P1 (Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLRend,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLRend,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.P IS NOT NULL AND
Crashes_Flag.WISLRend = Crashes_Flag.WISLRstart\
                ORDER BY        Unique_ID')
        cur.execute('UPDATE     STN_Points_P1\
                    SET     STN_Offset = STNstart')

        cur.execute('INSERT INTO    STN_Points_P2 (Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLRend,\
                                            STNid, STNstart, STNend,
Record_Historic, County)\
                SELECT          Crashes_Flag.Unique_ID, Crashes_Flag.WISLRid,
Crashes_Flag.WISLR_Offset, Crashes_Flag.WISLRstart, Crashes_Flag.WISLRend,\
                                Crashes_Flag.STNid, Crashes_Flag.STNstart,
Crashes_Flag.STNend, Crashes_Flag.Record_Historic, Crashes_Flag.County\
                FROM            Crashes_Flag\
                WHERE           Crashes_Flag.P IS NOT NULL \
                            AND Crashes_Flag.WISLRstart <>
Crashes_Flag.WISLRend\
                ORDER BY        Unique_ID')

        cur.execute('UPDATE     STN_Points_P2\
                    SET     STN_Offset = (STNstart + (STNend-
STNstart)*(WISLR_Offset-WISLRstart)/(WISLRend-WISLRstart))')

        cur.execute('INSERT INTO    STN_Points_P (Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLRend,\
                                            STNid, STN_Offset, STNstart,
STNend, Record_Historic, County)\
                SELECT          STN_Points_P1.Unique_ID,
STN_Points_P1.WISLRid, STN_Points_P1.WISLR_Offset, STN_Points_P1.WISLRstart,
STN_Points_P1.WISLRend,\
                                STN_Points_P1.STNid,
STN_Points_P1.STN_Offset, STN_Points_P1.STNstart, STN_Points_P1.STNend,
STN_Points_P1.Record_Historic, STN_Points_P1.County\
                FROM            STN_Points_P1\
                ORDER BY        Unique_ID')
```

```
        cur.execute('INSERT INTO    STN_Points_P (Unique_ID, WISLRid,
WISLR_Offset, WISLRstart, WISLRend,\
                                                STNid, STN_Offset, STNstart,
STNend, Record_Historic, County)\
                SELECT          STN_Points_P2.Unique_ID,
STN_Points_P2.WISLRid, STN_Points_P2.WISLR_Offset, STN_Points_P2.WISLRstart,
STN_Points_P2.WISLRend,\
                                STN_Points_P2.STNid,
STN_Points_P2.STN_Offset, STN_Points_P2.STNstart, STN_Points_P2.STNend,
STN_Points_P2.Record_Historic, STN_Points_P2.County\
                FROM            STN_Points_P2\
                ORDER BY        Unique_ID')
        cur.execute("UPDATE     STN_Points_P\
                SET         Flag = 'P'")
        conn.commit()

        if cur.tables(table = 'STN_Points_P1').fetchone():
            cur.execute ('DROP TABLE STN_Points_P1')
        if cur.tables(table = 'STN_Points_P2').fetchone():
            cur.execute ('DROP TABLE STN_Points_P2')

        conn.commit()

        print 'Prob moved'

    def combine (self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()
        if cur.tables(table = 'STN_Points_P').fetchone():
            cur.execute('INSERT INTO    STN_Points (Unique_ID, Link_ID,
Link_Offset, Flag)\
                SELECT          STN_Points_P.Unique_ID, STN_Points_P.STNid,
STN_Points_P.STN_Offset, STN_Points_P.Flag\
                FROM            STN_Points_P\
                ORDER BY        Unique_ID')
        if cur.tables(table = 'STN_Points_M').fetchone():
            cur.execute('INSERT INTO    STN_Points (Unique_ID, Link_ID,
Link_Offset, Flag)\
                SELECT          STN_Points_M.Unique_ID, STN_Points_M.STNid,
STN_Points_M.STN_Offset, STN_Points_M.Flag\
                FROM            STN_Points_M\
                ORDER BY        Unique_ID')
        if cur.tables(table = 'STN_Points_T').fetchone():
            cur.execute('INSERT INTO    STN_Points (Unique_ID, Link_ID,
Link_Offset, Flag)\
                SELECT          STN_Points_T.Unique_ID, STN_Points_T.STNid,
STN_Points_T.STN_Offset, STN_Points_T.Flag\
                FROM            STN_Points_T\
                ORDER BY        Unique_ID')
        if cur.tables(table = 'STN_Points_W').fetchone():
            cur.execute('INSERT INTO    STN_Points (Unique_ID, Link_ID,
Link_Offset, Flag)\
                SELECT          STN_Points_W.Unique_ID, STN_Points_W.STNid,
STN_Points_W.STN_Offset, STN_Points_W.Flag\
                FROM            STN_Points_W\
                ORDER BY        Unique_ID')
```

```
        conn.commit()

        if cur.tables(table = 'STN_Points_Final').fetchone():
            cur.execute ('DROP table STN_Points_Final')
        cur.execute ('CREATE TABLE STN_Points_Final ( Unique_ID
string,Link_ID string, Link_Offset string, Flag string)')
        conn.commit()

        F = [i for i in cur.execute('SELECT * FROM STN_Points ORDER BY
Unique_ID')]
        F_1 = [min (g, key = itemgetter(0)) for z, g in groupby(F, key =
itemgetter(0))]
        i = 0
        while i < len (W):
            cur.execute ( "INSERT INTO STN_Points_Final (Unique_ID, Link_ID,
Link_Offset, Flag)\
                          VALUES ('%s', '%s', '%s', '%s')"\
                       % (F[i][0],F[i][1], F[i][2], F[i][3]))
            i = i + 1

            conn.commit
        conn.commit()

    def report (self):
        conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};
DBQ=' + self.filepath)
        cur = conn.cursor()
        A = [i for i in cur.execute('SELECT DISTINCT Unique_ID FROM
WISLR_Points ')]
        A1 = Label(self.r, text = (' Total Number of Crashes =' + str
(len(A))), font = ('Times', '12')).place (x = 10, y = 450)

        B = [i for i in cur.execute('SELECT DISTINCT Unique_ID FROM
STN_Points ')]
        PB1 = round ((100 * (float(len (B))/len (A))),2)
        PB2 = (len (A) - len (B))
        PB3 = round ((100 * (float (PB2)/len (A))),2)
        B1 = Label(self.r, text = (' Total Number of Crashes Moved     = ' +
str (len(B)) + '\t' + '( '+ str (PB1) + ' % ' + ' )'), font = ('Times',
'12')).place (x = 10, y = 480)
        B2 = Label(self.r, text = (' Total Number of Crashes Not Moved = ' +
str (PB2) + '\t' + '( '+ str (PB3) + ' % ' + ' )'), font = ('Times',
'12')).place (x = 10, y = 510)


        M = [i for i in cur.execute("SELECT DISTINCT Unique_ID FROM
STN_Points WHERE STN_Points.Flag = 'M'")]
        PM1 = round ((100 * (float(len (M))/len (A))),2)
        M1 = Label(self.r, text = (' Total Number of Crashes On Median Cross
Over     = ' + str (len(M)) + '\t' + '( '+ str (PM1) + ' % ' + ' )'), font =
('Times', '12')).place (x = 10, y = 540)

