2009-04-17

# Progressive Spatial Networks

Samuel Curren
*Brigham Young University - Provo*

# Progressive Spatial Networks:
# Learning from GPS tracklogs

by

Samuel Curren

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2009

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Samuel Curren

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

| | |
|---|---|
| Date | Phillip J. Windley, Chair |
| Date | Michael A. Goodrich |
| Date | Brandon S. Plewe |

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Samuel Curren in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

| | |
|---|---|
| Date | Phillip J. Windley<br>Chair, Graduate Committee |

Accepted for the Department

| | |
|---|---|
| Date | Kent E. Seamons<br>Graduate Coordinator |

Accepted for the College

| | |
|---|---|
| Date | Thomas W. Sederberg<br>Associate Dean, College of Physical and Mathematical Sciences |

# ABSTRACT

Digital street and trail maps are typically represented by an interconnected network of path segments. These spatial networks are used in map creation, route planning, and geo-location. Consumer GPS devices have become popular as a method of collecting data for use in spatial networks. Existing methods for creating spatial networks either require extensive hand editing or use inefficient algorithms that require re-computation when adding new data to an existing network. I demonstrate a method for creating and maintaining spatial networks that allows for incremental updates without complete re-computation. I also demonstrate how spatial limits on data set growth allows networks to be updated in linear time after initial path discovery. This approach allows networks to be rapidly and accurately updated using data from consumer GPS devices.

# Contents

**Contents** **v**

**1 Introduction** **1**

1.1 Thesis Statement . . . . . 2

**2 Related Work** **3**

2.1 Terminology . . . . . 3

2.2 Digital Trail Libraries . . . . . 3

2.3 Mining GPS Traces for Map Refinement . . . . . 5

2.4 A Theory of the Cartographic Line . . . . . 5

2.5 Bayesian Updates . . . . . 6

**3 Methodology** **8**

3.1 Variables . . . . . 9

3.2 Building the Network . . . . . 11

3.3 Update Process . . . . . 15

3.3.1 Intermediate Points . . . . . 15

3.3.2 Matching Existing Network Points . . . . . 16

3.3.3 Condition Checks . . . . . 18

3.3.4 Intersections . . . . . 20

3.4 Combining Points . . . . . 24

**4 Results** **26**

4.1 Situational Handling . . . . . 26

4.2 Algorithmic Analysis . . . . . 29
4.3 Numerical Comparison . . . . . 32
4.4 Speed . . . . . 34

**5 Analysis** **39**

5.1 Intermediate Data . . . . . 39
5.2 Spatial Limits . . . . . 40
5.3 Stream Processing . . . . . 40
5.4 Bayesian Updates . . . . . 41
5.5 Accuracy of Source Data . . . . . 41

**6 Conclusion and Future Work** **42**

6.1 Conclusion . . . . . 42
6.2 Future Work . . . . . 43

**A Debugging Tricks** **44**

A.1 Geographic Data Visualization . . . . . 44
A.2 Stream Processing . . . . . 46

**Bibliography** **48**

# Chapter 1

## Introduction

Spatial networks, also known as electronic maps, can be very useful in information systems, allowing such things as geo-location (finding the location of an address) and route planning (finding the fastest route between two locations). Electronic maps are typically represented as a path network or graph, where each segment joins other segments at each end.

Global Positioning System (GPS) receivers have become popular as a data collection device for spatial networks. GPS receivers use timestamped signals from orbiting satellites to determine the receiver's position on the earth's surface. Originally developed for military use, these receivers are now used in vehicle navigation systems, consumer receiver devices, and even cell phones. The decreasing cost of these devices allows them to be included in nearly every portable device. The number of GPS devices in use is exploding.

In addition to reporting current physical location, GPS receivers typically record "digital breadcrumbs" of the path the device has recently traveled. This log, called a 'trace' or a 'tracklog' is a series of timestamped latitude, longitude, and elevation data points. This tracklog can be downloaded onto a computer for use in mapping projects.

OpenStreetMap [3] is a project which demonstrate the usefulness of GPS tracklogs in mapping projects. OpenStreetMap allows users to add to and update street map data in a geographical wiki format. The system allows users to upload tracklogs

collected while traveling streets. Users can then edit the tracklogs by hand and add additional information such as street names to be used on the map. While tracklogs enable geographic data to be easily reused, hand editing tracklogs is very tedious.

An unsupervised method for combining GPS tracklogs into a map has been proposed by Morris et al. [7]. It relies on intersections between multiple tracklogs of the same trails to determine intersections and eliminate redundant tracklog segments. Unfortunately, their algorithm does not easily allow data to be incrementally added while maintaining the information of previously added tracklogs.

The need to update spatial networks with new data is being driven by the explosion of consumer GPS devices and devices with built in wireless cards. These devices will make large volumes of GPS data available in near real time. Having the ability to incrementally add data to an existing path network in a computationally efficient way is becoming an important requirement for mapping algorithms.

## 1.1 Thesis Statement

Spatial networks can be created and maintained in an efficient manner through the use of Bayesian-like update methods that update networks without requiring storage or re-computation of all source data. Using these update methods, the accuracy of the network will improve as additional data is added.

# Chapter 2

## Related Work

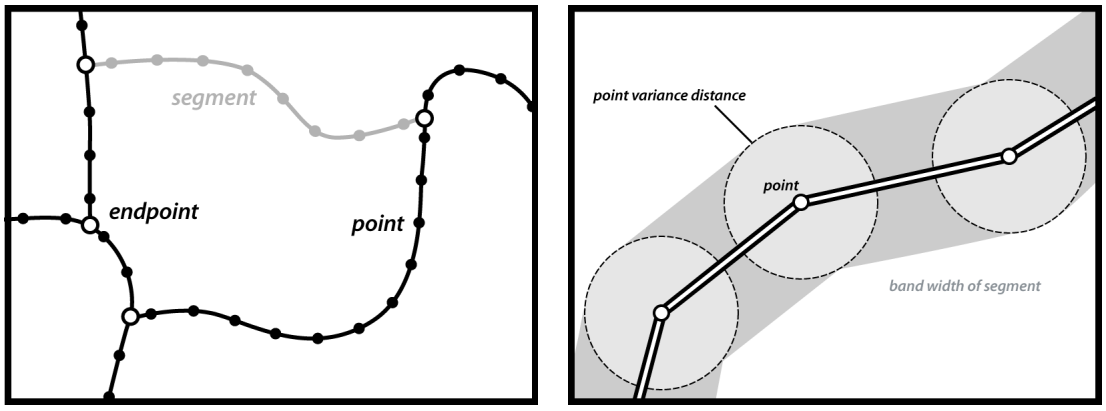
### 2.1 Terminology

A **network** in this paper refers to a collection of path segments. Path **segments** are a linear sequence of points, each **point** consisting of a latitude, longitude, and elevation. Each segment also has two **endpoints**, with each endpoint connecting to zero or more other endpoints. Endpoints can be considered nodes on the network graph, with segments connecting the nodes. Examples of these concepts can be seen in Figure 2.1(a).

Data retrieved from a GPS receiver is a single segment and is commonly referred to as a **tracklog**. The **band width** [8] is the variance of the segment, or estimation of accuracy of a point in a segment. **Point Variance Distance** is the band width of a segment at each point in the segment, as seen in Figure 2.1(b).

### 2.2 Digital Trail Libraries

In a paper titled "Digital Trail Libraries" [7], Scott Morris, Alan Morris, and Kobus Barnard propose an unsupervised method for reducing tracklogs into a path network. Their method relies on combining tracklogs into a graph, where segments are created where tracklogs intersect. This creates a graph with many small, and typically narrow, faces. They reduce the graph so that all faces with a width less then a reduction threshold are combined. The result of their algorithm is a path network with the




(a) Basic network composition (b) Point variance distance

Figure 2.1: Terminology Diagrams

same structure that this paper proposes, and so the lessons learned and their methods are very relevant to my research.

The main limitation of their method is the process for adding new data after the initial computation of the network. They claim that the addition can be made by running the algorithm again with the network and the new tracklog as source tracklogs, but this method has several flaws. Because no information is retained about the original tracklogs from which a segment is constructed, the algorithm improperly favors the newly added data.