        T = [i for i in cur.execute("SELECT DISTINCT Unique_ID FROM
STN_Points WHERE STN_Points.Flag = 'T'")]
        PT1 = round ((100 * (float(len (T))/len (A))),2)
```

```
        T1 = Label(self.r, text = (' Total Number of Crashes On Turn Lane
= ' + str (len(T)) + '\t' + '( '+ str (PT1) + ' % ' + ' )'), font = ('Times',
'12')).place (x = 10, y = 570)

        W = [i for i in cur.execute("SELECT DISTINCT Unique_ID FROM
STN_Points WHERE STN_Points.Flag = 'W'")]
        PW1 = round ((100 * (float(len (W))/len (A))),2)
        W1 = Label(self.r, text = (' Total Number of Crashes On Way Side
= ' + str (len(W)) + '\t' + '( '+ str (PW1) + ' % ' + ' )'), font = ('Times',
'12')).place (x = 10, y = 600)

        P = [i for i in cur.execute("SELECT DISTINCT Unique_ID FROM
STN_Points WHERE STN_Points.Flag = 'P'")]
        PP1 = round ((100 * (float(len (P))/len (A))),2)
        P1 = Label(self.r, text = (' Total Number of Crashes On Problem Links
= ' + str (len(P)) + '\t' + '( '+ str (PP1) + ' % ' + ' )'), font = ('Times',
'12')).place (x = 10, y = 630)

    def main(self):
        self.required()

        if self.flag == 1:  self.medStart()
        elif self.flag == 2:self.medEnd ()
        elif self.flag == 3:self.medMid()
        else:pass

        if self.T == 1:   self.turLon()
        elif self.T == 2: self.turSho()
        elif self.T == 3: self.turHid()
        elif self.T == 4: self.turLid()
        elif self.T == 5: self.turMul()
        else:pass

        if self.W == 1:  self.wayLon()
        elif self.W == 2:self.waySho()
        elif self.W == 3:self.wayHid()
        elif self.W == 4:self.wayLid()
        elif self.W == 5:self.wayMul()
        else:pass


        if self.P == 1:self.probMov()
        else:pass

        self.combine()
        self.report()


InterFace(Tk())

print 'Success'
```

APPENDIX D

```
import pyodbc
from itertools import groupby
from operator import itemgetter

DBfile = 'Ramp.mdb'
conn = pyodbc.connect('DRIVER= {Microsoft Access Driver (*.mdb)}; DBQ= ' +
DBfile)
cur = conn.cursor()

''' IDENTIFY RDWY CHN WHERE RDY_CHN_RTE IS Ramp '''
if cur.tables(table = 'Ramp_list').fetchone():
    cur.execute('DROP TABLE Ramp_list')
cur.execute ('CREATE TABLE Ramp_list   (RDWY_CHN_ID integer,STUS
string,RDWY_CHN_Name string)')
cur.execute (" INSERT INTO Ramp_list   (RDWY_CHN_ID,STUS,RDWY_CHN_Name )\
                           SELECT      rdwy_chn_arc.RDWY_CHN_ID,
rdwy_chn_arc.STUS,rdwy_chn_arc.RDWY_CHN_RTE\
                           FROM        rdwy_chn_arc WHERE
rdwy_chn_arc.RDWY_CHN_RTE = 'Ramp'")
conn.commit()

''' IDENTIFY ALL STN RAMP AND MAKE A LIST OF STN RAMP'''

STN_ramp_chn = [ ]
STN_ramp_chn = [ i for i in cur.execute("SELECT rdwy_link_chn.STNid FROM
Ramp_list LEFT JOIN   rdwy_link_chn\
                                        ON  rdwy_link_chn.RDWY_CHN_ID =
Ramp_list.RDWY_CHN_ID\
                                        WHERE rdwy_link_chn.STUS = 'C' AND
Ramp_list.STUS = 'C'")]
STN_ramp_link = []
STN_ramp_link = [ i for i in cur.execute("SELECT   STNid FROM rdwy_rte_link
WHERE STUS = 'C' AND STN_Name = 'OFF'")]


STN_ramp = (STN_ramp_chn + STN_ramp_link)
STN_ramp.sort()
STN_ramp_all = [max(g, key = itemgetter(0)) for z, g in groupby(STN_ramp, key
= itemgetter(0))]

if cur.tables(table = 'Ramp_STN').fetchone():
    cur.execute('DROP TABLE Ramp_STN')
cur.execute ('CREATE TABLE Ramp_STN     (STNid integer)')

for i in STN_ramp_all:
    cur.execute ('  INSERT INTO Ramp_STN     (STNid)\
                VALUES (?);', i[0])

    conn.commit()
```

```
conn.commit()

''' TABLE <Ramp_STN_SITE_F> INSERT ALL STN AND WISLR ALONG WITH START END'''
if cur.tables(table = 'Ramp_STN_SITE_F_T').fetchone():
    cur.execute('DROP TABLE Ramp_STN_SITE_F_T')
cur.execute ( 'CREATE TABLE Ramp_STN_SITE_F_T(STNid integer,STN_SITE_F
integer,STN_F integer,STN_SITE_T integer, STN_T integer)')
conn.commit()

cur.execute (" INSERT INTO
Ramp_STN_SITE_F_T(STNid,STN_SITE_F,STN_SITE_T)SELECT
Ramp_STN.STNid,STN_Sites.REF_SITE_F,STN_Sites.REF_SITE_T\
                FROM Ramp_STN LEFT JOIN STN_Sites ON Ramp_STN.STNid =
STN_Sites.STNid WHERE STN_Sites.STUS = 'C'")
conn.commit()

''' TABLE <Ramp_STN_F> INSERT STN_F'''
if cur.tables(table = 'Ramp_STN_F').fetchone():
    cur.execute('DROP TABLE Ramp_STN_F')
cur.execute ( 'CREATE TABLE Ramp_STN_F(STNid integer,STN_SITE_F integer,STN_F
integer,STN_SITE_T integer,STN_T integer)')
cur.execute (" INSERT INTO Ramp_STN_F
(STNid,STN_SITE_F,STN_F,STN_SITE_T SELECT      Ramp_STN_SITE_F_T.STNid,\

Ramp_STN_SITE_F_T.STN_SITE_F,STN_Sites.STNid,Ramp_STN_SITE_F_T.STN_SITE_T
FROM Ramp_STN_SITE_F_T \
                LEFT JOIN   STN_Sites ON Ramp_STN_SITE_F_T.STN_SITE_F =
STN_Sites.REF_SITE_T WHERE STN_Sites.STUS = 'C'\
                AND Ramp_STN_SITE_F_T.STN_SITE_T <> STN_Sites.REF_SITE_F")

conn.commit()

''' TABLE <Ramp_STN_F_link> '''
if cur.tables(table = 'Ramp_STN_F_link').fetchone():
    cur.execute('DROP TABLE Ramp_STN_F_link')

cur.execute ( 'CREATE TABLE Ramp_STN_F_link (STNid integer,STN_F
integer,STN_F_start integer,STN_F_end integer,WISLR_F integer)')
cur.execute (" INSERT INTO Ramp_STN_F_link
(STNid,STN_F,STN_F_start,STN_F_end,WISLR_F) SELECT Ramp_STN_F.STNid,
Ramp_STN_F.STN_F,\
                link_link.STNstart,link_link.STNend,link_link.WISLRid FROM
Ramp_STN_F LEFT JOIN link_link \
                ON Ramp_STN_F.STN_F = link_link.STNid WHERE
link_link.Record_Historic IS NULL AND link_link.M IS NULL")

conn.commit()

''' IDENTIFY THE VERY LAST SEGMENT OF STN_F AND INSERT INTO TABLE
<Ramp_STN_F_only>'''

S = [ i for i in cur.execute( 'SELECT * FROM Ramp_STN_F_link ORDER BY STNid,
STN_F, STN_F_start ')]
S_L = [max(g, key = itemgetter(3)) for z, g in groupby(S, key =
itemgetter(0,1))]
print 'L = ', len(S)
print 'S_L = ', len (S_L)
```

```
if cur.tables(table = 'Ramp_STN_F_only').fetchone():
    cur.execute('DROP TABLE Ramp_STN_F_only')
cur.execute ( 'CREATE TABLE Ramp_STN_F_only (STNid integer,STN_F
integer,STN_F_start integer,STN_F_end integer,WISLR_F integer)')

for i in S_L:
    cur.execute (" INSERT INTO Ramp_STN_F_only
(STNid,STN_F,STN_F_start,STN_F_end,WISLR_F)\
                    VALUES ( ?, ?, ?, ?, ?);",(i[0], i[1], i[2], i[3], i[4]))

conn.commit()

''' NAME THE RAMP NAME FOLLOWING STNID'''
if cur.tables(table = 'Ramp_STN_F_name').fetchone():
    cur.execute('DROP TABLE Ramp_STN_F_name')

cur.execute ( 'CREATE TABLE Ramp_STN_F_name (STNid integer,STN_F
integer,WISLR_F integer,WISLR_F_name string)')
conn.commit()



cur.execute (" INSERT INTO Ramp_STN_F_name
(STNid,STN_F,WISLR_F,WISLR_F_name)SELECT
Ramp_STN_F_only.STNid,Ramp_STN_F_only.STN_F,\
                Ramp_STN_F_only.WISLR_F,(DT_ST_PRTE_OVLY_LINE.ST_PRMY + ' ' +
DT_ST_PRTE_OVLY_LINE.ST_LABL_NM) AS COL1\
                FROM Ramp_STN_F_only LEFT JOIN DT_ST_PRTE_OVLY_LINE ON
Ramp_STN_F_only.WISLR_F = DT_ST_PRTE_OVLY_LINE.RDWY_LINK\
                WHERE DT_ST_PRTE_OVLY_LINE.ST_PRMY IS NOT NULL AND
DT_ST_PRTE_OVLY_LINE.ST_LABL_NM IS NOT NULL")

cur.execute (" INSERT INTO Ramp_STN_F_name (STNid,STN_F,WISLR_F,WISLR_F_name)
SELECT Ramp_STN_F_only.STNid,Ramp_STN_F_only.STN_F,\
                Ramp_STN_F_only.WISLR_F,FROM Ramp_STN_F_only LEFT JOIN
DT_ST_PRTE_OVLY_LINE ON Ramp_STN_F_only.WISLR_F =
DT_ST_PRTE_OVLY_LINE.RDWY_LINK\
                WHERE DT_ST_PRTE_OVLY_LINE.ST_PRMY IS NOT NULL AND
DT_ST_PRTE_OVLY_LINE.ST_LABL_NM IS NULL")

cur.execute (" INSERT INTO Ramp_STN_F_name
(STNid,STN_F,WISLR_F,WISLR_F_name)SELECT
Ramp_STN_F_only.STNid,Ramp_STN_F_only.STN_F,\
                Ramp_STN_F_only.WISLR_F,DT_ST_PRTE_OVLY_LINE.ST_LABL_NM FROM
Ramp_STN_F_only\
                LEFT JOIN DT_ST_PRTE_OVLY_LINE ON Ramp_STN_F_only.WISLR_F =
DT_ST_PRTE_OVLY_LINE.RDWY_LINK\
                WHERE DT_ST_PRTE_OVLY_LINE.ST_PRMY IS NULL AND
DT_ST_PRTE_OVLY_LINE.ST_LABL_NM IS NOT NULL")
conn.commit()

''' INSERT ASSOCIATED WISLR ID INTO NAME TABLE'''

if cur.tables(table = 'Ramp_STN_WISLR_F_name').fetchone():
    cur.execute('DROP TABLE Ramp_STN_WISLR_F_name')
```

```
cur.execute ( 'CREATE TABLE Ramp_STN_WISLR_F_name (STNid integer,WISLRid
integer,STN_F integer,WISLR_F integer,WISLR_F_name string)')
cur.execute (" INSERT INTO Ramp_STN_WISLR_F_name
(STNid,WISLRid,STN_F,WISLR_F,WISLR_F_name)SELECT Ramp_STN_F_name.STNid,\

link_link.WISLRid,Ramp_STN_F_name.STN_F,Ramp_STN_F_name.WISLR_F,Ramp_STN_F_na
me.WISLR_F_name\
                FROM Ramp_STN_F_name LEFT JOIN link_link ON
Ramp_STN_F_name.STNid = link_link.STNid WHERE link_link.Record_Historic IS
NULL\
                AND link_link.M IS NULL")

''' INSERT RAMP WISLR NAME'''

if cur.tables(table = 'Ramp_STN_WISLR_name_F_name').fetchone():
    cur.execute('DROP TABLE Ramp_STN_WISLR_name_F_name')

cur.execute ( 'CREATE TABLE Ramp_STN_WISLR_name_F_name (STNid integer,WISLRid
integer,WISLR_name string,STN_F integer,WISLR_F integer,WISLR_F_name
string)')
cur.execute (" INSERT INTO Ramp_STN_WISLR_name_F_name
(STNid,WISLRid,WISLR_name,STN_F,WISLR_F,WISLR_F_name)\
               SELECT
Ramp_STN_WISLR_F_name.STNid,Ramp_STN_WISLR_F_name.WISLRid,DT_ST_PRTE_OVLY_LIN
E.ST_LABL_NM,Ramp_STN_WISLR_F_name.STN_F,\

Ramp_STN_WISLR_F_name.WISLR_F,Ramp_STN_WISLR_F_name.WISLR_F_name FROM
Ramp_STN_WISLR_F_name LEFT JOIN DT_ST_PRTE_OVLY_LINE\
              ON Ramp_STN_WISLR_F_name.WISLRid =
DT_ST_PRTE_OVLY_LINE.RDWY_LINK")