An additional limitation is the case where there are not enough tracklogs of a particular trail segment to produce an intersection whenever there is data that should be combined. For example, consider the case where a hiker doubles back to pick up something lost along the trail, and then continues back up the trail after finding it. If the tracklog does not happen to intersect the original tracklog, then an awkward spur will be created in their trail network, instead of combining that data in with the original tracklog. An example of this situation be seen in Figure 2.2.

Because of these limitations, all original data must be retained to be used in the re-computation of the path network when new data is added.

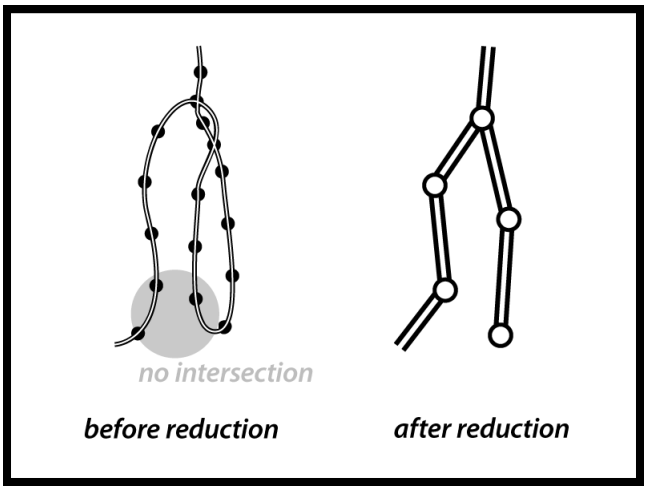



Figure 2.2: Close segments will not be combined unless there is an explicit intersection.

## 2.3 Mining GPS Traces for Map Refinement

In a paper titled "Mining GPS Traces for Map Refinement" [9], Stefan Schroedl et al. describe a method for using GPS traces to create detailed models of roads that include lane information and intersection travel paths. While their method also requires all source data for a re-computation, it introduces the concept of road width. They first establish a road center line using the source tracklogs. They then analyze the source tracklogs to find lanes by finding clusters of tracklogs with a regular offset from the road center line.

The concept of road width relates to the concept of the *band width* of a cartographic line. This concept is used in my algorithm to indicate the confidence in the position of a path segment.

## 2.4 A Theory of the Cartographic Line

The most fundamental work in this area is a 1976 paper entitled "A Theory of the Cartographic Line" [8], wherein Thomas Peucker (later changed to Poiker) describes the process of matching two lines by using the variance of the digitizing error, and

matching percentage of band overlap. He points out that there are two factors that cause two representations of the same line to differ: encoding noise and a difference in sampling points. Although the paper does not address GPS data, these two factors are precisely the same factors the exist in GPS tracklogs, and his ideas are directly applicable. He also talks in the paper about the concept of line band width, which is the width of a line computed during a generalization process. This concept of band width is precisely what I use in the process of tracklog matching and combination.

## 2.5 Bayesian Updates

The concept of Bayesian updates has been around for some time, and despite its muddied past [11], it has become a core foundational concept in computer science. Descriptions of Bayesian principles and examples of use can be found in modern machine learning and artificial intelligence textbooks [6] [10].

Bayes rule is commonly used to update the estimate of a value upon receiving additional readings of that value. It is very popular in systems where readings are noisy and accurate estimates are desired. It works by updating the probability of an event based on the previously calculated probability and the probability of receiving the data being evaluated.

If our hypothesis is the position of a point on the line, then the posterior estimate of the location of that point, given new data, is the product of the prior probability (variance of the existing point) and the likelihood function. The likelihood function is the probability of the new data containing a point, given that our estimate of the location of the existing point is correct.

While applying Bayes law in our research, we found that it reduced the accuracy estimate of the points's location much smaller and much faster then desired. We modified the equations to produce results that worked in our research. Consequently we do not use the actual Bayeisn update equations, nevertheless the concepts are core

to how the algorithm works. Our existing spatial network is our prior information, newly added tracklogs become our evidence, and the posterior estimate of point location is used to update and improve the path network itself. The actual equations used to update network data are described in section 3.4.

# Chapter 3

## Methodology

Existing methods [7] of creating spatial networks combine all source data into one large graph, and then perform reductions to simplify the graph and remove redundant segments.

Instead of creating the network from all input tracklogs simultaneously, I will apply the tracklogs to the existing network one at a time. Each tracklog is applied one point at a time in sequence. The existing network is updated based on the information provided by the new point. Adding new data one point at a time allows us to impose spatial limits on data set growth, resulting in a significant run-time advantage.

This method will allow a computationally fast way to add new data to a path network, with applications in mapping, agent path discovery (robots and search and rescue), and location based services.

A specific example of this algorithm is the process of discovering and adding new roads to maps. When a new road is constructed, mapping service companies typically send an agent to map the road with expensive industrial GPS equipment. By using the algorithm presented in this paper, the company could collect data from cheap consumer GPS navigation units in cars. The data from multiple cars can be added into their existing spatial network quickly and will become refined as more data is collected and processed. In addition to detecting new roads, the system would also be capable of detecting roads no longer in use and roads that have been rerouted during construction.

## 3.1 Variables

This algorithm has several parameters that can be adjusted to the characteristics of the source data. Points in our algorithm are considered to be an estimate of point location, and are stored as a probability distribution. Each point is represented by the mean and the variance distance of the distribution. The mean is stored as a longitude and latitude, and the variance distance is stored as the radius around the mean with a distance equal to two standard deviations of the distribution. I describe each variable below and the effect that it has upon the computation of the final spatial network.

The **Initial Point Variance Distance** is the band width [8] applied to points that are part of new network segments. The Point Variance Distance is used during the search for nearby points. New points within the Variance Distance of an existing point are typically combined with the existing point. New points that are not located within the variance distance of existing points are typically added as part of a new segment or an extension to an existing segment.

This value can be adjusted to control the minimum distance between segments in the computed network. Reducing this value will allow for more detailed networks. If the source data is fairly accurate this value can be quite smaller. If this value is set too small for the accuracy of the source data, then network segments will begin to appear that do not represent real trail segments, and should have been combined with other nearby segments.

The **Connected Search Factor** is a multiplier used to find a connected path between the previous point updated and the next point to be updated. If an existing path can be found with a path distance less then the product of the direct distance between the two points and the Connected Search Factor, then no additional segment is created linking the two points. If no existing path is found within this limit, then an intersection is created at the previous and next points, and a new segment is created linking them.

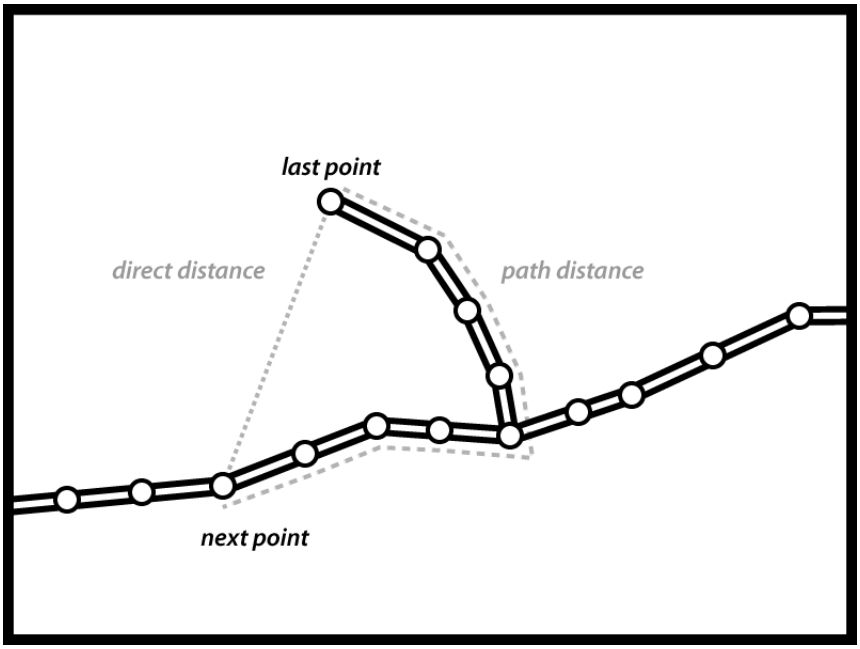



Figure 3.1: Connected Search Distance

The effects of this factor can be observed when data augmenting existing segments is added to the network. When the new data crosses an intersection, the Connected Search Factor prevents small connecting segments from being created. When this value is set too small, intersections between segments will start to turn into a mesh of connected segments as 'corner cutting' connecting segments are added. When set too large, splinter segments will begin to appear in places where the segment should be connected at both ends. Figure 3.1 shows the variables used when calculating the Connected Search Factor.

To preserve tracklog detail when adding new points, the **New Point Variance Factor** is used to shrink the variance distance of the most recently updated points. The effect of this factor is shown in Figure 3.2. This adjustment of the most recently updated point allows new points to be added that would normally be combined into previous points. This factor helps preserve the detail of the collected tracklogs when creating a new segment.

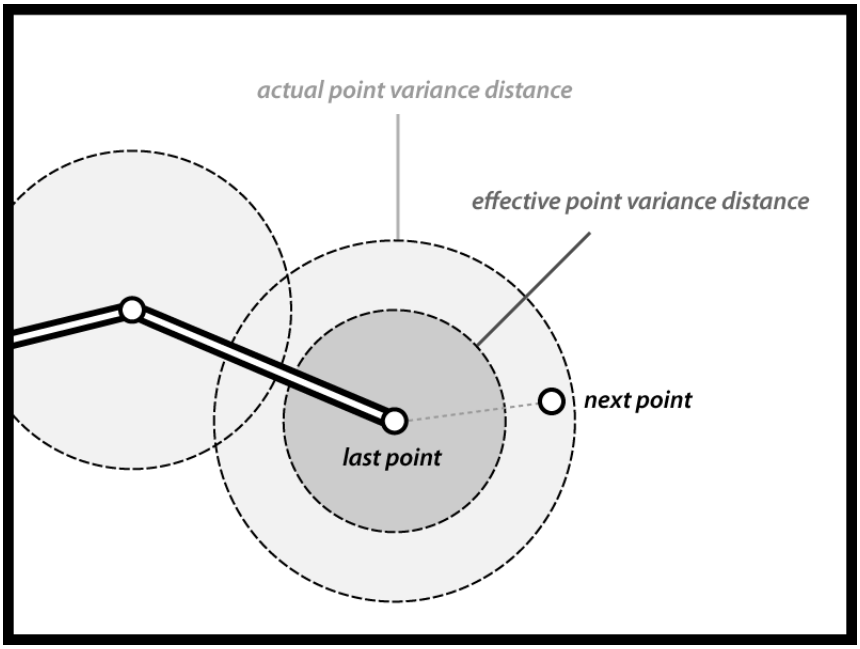


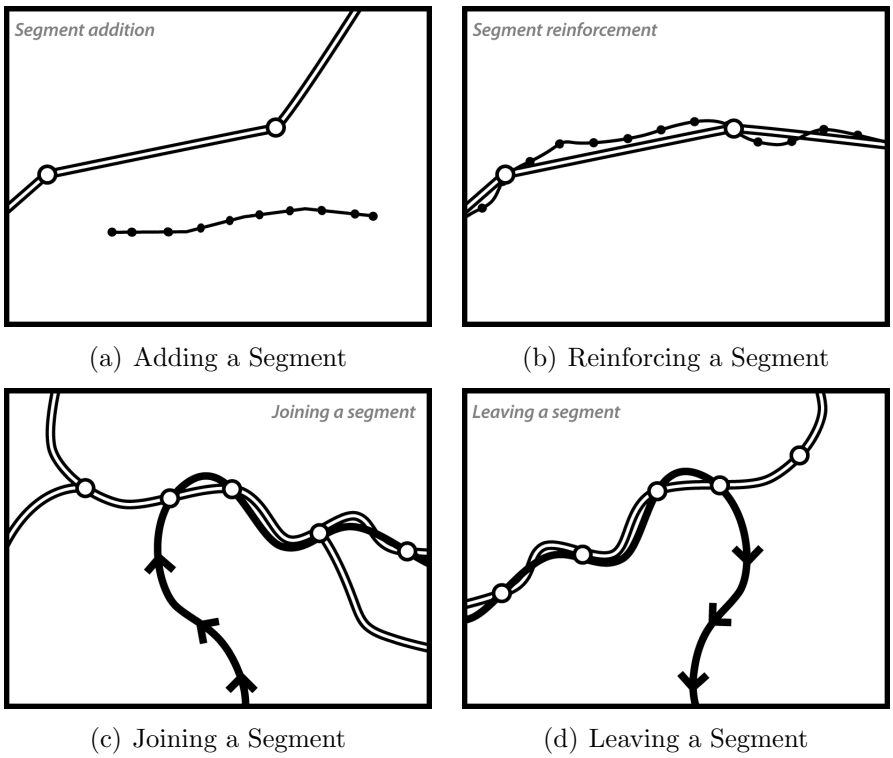

Figure 3.2: New Point Variance Factor

Set at the maximum value of 1, this factor has no effect upon the variance of the most recently added point. Set too small, and the noise present in GPS signals will produce too much detail within each tracklog segment. A value of 0 would prevent points from ever being joined to the most recently added point. The process of combining multiple points from the source tracklog into a single point in the resulting segment has an inherent smoothing effect, which helps reduce the effects of GPS receiver noise.

## 3.2 Building the Network

In the process of adding new trail information into an existing trail network, there are a variety of situations which must be handled properly. I detail them here with a description of the situation and the issues with each. Basic situations are also shown in Figures 3.3 and 3.4.

The base situation in updating a path network is the **addition of a segment**. [Figure 3.3(a)] With an empty path network, this will be the first segment added. It




(a) Adding a Segment (b) Reinforcing a Segment

(c) Joining a Segment (d) Leaving a Segment

Figure 3.3: Basic composition of a network.

will also occur when a segment is added to the network in an area without any existing segments. Areas without existing segments are simply geographic areas where none of the existing points are within their variance distance of any of the points in the new tracklog. When a new segment is added, the points in the source tracklog are added, with a default variance for each point.

When a new tracklog matches an existing segment, then the tracklog data is used to **reinforce the existing segment** [Figure 3.3(b)]. This will occur whenever there is an overlap between the new tracklog and the existing trail network. The new tracklog reinforces the existing segment, updating its position and variance in the process. When the existing segment ends, the tracklog may match another connected segment, in which case the new segment is reinforced with the new data. If there are no connected segments, then new segments are added. Moving from one existing segment to another existing segment always happens near an existing endpoint.

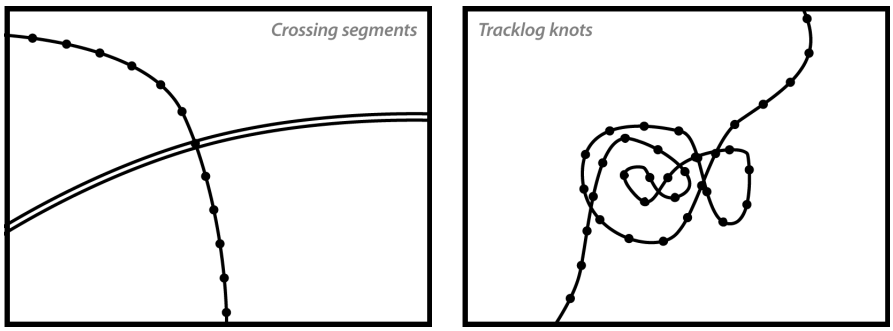
When a newly added segment (with either a new segment or an extension of an existing segment) **joins an existing segment** [Figure 3.3(c)], an intersection between the new segment and the existing segment must be created. When joining the middle of an existing segment, the existing segment is split into two, with a new endpoint at the point where the new segment joins. The three segments (two existing, one new) are then connected together. The future points of the new tracklog are then added to the existing segment, as described previously.