'''USE STN NAME'''
if cur.tables(table = 'Ramp_STN_WISLR_name_F_name_STN').fetchone():
    cur.execute('DROP TABLE Ramp_STN_WISLR_name_F_name_STN')

cur.execute ( 'CREATE TABLE Ramp_STN_WISLR_name_F_name_STN (STNid
integer,WISLRid integer,WISLR_name string,STN_F integer,STN_F_name string,\
                                                    WISLR_F
integer,WISLR_F_name string)')
cur.execute (" INSERT INTO Ramp_STN_WISLR_name_F_name_STN
(STNid,WISLRid,WISLR_name,STN_F,STN_F_name,WISLR_F,WISLR_F_name)\
            SELECT Ramp_STN_WISLR_name_F_name.STNid,
Ramp_STN_WISLR_name_F_name.WISLRid, Ramp_STN_WISLR_name_F_name.WISLR_name,\

Ramp_STN_WISLR_name_F_name.STN_F,rdwy_rte_link.STN_Name,Ramp_STN_WISLR_name_F
_name.WISLR_F,Ramp_STN_WISLR_name_F_name.WISLR_F_name\
            FROM Ramp_STN_WISLR_name_F_name LEFT JOIN rdwy_rte_link ON
rdwy_rte_link.STNid = Ramp_STN_WISLR_name_F_name.STN_F \
            WHERE rdwy_rte_link.STUS = 'C' ")
conn.commit()

''' SELECT WISLR NOT HAVING START NAME RAMP'''
Ramp_F_1 = [ i for i in cur.execute("SELECT * FROM
Ramp_STN_WISLR_name_F_name_STN ORDER BY WISLRid, WISLR_F, STN_F")]
Ramp_F_2 = []
Ramp_F_3 = [min(k, key = itemgetter(5,3)) for z, k in groupby( Ramp_F_1, key
= itemgetter(1))]
```

```
''' MAKE FROM TABLE Ramp_F_name'''
if cur.tables(table = 'Ramp_F_name').fetchone():
    cur.execute('DROP TABLE Ramp_F_name')
cur.execute ( 'CREATE TABLE Ramp_F_name (STNid integer,WISLRid integer,STN_F
integer,STN_F_name string,WISLR_F integer,WISLR_F_name string)')
conn.commit()

for i in Ramp_F_3:
    cur.execute (' INSERT INTO Ramp_F_name
(STNid,WISLRid,STN_F,STN_F_name,WISLR_F,WISLR_F_name)\
                    VALUES (?, ?, ?, ?, ?, ?);',(i[0], i[1], i[3], i[4],
i[5], i[6]))
conn.commit()
print 'done'

#================================================================================
'''STN IN RAMP LIST NOT IN FROM TABLE Ramp_F_name'''
#================================================================================
if cur.tables(table = 'STN_Extra').fetchone():
    cur.execute('DROP TABLE STN_Extra')

cur.execute ( 'CREATE TABLE STN_Extra (STNid integer)')
cur.execute ( 'INSERT INTO STN_Extra (STNid)SELECT Ramp_STN.STNid FROM
Ramp_STN LEFT JOIN Ramp_F_name\
            ON Ramp_STN.STNid = Ramp_F_name.STNid WHERE Ramp_F_name.STNid IS
NULL')
conn.commit()

if cur.tables(table = 'STN_Extra_link').fetchone():
    cur.execute('DROP TABLE STN_Extra_link')
cur.execute ( 'CREATE TABLE STN_Extra_link (STNid integer, WISLRid integer)')
cur.execute ( 'INSERT INTO STN_Extra_link (STNid, WISLRid) SELECT
STN_Extra.STNid, link_link.WISLRid\
                FROM STN_Extra LEFT JOIN link_link ON STN_Extra.STNid =
link_link.STNid\
                WHERE link_link.Record_Historic IS NULL')
conn.commit()

'''WISLR CALLED RAMP IN WISLR BUT NOT INCLUDED IN LIST'''
Ramp_F_W1 = [ i for i in (cur.execute( 'SELECT
DT_ST_PRTE_OVLY_LINE.RDWY_LINK, DT_ST_PRTE_OVLY_LINE.ST_LABL_NM\
                                        FROM    DT_ST_PRTE_OVLY_LINE'))]
for i in Ramp_F_W1:
    if i[1][:4] == 'Ramp':
        cur.execute ( 'INSERT INTO STN_Extra_link (WISLRid) VALUES (?);',
(i[0]))
        conn.commit()

'''All Extra WISLR '''
Ramp_F_W2 = [ i for i in (cur.execute(' SELECT DISTINCT WISLRid FROM
STN_Extra_link '))]

Ramp_F_W_a = [ i for i in (cur.execute(' SELECT * FROM Ramp_F_name'))]
Ramp_F_W_b = [ ]
for i in Ramp_F_W_a:
    if i[3][:4] <> 'Ramp':
```

```
        Ramp_F_W_b.append(i[1])
Ramp_F_W3 = list (set (Ramp_F_W_b))


#Ramp_F_W3 = [ i for i in (cur.execute(' SELECT DISTINCT WISLRid FROM
Ramp_F_name'))]

if cur.tables(table = 'WISLR_Extra').fetchone():
    cur.execute('DROP TABLE WISLR_Extra')

cur.execute ( 'CREATE TABLE WISLR_Extra(WISLRid integer,Flag string)')

for i in Ramp_F_W2:
    if i not in Ramp_F_W3:
        cur.execute ('INSERT INTO WISLR_Extra (WISLRid, Flag) VALUES (?,
?);',(i[0], '1'))
        conn.commit()

#================================================================================
'''WISLR_F FOR EXTRA WISLR '''
#================================================================================
if cur.tables(table = 'WISLR_Extra_F').fetchone():
    cur.execute('DROP TABLE WISLR_Extra_F')

cur.execute ( 'CREATE TABLE WISLR_Extra_F(  WISLRid integer,REF_SITE_F
integer,REF_SITE_T integer,Azimuth_F float)')
cur.execute (" INSERT INTO
WISLR_Extra_F(WISLRid,REF_SITE_F,REF_SITE_T,Azimuth_F)\
                SELECT
WISLR_Extra.WISLRid,WISLR_Links.REF_SITE_F,WISLR_Links.REF_SITE_T,WISLR_Links
.Azimuth_F\
                FROM WISLR_Extra LEFT JOIN WISLR_Links ON WISLR_Extra.WISLRid
= WISLR_Links.RDWY_LINK\
                WHERE WISLR_Links.Status = 'C'")
conn.commit()

if cur.tables(table = 'WISLR_Extra_F_WISLR').fetchone():
    cur.execute('DROP TABLE WISLR_Extra_F_WISLR')
cur.execute ( 'CREATE TABLE WISLR_Extra_F_WISLR(WISLRid integer, REF_SITE_F
integer, Azimuth_F float, WISLRid_F integer,\
                Azimuth_T float, Difference float)')

cur.execute (" INSERT INTO WISLR_Extra_F_WISLR (WISLRid, REF_SITE_F,
Azimuth_F, WISLRid_F, Azimuth_T)\
            SELECT WISLR_Extra_F.WISLRid, WISLR_Extra_F.REF_SITE_F,
WISLR_Extra_F.Azimuth_F,WISLR_Links.RDWY_LINK,\
            WISLR_Links.Azimuth_T FROM WISLR_Extra_F LEFT JOIN WISLR_Links ON
WISLR_Extra_F.REF_SITE_F = WISLR_Links.REF_SITE_T\
            WHERE WISLR_Links.Status = 'C' AND WISLR_Extra_F.REF_SITE_T <>
WISLR_Links.REF_SITE_F")
conn.commit()

cur.execute( 'UPDATE WISLR_Extra_F_WISLR SET Difference = abs ( Azimuth_F -
Azimuth_T)')
conn.commit()
```