If the tracklog **leaves the existing segment** [Figure 3.3(d)] at any place other then an endpoint, then the existing tracklog is split, and a new segment is created with the first endpoint connected to the two endpoints created by the split. New points will be added to the new segment. This is similar (though in reverse) of the situation that occurs when a new tracklog joins an existing segment in the middle.

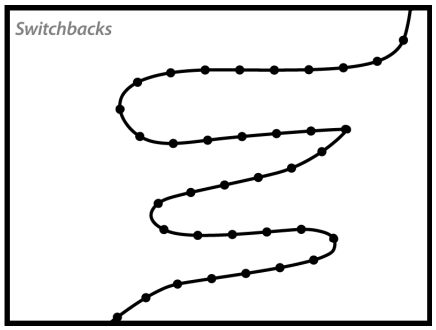
A special case of joining or leaving a segment is where the tracklog joins or leaves the existing segment at an endpoint. In this case, no segment is split, and the segment currently being added is joined to the existing endpoint. This special case will be common in a new network, as segments added by one tracklog are extended or joined by someone who traveled further along the road or trail before heading back.

A second special case of joining a segment and leaving a segment is where the new tracklog **crosses an existing segment** [Figure 3.4(a)]. While this can be handled as first joining and then leaving the existing segment, care must be taken that the intersection is detected even if the points of the tracklog are not close enough to any points of the segment to be combined together.

A consideration of adding new data is that of a **knot** [Figure 3.4(b)] in the tracklog. Knots are common when the GPS receiver remains in one place over time. The noise of the GPS signal will make it appear that the position is moving randomly around a point. It is common to apply smoothing algorithms to GPS tracklogs during analysis, so as to avoid the problem. In my research, I subjectively found that the




(a) Crossing a Segment (b) Tracklog Knots




(c) Switchbacks

Figure 3.4: Special Cases when building a network.

algorithm smoothed the data itself in the course of adding each point, making prior smoothing unnecessary. My analysis of the smoothing results can be found in Section 3.3.4.

A special case to watch out for is the case of the **switchback** [Figure 3.4(c)], where a path makes a sharp turn back on itself. This situation is common where trails climb steep grades, and can cause the tracklog to appear as if it is backtracking on itself. Luckily, this is only a problem when adding the segment for the first time, since subsequent tracklogs will be combined into the existing segment. Detecting switchbacks may be possible through the use of elevation data present in tracklogs, or possibly through the evaluation of the heading of the next several points. If the angle is different enough from the direction of the previous segment, then the close points at the point of the switchback could not be combined in order to record the proper shape of the trail. I had planned to explore these options in the course of my research, but time and data constraints require that it be left as future work.

## 3.3 Update Process

The process of adding new information to the path network follows a Bayesian-like approach. The existing path network is our prior knowledge of path locations, and we update the path network with new tracklogs. As tracklogs are added to the network, the new data is used to add new segments and update existing segments. When existing segments are updated, the position of the segment's points are updated to reflect the new data.

This approach only adds new points to the network when they describe segments not already represented. When a point matches an existing segment, then the position of the nearest segment points are updated to reflect the newly added data point. Only adding data when it improves the spatial network is one of the concepts that keeps this algorithm fast and memory efficient.

### 3.3.1 Intermediate Points

During the process of adding each tracklog's points to the network, we store the series of points in a stack. At the beginning of each evaluative loop, we pop a point from the stack, and consider it the next point to be added.

To reduce the differences caused by the order in which tracklogs are added to the network, we first look for any points in the existing network that lie between the last point evaluated and the next point to be evaluated. To find such an intermediate point, we search the network segments for a point closest to the line segment formed between the last point evaluated and the next. We take the closest existing point and interpolate the point between the last point and next point closest to the existing point. If the existing point is within the variance distance of the interpolated point, then we consider it to be an intermediate point.

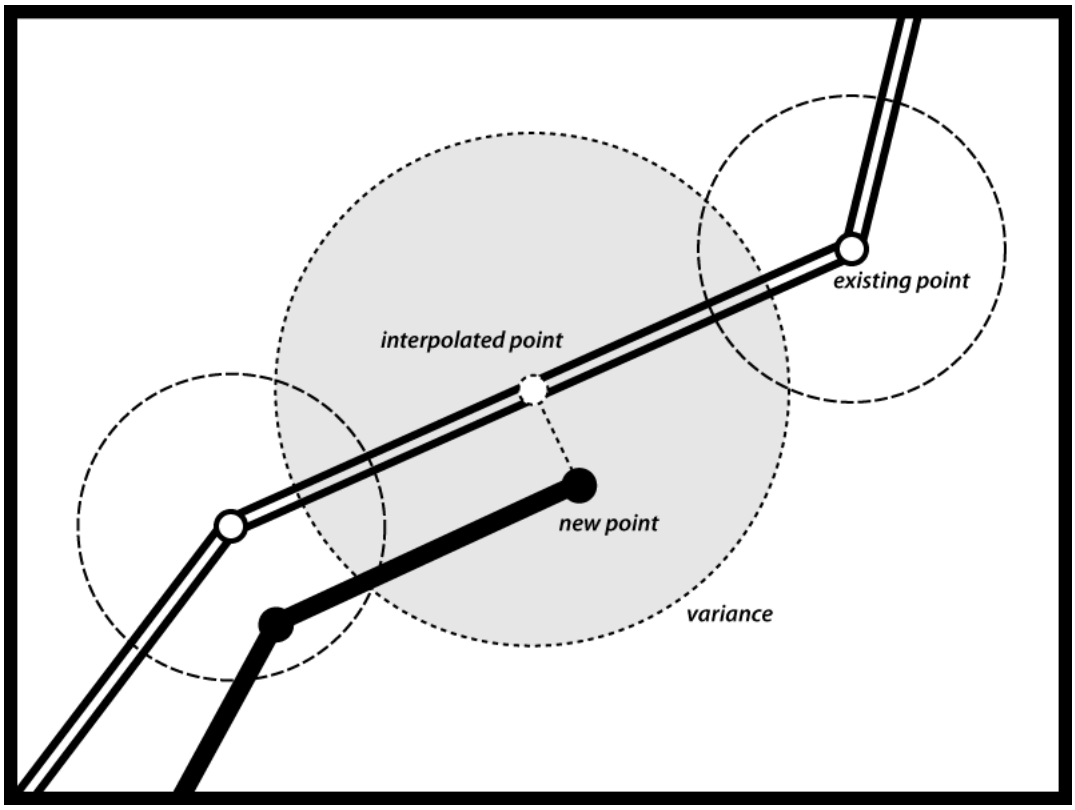



Figure 3.5: Updating a point

When an intermediate point is found, we push the last point removed from the processing stack back onto the stack, and continue evaluation with the interpolated point as the next point to be evaluated.

### 3.3.2 Matching Existing Network Points

During evaluation of the new point, we first search the existing network for any points for which the distance between the new point and the existing point is less then the existing point's variance distance. If there is more then one point, we choose the point with the smallest variance distance. The speed of this search is improved by first testing each network segment with a simple point bounds check. Stored with each segment is a maximum and minimum latitude and longitude that can be used to quickly test if a point falls within its bounds. If the new point falls within these

bounds for any segment, each point in the segment is then evaluated for a potential match.

In addition to matching existing points, we also consider interpolated points when searching the existing network. At times, the point for which we are searching for a match has a shorter distance from the line between two segment points then the distance to either point. In this case, we interpolate a point on the segment between those two closest points. The interpolated point's position is calculated to be the point on the line with the shortest distance to the point, and the population and variance of the interpolated point are calculated using a function based on the variance and population of the two closest points. This interpolated point is then evaluated as with any point according to the process described in the previous paragraph, with the exception that if it is chosen as the nearest point, then it is inserted in the sequence of segment points between the points from which it was interpolated. The process of point interpolation is demonstrated in Figure 3.6 and shown visually in Figure 3.5. Although not applied in our work, the method used to interpolate points can be augmented with a factor that expands the variance of the interpolated point as a function of distance from the source points used in the interpolation. In our algorithm, we produce a distance weighted average of both population size and variance distance. We found the process of point interpolation to be so infrequent that testing different interpolation functions was not possible for a lack of test data.