```
Name_F1 = [i for i in cur.execute(" SELECT  WISLR_Extra_F_WISLR.WISLRid,
WISLR_Extra_F_WISLR.WISLRid_F,\
                                   WISLR_Extra_F_WISLR.Difference,
DT_ST_PRTE_OVLY_LINE.ST_PRMY,\
                                   DT_ST_PRTE_OVLY_LINE.ST_LABL_NM FROM
WISLR_Extra_F_WISLR LEFT JOIN DT_ST_PRTE_OVLY_LINE\
                                   ON WISLR_Extra_F_WISLR.WISLRid_F =
DT_ST_PRTE_OVLY_LINE.RDWY_LINK")]
Name_F2 = [min(g, key = itemgetter(2)) for z, g in groupby(Name_F1, key=
itemgetter(0))]
if cur.tables(table = 'Ramp_F_W_name').fetchone():
    cur.execute('DROP TABLE Ramp_F_W_name')

cur.execute ( 'CREATE TABLE Ramp_F_W_name (WISLRid integer, WISLR_F integer,
WISLR_F_name_pre string,\
                                          WISLR_F_name_post string, Flag
string)')
conn.commit()

for i in Name_F2:
    cur.execute (' INSERT INTO Ramp_F_W_name (WISLRid,WISLR_F,
WISLR_F_name_pre, WISLR_F_name_post, Flag)\
                           VALUES (?, ?, ?, ?, ?);', (i[0], i[1], i[3],
i[4], '1'))
conn.commit()

'''All RAMP FROM WISLR '''
if cur.tables(table = 'Ramp_FROM').fetchone():
    cur.execute('DROP TABLE Ramp_FROM')

cur.execute ( 'CREATE TABLE Ramp_FROM (WISLRid integer, WISLR_F integer,
WISLR_F_name string,Flag string, STN_F_name string)')
conn.commit()

cur.execute (' INSERT INTO Ramp_FROM (  WISLRid, WISLR_F, WISLR_F_name,
STN_F_name) SELECT Ramp_F_name.WISLRid,\
             Ramp_F_name.WISLR_F, Ramp_F_name.WISLR_F_name,
Ramp_F_name.STN_F_name FROM Ramp_F_name\
            WHERE Ramp_F_name.WISLRid IS NOT NULL')

cur.execute (" INSERT INTO Ramp_FROM (  WISLRid,WISLR_F, WISLR_F_name, Flag)
SELECT Ramp_F_W_name.WISLRid,\
            Ramp_F_W_name.WISLR_F,
Ramp_F_W_name.WISLR_F_name_pre,Ramp_F_W_name.Flag FROM Ramp_F_W_name\
            WHERE Ramp_F_W_name.WISLR_F_name_post IS NULL")

cur.execute (" INSERT INTO Ramp_FROM (WISLRid,WISLR_F,WISLR_F_name,Flag)
SELECT Ramp_F_W_name.WISLRid,\
            Ramp_F_W_name.WISLR_F, Ramp_F_W_name.WISLR_F_name_post,
Ramp_F_W_name.Flag FROM Ramp_F_W_name\
            WHERE Ramp_F_W_name.WISLR_F_name_pre IS NULL")

cur.execute (" INSERT INTO Ramp_FROM (  WISLRid,WISLR_F,WISLR_F_name,Flag)
SELECT Ramp_F_W_name.WISLRid,\
            Ramp_F_W_name.WISLR_F,(Ramp_F_W_name.WISLR_F_name_pre + ' ' +
Ramp_F_W_name.WISLR_F_name_post) AS COL1,\
```

```
            Ramp_F_W_name.Flag FROM Ramp_F_W_name WHERE
Ramp_F_W_name.WISLR_F_name_pre IS NOT NULL\
            AND Ramp_F_W_name.WISLR_F_name_post IS NOT NULL")
conn.commit()
cur.execute ('  DELETE FROM Ramp_FROM WHERE  Ramp_FROM.WISLRid =
Ramp_FROM.WISLR_F')
conn.commit()

'''CONNECTING RAMP INDENTIFYING'''
X = [ i for i in cur.execute ('SELECT * FROM Ramp_FROM ORDER BY WISLRid')]
X1 = [i for i in X if i[2][:4] == 'Ramp']
X2 = [i for i in X if i[2][:4] != 'Ramp']

k = 0
while k<2:
    for i in X1:
        for j in X2:
            if i[1] == j[0]:
                X2.append ((i[0], i[1], j[2], i[3], i[4]))
    k = k + 1

X3 = sorted (X2, key = itemgetter(0))
X4 = [min(g, key = itemgetter(0)) for z, g in groupby(X3, key =
itemgetter(0))]

if cur.tables(table = 'Name_From').fetchone():
    cur.execute ('DROP TABLE Name_From')

cur.execute ('CREATE TABLE Name_From (  WISLRid integer, WISLRid_F integer,
WISLR_F_Name string, Flag string, STN_F_Name string)')
for i in X4:
    cur.execute (" INSERT INTO Name_From (WISLRid, WISLRid_F, WISLR_F_Name,
Flag, STN_F_Name) \
                VALUES (?, ?, ?, ?, ?);",( i[0], i[1], i[2],i[3], i[4] ))
    conn.commit()
conn.commit()


''' MANAGE TO WISLR #### TABLE <Ramp_STN_T> INSERT STN_T'''
if cur.tables(table = 'Ramp_STN_T').fetchone():
    cur.execute('DROP TABLE Ramp_STN_T')

cur.execute ( 'CREATE TABLE Ramp_STN_T (STNid integer, STN_SITE_T integer,
STN_T integer)')

cur.execute (" INSERT INTO Ramp_STN_T(STNid, STN_SITE_T, STN_T) SELECT
Ramp_STN_SITE_F_T.STNid, Ramp_STN_SITE_F_T.STN_SITE_T,\
            STN_Sites.STNid FROM Ramp_STN_SITE_F_T LEFT JOIN STN_Sites ON
Ramp_STN_SITE_F_T.STN_SITE_T = STN_Sites.REF_SITE_F\
            WHERE STN_Sites.STUS = 'C'")
conn.commit()

''' TABLE <Ramp_STN_T_link> '''
if cur.tables(table = 'Ramp_STN_T_link').fetchone():
    cur.execute('DROP TABLE Ramp_STN_T_link')
```