When searching for a point, we avoid losing source tracklog detail through the use of the New Point Variance Factor. The New Point Variance Factor reduces the effective variance distance for the point in the network most recently updated. This reduction of the variance distance for only the most recently updated point prevents new points from being combined with the previous point, and allows greater detail to be represented. The New Point Variance Factor is described in Section 3.1 and the effects of its application can be seen in Figure 3.2.

```
LineMag = sqrt((B.lat - A.lat)^2 + (B.lon - A.lon)^2)

if LineMag != 0:
  u = (((C.lat - A.lat) * (B.lat - A.lat))
      + ((C.lon - A.lon) * (B.lon - A.lon))) / LineMag^2
else:
  u = 0

if u < 0 or u > 1:
  No Valid Interpolation
else:
  D.lat = A.lat + u * (B.lat - A.lat)
  D.lon = A.lon + u * (B.lon - A.lon)
  D.populationSize = A.populationSize
       + u*(A.populationsize - B.populationSize))
  D.varianceDist = A.varianceDist
       + u*(A.varianceDist - B.varianceDist)
```

Figure 3.6: Pseudo code for the interpolation of point D as a projection of point C onto the line between point A and point B.

### 3.3.3 Condition Checks

After a point is found, we test for conditions that can confuse the process of combining points. These two situations are the detection of switchbacks and the detection of a tracklog which appears to switch to a new segment for only one point before it switches back to the original segment. Both of these situations produce networks which do not accurately represent the true path network.

```
nearestPoint = findNearestNetworkPoint(currentPoint)

if lastPoint at end of segment AND nearestPoint != lastPoint:
  nextPoint = next point in processing list
  nearestPointToNext = findNearestNetworkPoint(nextPoint)
  if nearestPointToNext not found
          OR nearestPoint.segment != nearestPointToNext.segment:
    nearestPoint = Null //forget that we found a nearestPoint
```

Figure 3.7: Pseudo code for detecting switchbacks and preventing undesired point combinations.

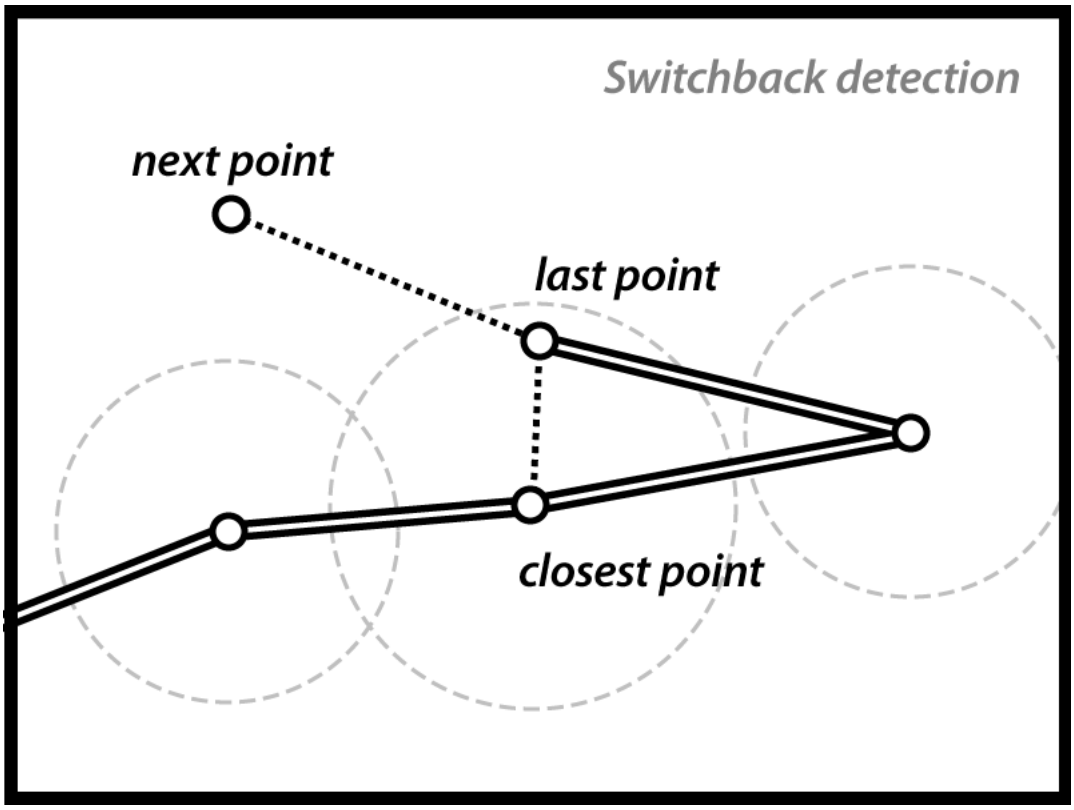
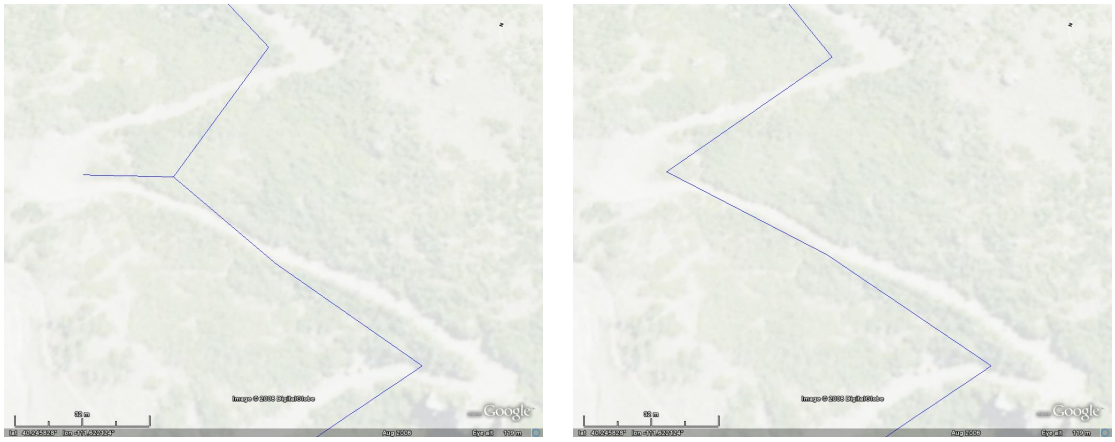



Figure 3.8: Switchback detection.

Switchback detection is performed by looking ahead one point, and finding its closest existing network point. If the last point is at the end of a segment, the nearest existing point to the current point is on the same segment but not at the end, and the look-ahead point does not match any existing segment at all, then the nearest network point is forgotten, and processing continues as if no nearest existing point had been found. Pseudo code for this process can be found in Figure 3.7, accompanied by a visual example of Figure 3.8. An example found within the test data set shows the difference between not using any form of switchback detection (Figure 3.9(a)) and using the switchback detection described in this paper (Figure 3.9(b)).

Detecting a tracklog which appears to hop to a new network segment only to return to the same segment with the next point is detected in a similar way. If the nearest point is on a different segment than the last point, and the look-ahead point finds a nearest point on the same segment as the last point, then the nearest point is



(a) Without Switchback Detection (b) With Switchback Detection

Figure 3.9: Switchback detection in the Progressive algorithm.

```
nearestPoint = findNearestNetworkPoint(currentPoint)

if nearestPoint.segment != lastPoint.segment:
  nextPoint = next point in processing list
  nearestPointToNext = findNearestNetworkPoint(nextPoint)
  if nearestPointToNext is found
          AND lastPoint.segment == nearestPointToNext.segment:
    nearestPoint = Null //forget that we found a nearestPoint
```

Figure 3.10: Pseudo code for detecting tracklogs which hop to a new segment and return the next point.

forgotten, and processing continues as if no point had been found. This situation is found most often when segments parallel each other for a little while before joining, common in switchbacks before they reach the point of the switchback. Pseudo code for this process can be found in Figure 3.10.

If the nearest point was found, and the distance between the next point and the nearest point is less then the variance distance of the nearest point, then the next point is combined with the nearest point, as described in Section 3.4.

### 3.3.4 Intersections

If the segment of the nearest point is on a different segment then the last point, then we may need to leave the previous segment (inserting an intersection if it does not

already exist) and/or join the new segment (also inserting an intersection if it does not already exist).

Inserting an intersection into a segment involves splitting the point sequences of the segment at the desired point of intersection, creating an intersection, and joining the old and the new segment portions together at the intersection. Care must be taken to preserve the existing intersections at both the start and the end of the original segment during this process.

When a segment is split and a new intersection is inserted, a new segment is created which starts at the intersection and will be added to in further processing. This segment is also joined to the intersection.

When adding points to existing segments, new intersections and connecting segments are not created if the last point and the nearest point are found to be connected through the existing network with a connected distance less then the direct distance between the last point and the nearest point, multiplied by the Connected Search Factor described in Section 3.1. A visual explanation of this factor and its calculation is shown in Figure 3.11.

The connected distance for any two points is the 'walking distance' between those two points, and is calculated as explained by the pseudo code in Figure 3.12.

To calculate the connected distance between any two points A and B, we first find the list of segments that connect A's segment and B's segment. A point's segment is the segment that contains the point. This list will be empty if the segments of A and B connect directly. The FindPath() method refered to in the pseudo code is a bounded graph search algorithm.

We start the distance calculation by computing the distance between each pair of points between A and the end of the A's segment. The dist() method in the pseudo code calculates the direct distance between two points, with each point being represented by a longitude and a latitude.

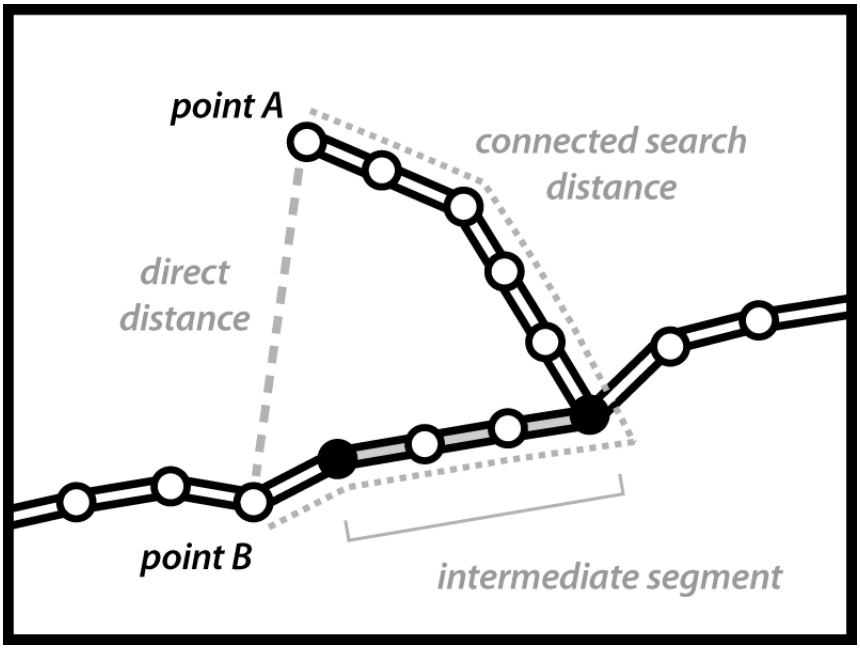



Figure 3.11: Connected Search Distance

```
//A, B = start and end points of connected distance measurement

intermediateSegments = FindPath(A.segment, B.segment)

connectedDistance = 0 //initialize result

for each point p between A and A.segment.end:
        connectedDistance += dist(p, p.next)

for each segment s in intermediateSegments:
        connectedDistance += s.length

for each point p between B.segment.end and B:
        connectedDistance += dist(p, p.next)

// connectedDistance now holds
//    the connected distance between A and B
```

Figure 3.12: Pseudo code for calculating the connected distance between two points.

We then calculate the sum of the lengths of the intermediate segments found through the FindPath() method.

Finally, we calculate the length of the portion of the last segment between B and the end of B's segment as the sum of the distance between each pair of sucessive points, the same way we calculated it for A.

The sum of these distances (portion of first segment, middle segments, portion of last segment) is the connected search distance.

If a path with a connected distance less then the calculated distance can be found, then the points are considered sufficiently connected, and no additional intersections or connecting segments are required. This process prevents jumbles of connecting segments near intersections, if the points do not happen to pass directly through the point of intersection.

If the new point is not combined with the points (existing or interpolated) of the current existing segment, then it will either leave the existing segment or extend the existing segment, depending on the situation of the prior point. This process is described in more detail in Section 3.2.

If the prior point is at the end of its segment and is not at an intersection, then the new point is appended at the appropriate end of the segment's point sequence. If an intersection is present, then a new segment is created, starting at the point of the intersection, and extended to the new point.

When the prior point is not at the end of its segment, then it must leave the segment in the same manner described for an updated point which leaves its previous segment. If an intersection does not exist at the point of leaving, then it is inserted, and a new segment is created starting at the intersection and extending to the position of the new point.

When a new point is added, it is given a Initial Point Variance Distance, and population size of 1.

A common problem of processing GPS tracklogs is properly dealing with tracklog knots. These sections of the tracklog are common when the GPS has been sitting still for a period of time. The noise present in GPS signals causes the calculated position to jump around, creating a 'knot' in the tracklog. An example of a knot can be seen in Figure 3.4(b). The process described above subjectively handles tracklog knots very well, without prior filtering or smoothing.

## 3.4 Combining Points

During the early stages of this research, I had planned to use a method for updating the stored variance for each point as you would a population of points. Due to the complex nature of the data I was working with, I was unable to properly test this method. I subjectively found that a population weighted average produced good results. This method reduced the averaging effects of new points as the number of point combinations grew, allowing the network point to settle with the addition of new data.

I believe that my original method of updating both the mean of the point (stored as longitude and latitude) and the variance may have merit, and leave further study of this method of refining accuracy estimates to future work.

The pseudo code in Figure 3.13 is my adaptation of the algorithm described by Knuth [5], who cites Welford [13]. It has been simplified to become a population weighted average.

I use a population weighted average as the learning rate in these update equations. A population weighted average is only one of the possible learning rates appropriate for use in these equations. In fact, any learning rate between zero and one will eventually converge to the true mean, according to the convergence proof of Watkins and Dayan [12]. In the proof presented by Watkins and Dayan, Q-Learning algorithms converge with any learning rate $\alpha$ such that $0 \leq \alpha < 1$.

```
existing //existing point
         //properties: populationsize, lat, lon, S, variance
new //new point to add with properties: lat, lon

existing.populationsize += 1
deltaLat = new.lat - existing.lat
deltaLon = new.lon - existing.lon
existing.lat += deltaLat/existing.populationsize
existing.lon += deltaLon/existing.populationsize
```

Figure 3.13: Pseudo code for updating a point.

This flexibility to choose any learning rate and achieve convergence allows for not only static learning rates within the allowed range, but learning rates that vary as well. Using a populated weighted average, for example, allows for a learning rate that gradually decreases with each added data point.

The best learning rate is best chosen by the application to data. A spatial network mapping streets might best favor a learning rate which quickly overcomes an initial bias, then drops quickly to avoid a constantly shifting road. A hiking trail network, on the other hand, might best allow for greater influence of new data to allow for trails that shift over time.

# Chapter 4

## Results

I verified the results of this algorithm by evaluating the networks compiled from two separate sets of tracklogs.

To evaluate proper handling of intersections, trail combinations, and switchbacks of the network I have chosen a set of trails that exist within the area just east of Provo, Utah between Provo Canyon and Hobble Creek Canyon. This set of trails represents a variety of situations, including areas with many duplicate trails, areas with just a single trail, many trail junctions and switchbacks. The resulting network was inspected to verify proper handling of segment intersections. Since the trails used in this portion of the evaluation are familiar to the author, previous experience with the trails and reference satellite photography will serve as the reference for these inspections.