```
cur.execute ( 'CREATE TABLE Ramp_STN_T_link (STNid integer, STN_T integer,
STN_T_start integer, STN_T_end integer, WISLR_T integer)')

cur.execute (" INSERT INTO Ramp_STN_T_link (STNid, STN_T, STN_T_start,
STN_T_end, WISLR_T) SELECT Ramp_STN_T.STNid, Ramp_STN_T.STN_T,\
            link_link.STNstart, link_link.STNend, link_link.WISLRid FROM
Ramp_STN_T LEFT JOIN link_link\
            ON Ramp_STN_T.STN_T = link_link.STNid WHERE
link_link.Record_Historic IS NULL")
conn.commit()

''' IDENTIFY THE VERY First SEGMENT OF STN_T AND INSERT INTO TABLE
<Ramp_STN_T_only>'''
E = [ i for i in cur.execute( 'SELECT * FROM Ramp_STN_T_link ORDER BY STNid,
STN_T, STN_T_start ')]
E_F = [min(g, key = itemgetter(2)) for z, g in groupby(E, key =
itemgetter(0,1))]

if cur.tables(table = 'Ramp_STN_T_only').fetchone():
    cur.execute('DROP TABLE Ramp_STN_T_only')

cur.execute ( 'CREATE TABLE Ramp_STN_T_only (STNid integer, STN_T integer,
STN_T_start integer, STN_T_end integer, WISLR_T integer)')

for i in E_F:
    cur.execute (" INSERT INTO Ramp_STN_T_only (STNid, STN_T, STN_T_start,
STN_T_end, WISLR_T)\
                VALUES ( ?, ?, ?, ?, ?);", (i[0], i[1], i[2], i[3], i[4]))
conn.commit()

''' NAME THE RAMP NAME FOLLOWING STNID'''
if cur.tables(table = 'Ramp_STN_T_name').fetchone():
    cur.execute('DROP TABLE Ramp_STN_T_name')

cur.execute ( 'CREATE TABLE Ramp_STN_T_name (STNid integer, STN_T integer,
WISLR_T integer, WISLR_T_name string)')
conn.commit()

cur.execute (" INSERT INTO Ramp_STN_T_name (STNid, STN_T, WISLR_T,
WISLR_T_name) SELECT Ramp_STN_T_only.STNid,\
            Ramp_STN_T_only.STN_T, Ramp_STN_T_only.WISLR_T,
(DT_ST_PRTE_OVLY_LINE.ST_PRMY + ' ' + DT_ST_PRTE_OVLY_LINE.ST_LABL_NM) AS
COL1\
            FROM Ramp_STN_T_only LEFT JOIN DT_ST_PRTE_OVLY_LINE ON
Ramp_STN_T_only.WISLR_T = DT_ST_PRTE_OVLY_LINE.RDWY_LINK\
            WHERE DT_ST_PRTE_OVLY_LINE.ST_PRMY IS NOT NULL AND
DT_ST_PRTE_OVLY_LINE.ST_LABL_NM IS NOT NULL")
cur.execute (" INSERT INTO Ramp_STN_T_name (STNid, STN_T, WISLR_T,
WISLR_T_name) SELECT Ramp_STN_T_only.STNid,\
            Ramp_STN_T_only.STN_T, Ramp_STN_T_only.WISLR_T,
DT_ST_PRTE_OVLY_LINE.ST_PRMY FROM Ramp_STN_T_only\
            LEFT JOIN DT_ST_PRTE_OVLY_LINE ON Ramp_STN_T_only.WISLR_T =
DT_ST_PRTE_OVLY_LINE.RDWY_LINK\
            WHERE DT_ST_PRTE_OVLY_LINE.ST_PRMY IS NOT NULL AND
DT_ST_PRTE_OVLY_LINE.ST_LABL_NM IS NULL")
cur.execute (" INSERT INTO Ramp_STN_T_name (STNid, STN_T, WISLR_T,
WISLR_T_name)SELECT Ramp_STN_T_only.STNid,\
```

```
           Ramp_STN_T_only.STN_T, Ramp_STN_T_only.WISLR_T,
DT_ST_PRTE_OVLY_LINE.ST_LABL_NM FROM Ramp_STN_T_only\
           LEFT JOIN DT_ST_PRTE_OVLY_LINE ON Ramp_STN_T_only.WISLR_T =
DT_ST_PRTE_OVLY_LINE.RDWY_LINK\
           WHERE DT_ST_PRTE_OVLY_LINE.ST_PRMY IS NULL AND
DT_ST_PRTE_OVLY_LINE.ST_LABL_NM IS NOT NULL")
conn.commit()

''' INSERT ASSOCIATED WISLR ID TO) INTO NAME TABLE'''
if cur.tables(table = 'Ramp_STN_WISLR_T_name').fetchone():
    cur.execute('DROP TABLE Ramp_STN_WISLR_T_name')

cur.execute ( 'CREATE TABLE Ramp_STN_WISLR_T_name (STNid integer, WISLRid
integer, STN_T integer, WISLR_T integer, WISLR_T_name string)')
cur.execute (" INSERT INTO Ramp_STN_WISLR_T_name  (STNid, WISLRid,STN_T,
WISLR_T, WISLR_T_name)\
            SELECT Ramp_STN_T_name.STNid, link_link.WISLRid,
Ramp_STN_T_name.STN_T, Ramp_STN_T_name.WISLR_T, Ramp_STN_T_name.WISLR_T_name\
            FROM  Ramp_STN_T_name LEFT JOIN link_link ON
Ramp_STN_T_name.STNid = link_link.STNid WHERE link_link.Record_Historic IS
NULL")

''' INSERT RAMP WISLR NAME'''
if cur.tables(table = 'Ramp_STN_WISLR_name_T_name').fetchone():
    cur.execute('DROP TABLE Ramp_STN_WISLR_name_T_name')

cur.execute ( 'CREATE TABLE Ramp_STN_WISLR_name_T_name (STNid integer,
WISLRid integer, WISLR_name string,STN_T integer,\
                WISLR_T integer, WISLR_T_name string)')
cur.execute (" INSERT INTO Ramp_STN_WISLR_name_T_name (STNid, WISLRid,
WISLR_name, STN_T, WISLR_T, WISLR_T_name)\
            SELECT Ramp_STN_WISLR_T_name.STNid,
Ramp_STN_WISLR_T_name.WISLRid, DT_ST_PRTE_OVLY_LINE.ST_LABL_NM,\
            Ramp_STN_WISLR_T_name.STN_T, Ramp_STN_WISLR_T_name.WISLR_T,
Ramp_STN_WISLR_T_name.WISLR_T_name\
            FROM Ramp_STN_WISLR_T_name LEFT JOIN DT_ST_PRTE_OVLY_LINE ON
Ramp_STN_WISLR_T_name.WISLRid  = DT_ST_PRTE_OVLY_LINE.RDWY_LINK")
conn.commit()

if cur.tables(table = 'Ramp_STN_WISLR_name_T_name_STN').fetchone():
    cur.execute('DROP TABLE Ramp_STN_WISLR_name_T_name_STN')

cur.execute ( 'CREATE TABLE Ramp_STN_WISLR_name_T_name_STN (STNid integer,
WISLRid integer, WISLR_name string, STN_T integer,\
            STN_T_name string, WISLR_T integer, WISLR_T_name string)')
cur.execute (" INSERT INTO Ramp_STN_WISLR_name_T_name_STN (STNid, WISLRid,
WISLR_name, STN_T, STN_T_name, WISLR_T, WISLR_T_name)\
            SELECT Ramp_STN_WISLR_name_T_name.STNid,
Ramp_STN_WISLR_name_T_name.WISLRid, Ramp_STN_WISLR_name_T_name.WISLR_name,\
            Ramp_STN_WISLR_name_T_name.STN_T, rdwy_rte_link.STN_Name,
Ramp_STN_WISLR_name_T_name.WISLR_T,\
            Ramp_STN_WISLR_name_T_name.WISLR_T_name FROM
Ramp_STN_WISLR_name_T_name LEFT JOIN rdwy_rte_link\
            ON rdwy_rte_link.STNid = Ramp_STN_WISLR_name_T_name.STN_T WHERE
rdwy_rte_link.STUS = 'C' ")
conn.commit()
```