The objective evaluation will be performed with a set of data gathered in the metro area of Provo. The path network created from the metro area will be compared against accurate street centerline records [4] as a baseline.

In both the subjective and objective tests, I compare the results of my algorithm with the results of using the algorithm proposed by Morris et al. [7].

## 4.1 Situational Handling

The dataset used for situational handling analysis was collected from user submitted tracklogs and a selection of national forest trails downloaded from ActiveTrails.com

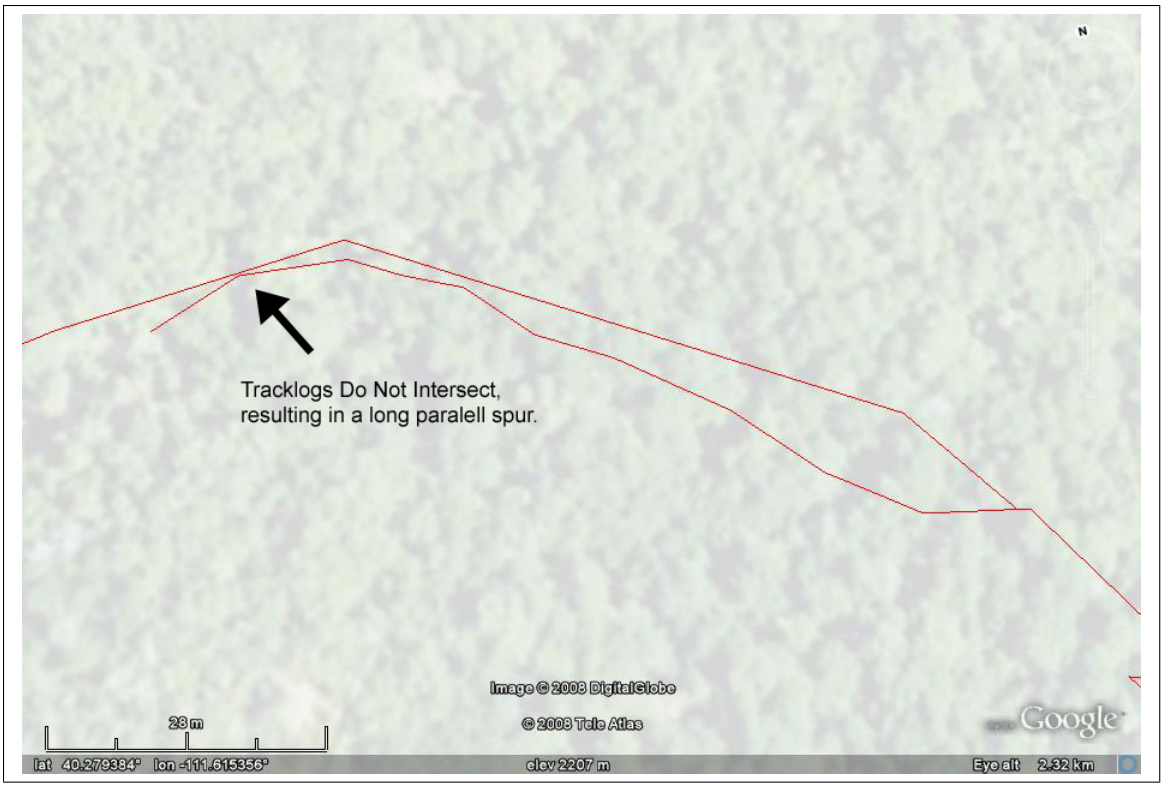



Figure 4.1: Parallel spur in the Trail Libraries Algorithm.

[1]. The area for this dataset was chosen for the availability of duplicate trails and the author's familiarity with the area. This dataset contains 41 tracklogs, for a total of 17,232 data points.

I used a variance distance of 30 meters when running the Progressive Algorithm. Values higher then 30 meters tended to reduce important trail features, while values less then this allowed small splinter branches not representative of the actual trails.

For the Trail Libraries algorithm, I used a value of 80 meters as the reduction threshold. For most of the network, a value of 50 meters worked just fine. Using a value of 80 meters did combine some of the more inaccurate tracklogs into the same segment.

On nearly every situation, both algorithms handled the data properly. While the results of each were slightly different, the output of each was a reasonable representation of the trail.

Each algorithm had artifacts unique to the style of processing. The Trail Libraries algorithm is particularly vulnerable to tracklogs that travel close to each other for a time without actually intersecting each other. This condition leaves parallel segments in places where only one trail exists. This artifact can be seen in Figure 4.1.

A similar artifact exists in the output of the progressive algorithm, although it is caused by different conditions. To prevent 'loops' in the tracklog, a connected distance test is performed to determine if a new tracklog segment is added. If the Connected Distance Factor is too large, then undesired loops appear. If it is too small, then we observe small stubs that appear in the form of partially completed loops, as seen in Figure 4.2. The Connected Distance Factor must be adjusted to the dataset to minimize these problems.

While the parallel segment artifact of the Trail Libraries algorithm and the stub artifact of the Progressive Algorithm are similar, the stub artifact is less troublesome for two reasons. First, the size of the stub artifact is limited by the Connected Search Factor, while the size of the parallel segment is unbounded. Second, the stub artifact can be reduced and possibly eliminated through proper tuning of the Connected Distance Factor, while no algorithmic adjustments can eliminate a parallel segment.

Switchbacks were handled decently well by both algorithms, but the winner is clearly the Trail Libraries algorithm. In the absence of any parallel tracklog problems, the output was a much better match for the actual path. In the Progressive Algorithm, smaller variance distance values produced a better trail representation on switchbacks, but caused problems elsewhere in the dataset. Because of the high initial variance

value, some of the smaller switchbacks were compressed into a single line, as seen in Figure 4.3.

The situation most difficult to test was the case of one tracklog crossing another. The simple case in this situation finds a point on one of the tracklogs that matches a point on the other, and this happened every time with our test data. The more difficult situation is where the tracklogs do cross, but none of the points of the first are near the points of the second, so no intersection is computed. This did not occur in our tests, and is also unlikely to be found in data collected from consumer GPS devices, unless the signal is lost during the collection of both tracklogs for an extended period of time.

The main observable difference in quality of handling straight line segments is the results when an inaccurate tracklog is combined with accurate data. If the tracklogs intersect, and the width is less then the threshold of the Trail Libraries algorithm, then the resulting segment is skewed closer to the inaccurate tracklog quite a bit. If the width is greater then the threshold, then it is represented as two parallel lines. In the Progressive algorithm, the segments are joined when they are within the variance distance of each other, and split when they are not. This can result in segments that split, rejoin, split and rejoin several times along their length. An example of this condition is shown in Figure 4.4. While inaccurate data is not desirable in either case, the progressive algorithm prevents bad data from producing a major offset in segment position.

## 4.2 Algorithmic Analysis

Geospatial algorithms are natively $O(n^2)$, or $O(m * n)$, where m = number of existing data points and n = number of new data points. Through optimization of search algorithms, this can be reduced to $O(n \log n)$. As neither algorithm used these optimizations, they have been excluded from this analysis.