```
''' SELECT WISLR NOT HAVING TO NAME RAMP'''
Ramp_T_1 = [ i for i in cur.execute("SELECT * FROM
Ramp_STN_WISLR_name_T_name_STN ORDER BY WISLRid, WISLR_T")]
Ramp_T_2 = []
Ramp_T_3 = [min(k, key = itemgetter(5,3)) for z, k in groupby( Ramp_T_1, key
= itemgetter(1))]

''' MAKE TABLE Ramp_T_name'''
if cur.tables(table = 'Ramp_T_name').fetchone():
    cur.execute('DROP TABLE Ramp_T_name')

cur.execute ( 'CREATE TABLE Ramp_T_name (STNid integer, WISLRid integer,
STN_T integer, STN_T_name string,\
            WISLR_T integer, WISLR_T_name string)')
conn.commit()
for i in Ramp_T_3:
    cur.execute (' INSERT INTO Ramp_T_name (STNid, WISLRid, STN_T,
STN_T_name, WISLR_T, WISLR_T_name)\
                    VALUES (?, ?, ?, ?, ?, ?);', (i[0], i[1], i[3], i[4],
i[5], i[6]))
conn.commit()

'''STN IN RAMP LIST NOT IN TO TABLE Ramp_T_name'''
if cur.tables(table = 'STN_Extra_T').fetchone():
    cur.execute('DROP TABLE STN_Extra_T')

cur.execute ( 'CREATE TABLE STN_Extra_T (STNid integer)')
cur.execute ( 'INSERT INTO STN_Extra_T (STNid) SELECT Ramp_STN.STNid FROM
Ramp_STN LEFT JOIN Ramp_STN_WISLR_name_T_name\
            ON Ramp_STN.STNid = Ramp_STN_WISLR_name_T_name.STNid WHERE
Ramp_STN_WISLR_name_T_name.STNid IS NULL')
conn.commit()

if cur.tables(table = 'STN_Extra_link_T').fetchone():
    cur.execute('DROP TABLE STN_Extra_link_T')

cur.execute ( 'CREATE TABLE STN_Extra_link_T (STNid integer, WISLRid
integer)')
cur.execute ( 'INSERT INTO STN_Extra_link_T (STNid, WISLRid) SELECT
STN_Extra_T.STNid, link_link.WISLRid\
            FROM STN_Extra_T LEFT JOIN link_link ON STN_Extra_T.STNid =
link_link.STNid WHERE link_link.Record_Historic IS NULL')
conn.commit()

'''WISLR CALLED RAMP IN WISLR BUT NOT INCLUDED IN LIST T'''
Ramp_T_W1 = [ i for i in (cur.execute( 'SELECT
DT_ST_PRTE_OVLY_LINE.RDWY_LINK, DT_ST_PRTE_OVLY_LINE.ST_LABL_NM\
                                        FROM    DT_ST_PRTE_OVLY_LINE'))]
for i in Ramp_T_W1:
    if i[1][:4] == 'Ramp':
        cur.execute ( 'INSERT INTO STN_Extra_link_T (WISLRid) VALUES (?);',
(i[0]))
        conn.commit()

'''All Extra WISLR T'''
Ramp_T_W2 = [ i for i in (cur.execute(' SELECT DISTINCT WISLRid FROM
STN_Extra_link_T '))]
```

```
Ramp_T_W_a = [ i for i in cur.execute( ' SELECT * FROM Ramp_T_name ')]
Ramp_T_W_b = [ ]

for i in Ramp_T_W_a:
    if i[3][:4] <> 'Ramp':
        Ramp_T_W_b.append(i[1])
Ramp_T_W3 = list (set (Ramp_T_W_b))

if cur.tables(table = 'WISLR_Extra_T').fetchone():
    cur.execute('DROP TABLE WISLR_Extra_T')
conn.commit()

cur.execute ( 'CREATE TABLE WISLR_Extra_T(WISLRid integer,Flag string)')
conn.commit()

for i in Ramp_T_W2:
    if i not in Ramp_T_W3:
        cur.execute ('INSERT INTO WISLR_Extra_T (WISLRid, Flag VALUES (?,
?);', (i[0], '1'))
        conn.commit()

'''WISLR_T_T FOR EXTRA WISLR_T '''
if cur.tables(table = 'WISLR_Extra_T_T').fetchone():
    cur.execute('DROP TABLE WISLR_Extra_T_T')

cur.execute ( 'CREATE TABLE WISLR_Extra_T_T(  WISLRid integer,REF_SITE_F
integer,REF_SITE_T integer,Azimuth_T float)')
cur.execute (" INSERT INTO
WISLR_Extra_T_T(WISLRid,REF_SITE_F,REF_SITE_T,Azimuth_T)\
            SELECT
WISLR_Extra_T.WISLRid,WISLR_Links.REF_SITE_F,WISLR_Links.REF_SITE_T,WISLR_Lin
ks.Azimuth_T\
            FROM WISLR_Extra_T LEFT JOIN WISLR_Links ON WISLR_Extra_T.WISLRid
= WISLR_Links.RDWY_LINK\
            WHERE           WISLR_Links.Status = 'C'")
conn.commit()

if cur.tables(table = 'WISLR_Extra_T_WISLR').fetchone():
    cur.execute('DROP TABLE WISLR_Extra_T_WISLR')
cur.execute ( 'CREATE TABLE WISLR_Extra_T_WISLR(WISLRid integer, REF_SITE_T
integer, Azimuth_T float,WISLRid_T integer,\
                                                Azimuth_F float,Difference
float)')

cur.execute (" INSERT INTO WISLR_Extra_T_WISLR (WISLRid,
REF_SITE_T,Azimuth_T,WISLRid_T,Azimuth_F)\
            SELECT WISLR_Extra_T_T.WISLRid, WISLR_Extra_T_T.REF_SITE_F,
WISLR_Extra_T_T.Azimuth_T,\
            WISLR_Links.RDWY_LINK,WISLR_Links.Azimuth_F FROM WISLR_Extra_T_T
LEFT JOIN WISLR_Links\
            ON WISLR_Extra_T_T.REF_SITE_T = WISLR_Links.REF_SITE_F WHERE
WISLR_Links.Status = 'C'\
            AND WISLR_Extra_T_T.REF_SITE_F <> WISLR_Links.REF_SITE_T")
conn.commit()

cur.execute( 'UPDATE WISLR_Extra_T_WISLR SET Difference = abs ( Azimuth_F -
Azimuth_T)')
```

```
conn.commit()

Name_T1 = [i for i in cur.execute(" SELECT
WISLR_Extra_T_WISLR.WISLRid,WISLR_Extra_T_WISLR.WISLRid_T,\
                                    WISLR_Extra_T_WISLR.Difference,
DT_ST_PRTE_OVLY_LINE.ST_PRMY,\
                                    DT_ST_PRTE_OVLY_LINE.ST_LABL_NM FROM
WISLR_Extra_T_WISLR\
                                    LEFT JOIN DT_ST_PRTE_OVLY_LINE ON
WISLR_Extra_T_WISLR.WISLRid_T \
                                    = DT_ST_PRTE_OVLY_LINE.RDWY_LINK ORDER BY
WISLR_Extra_T_WISLR.WISLRid")]
Name_T2 = [min(g, key = itemgetter(2)) for z, g in groupby(Name_T1, key=
itemgetter(0))]

if cur.tables(table = 'Ramp_T_W_name').fetchone():
    cur.execute('DROP TABLE Ramp_T_W_name')
cur.execute ( 'CREATE TABLE Ramp_T_W_name (WISLRid integer,WISLR_T
integer,WISLR_T_name_pre string,WISLR_T_name_post string,\
                                        Flag string)')
conn.commit()
for i in Name_T2:
    cur.execute (' INSERT INTO Ramp_T_W_name
(WISLRid,WISLR_T,WISLR_T_name_pre,WISLR_T_name_post,Flag)\
                    VALUES (?, ?, ?, ?, ?);',(i[0], i[1], i[3], i[4], '1'))
conn.commit()

'''All T RAMP FROM WISLR '''
if cur.tables(table = 'Ramp_TO').fetchone():
    cur.execute('DROP TABLE Ramp_TO')
cur.execute ( 'CREATE TABLE Ramp_TO (WISLRid integer,WISLR_T
integer,WISLR_T_name string,Flag string,STN_T_name string)')
conn.commit()
cur.execute (' INSERT INTO Ramp_TO (
WISLRid,WISLR_T,WISLR_T_name,STN_T_name)SELECT Ramp_T_name.WISLRid,\

Ramp_T_name.WISLR_T,Ramp_T_name.WISLR_T_name,Ramp_T_name.STN_T_name FROM
Ramp_T_name')
conn.commit()
cur.execute (" INSERT INTO Ramp_TO (
WISLRid,WISLR_T,WISLR_T_name,Flag)SELECT Ramp_T_W_name.WISLRid,\

Ramp_T_W_name.WISLR_T,Ramp_T_W_name.WISLR_T_name_post,Ramp_T_W_name.Flag FROM
Ramp_T_W_name\
                WHERE Ramp_T_W_name.WISLR_T_name_pre IS NULL")
conn.commit()
cur.execute (" INSERT INTO Ramp_TO (
WISLRid,WISLR_T,WISLR_T_name,Flag)SELECT
Ramp_T_W_name.WISLRid,Ramp_T_W_name.WISLR_T,\
                Ramp_T_W_name.WISLR_T_name_pre,Ramp_T_W_name.Flag FROM
Ramp_T_W_name WHERE Ramp_T_W_name.WISLR_T_name_post IS NULL")
conn.commit()
cur.execute (" INSERT INTO Ramp_TO (
WISLRid,WISLR_T,WISLR_T_name,Flag)SELECT Ramp_T_W_name.WISLRid,\
                Ramp_T_W_name.WISLR_T,(Ramp_T_W_name.WISLR_T_name_pre + ' ' +
Ramp_T_W_name.WISLR_T_name_post) AS COL,\
```

```
                Ramp_T_W_name.Flag FROM Ramp_T_W_name WHERE
Ramp_T_W_name.WISLR_T_name_pre IS NOT NULL AND
Ramp_T_W_name.WISLR_T_name_post IS NOT NULL")
conn.commit()
cur.execute ('  DELETE FROM Ramp_TO\
                WHERE  Ramp_TO.WISLRid = Ramp_TO.WISLR_T')
conn.commit()

'''CONNECTING T RAMP INDENTIFYING'''
Y = [ i for i in cur.execute ('SELECT * FROM Ramp_TO ORDER BY WISLRid')]
Y1 = [i for i in Y if i[2][:4] == 'Ramp']
Y2 = [i for i in Y if i[2][:4] != 'Ramp']

p = 0
while p<2:
    for i in Y1:
        for j in Y2:
            if i[1] == j[0]:
                Y2.append ((i[0], i[1], j[2], i[3], i[4]))
    p = p + 1

Y3 = sorted (Y2, key = itemgetter(0))
Y4 = [min(g, key = itemgetter(0)) for z, g in groupby(Y3, key =
itemgetter(0))]

if cur.tables(table = 'Name_To').fetchone():
    cur.execute ('DROP TABLE Name_To')

cur.execute ('CREATE TABLE Name_To (  WISLRid integer,WISLRid_T
integer,WISLR_T_Name string,Flag string,STN_T_Name string)')
for i in Y4:
    cur.execute (" INSERT INTO Name_To (WISLRid, WISLRid_T, WISLR_T_Name,
Flag, STN_T_Name)\
                VALUES (?, ?, ?, ?, ?);",( i[0], i[1], i[2], i[3], i[4] ))
    conn.commit()
conn.commit()

# Combine from and to table
if cur.tables(table = 'Ramp_Name_All').fetchone():
    cur.execute('DROP TABLE Ramp_Name_All')

cur.execute ( 'CREATE TABLE Ramp_Name_All (WISLRid integer,WISLRid_F
integer,WISLRid_T integer,WISLR_F_Name string,\
            WISLR_T_Name string, Flag_F string,Flag_T string,STN_F_Name
string,STN_T_Name string)')
conn.commit()
cur.execute ( 'INSERT INTO Ramp_Name_All
(WISLRid,WISLRid_F,WISLRid_T,WISLR_F_Name,WISLR_T_Name,Flag_F,Flag_T,STN_F_Na
me,STN_T_Name)\
            SELECT
Name_From.WISLRid,Name_From.WISLRid_F,Name_To.WISLRid_T,Name_From.WISLR_F_Nam
e,Name_To.WISLR_T_Name,\

Name_From.Flag,Name_To.Flag,Name_From.STN_F_name,Name_To.STN_T_Name FROM
Name_From LEFT JOIN Name_To\
            ON Name_From.WISLRid = Name_To.WISLRid')
```

```
cur.execute ( 'INSERT INTO Ramp_Name_All
(WISLRid,WISLRid_T,WISLR_T_Name,Flag_T,STN_T_Name)SELECT
Name_To.WISLRid,Name_To.WISLRid_T,\
                Name_To.WISLR_T_Name,Name_To.Flag,Name_To.STN_T_Name FROM
Name_To LEFT JOIN   Name_From\
                ON Name_From.WISLRid = Name_To.WISLRid WHERE
Name_From.WISLRid IS NULL')
conn.commit()
cur.close()
conn.close()
```