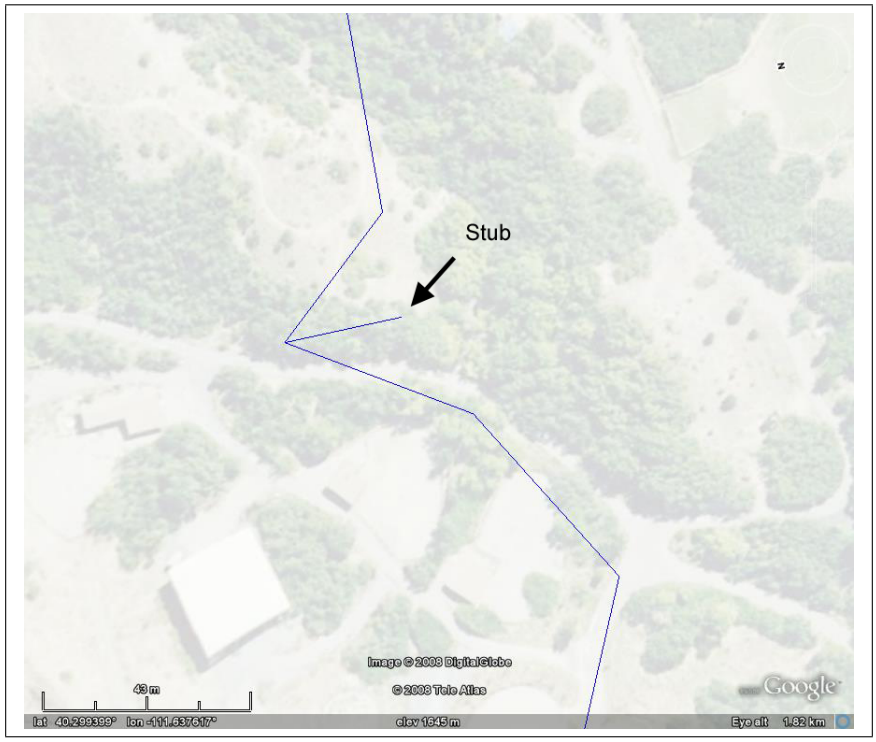



Figure 4.2: Stub resulting from too large of a Connected Distance Factor in the Progressive Algorithm.

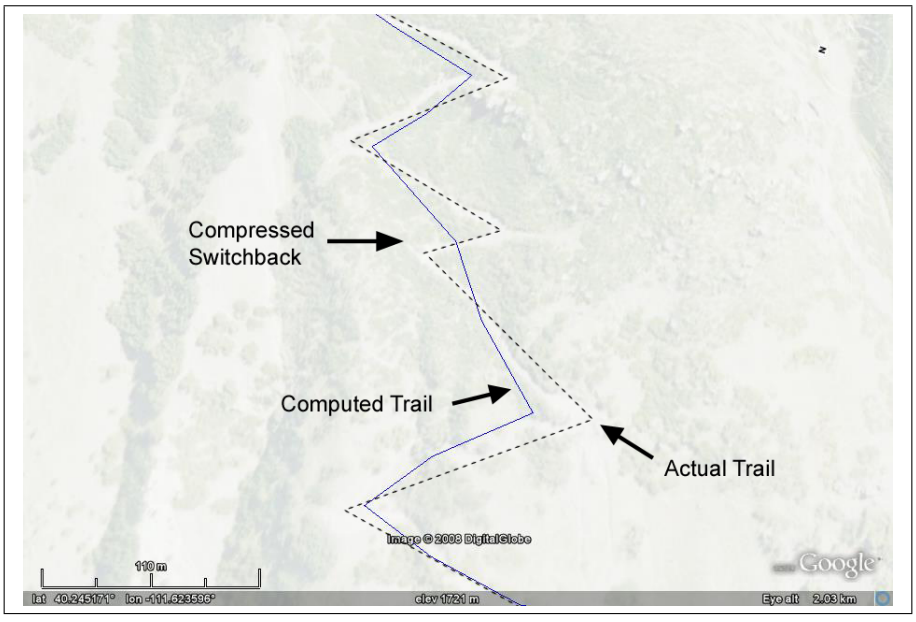



Figure 4.3: Compressed switchbacks as a result of a high Initial Point Variance value in the Progressive Algorithm.

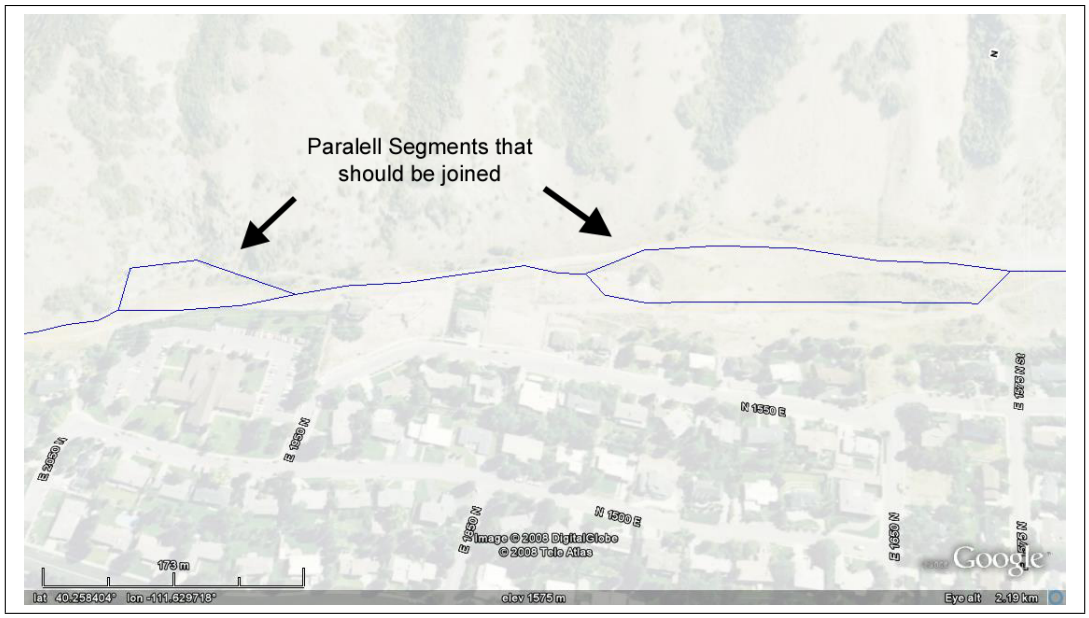



Figure 4.4: Segments partially joined in the Progressive Algorithm. Portions of the segment were less then the variance distance and were combined.

In the Trail Libraries algorithm, $m \geq n$, as new data points are added when computing intersections between segments. When the source data is dense and many intersections occur, $m \gg n$. Because all source points are added prior to any reductions, the algorithm performs at $O(n^2)$.

The Progressive Algorithm performs updates as points are added. When a new point augments an existing point, no new points are added to the existing set, allowing m to stay constant for that update. As points that fall within the variance distance of an existing point are combined instead of added, there is a upper bound to the number of points within a specific geographic area.

This upper bound causes the algorithm to behave as $O(n^2)$ only during the discovery of new paths. After discovery is complete, m does not grow, and when m is constant, the algorithm performs at $O(m * n)$, which is to say linear in the number of new data points.

The effects of this algorithmic difference can be seen in the computational speed of processing data presented in Table 4.1. With only six tracklogs, the process-

ing cost of the Trail Libraries algorithm demonstrates fast growth, and the Progressive algorithm shows steady growth.

The speed difference is directly affected by the data set size produced by each algorithm, shown in the dataset size table in Table 4.2. The data set size produced by the Progressive algorithm (shown in the P column) shows that after the first tracklog (which covers the entire area of testing) only a few additional points are added. The Tu column of the same table shows the very rapid growth of the dataset under the Trail Libraries algorithm.

There are only a few exceptions within the Progressive Algorithm which allow a point in the dataset that is within the variance distance of an existing point. The best example of this includes switchback detection, which allows points within variance distance to preserve the actual shape of the trail when two trails exist in close proximity. These exceptions are only added during the process of adding a new segment to the network. Additional points added after the first segment are combined into existing points as normal. Because these exceptions are only added during the discovery phase of network creation, they do not prevent m from becoming constant.

## 4.3 Numerical Comparison

The dataset I used for numerical evaluation comprises data from a metro area of Provo. The dataset contains six tracklogs, for a total of 1,216 points. The tracklogs were gathered during three separate trips in a passenger vehicle using two consumer GPS units. The first four tracklogs (first two trips) contained complete coverage. The fifth and sixth tracklogs (third trip) covered only a portion of the area, but contained driving paths not previously collected. During the collection process, all traffic laws were followed, and travel was in normal lanes of traffic. After collection, each tracklog was edited slightly to remove the start/end data, and correct a few errors.

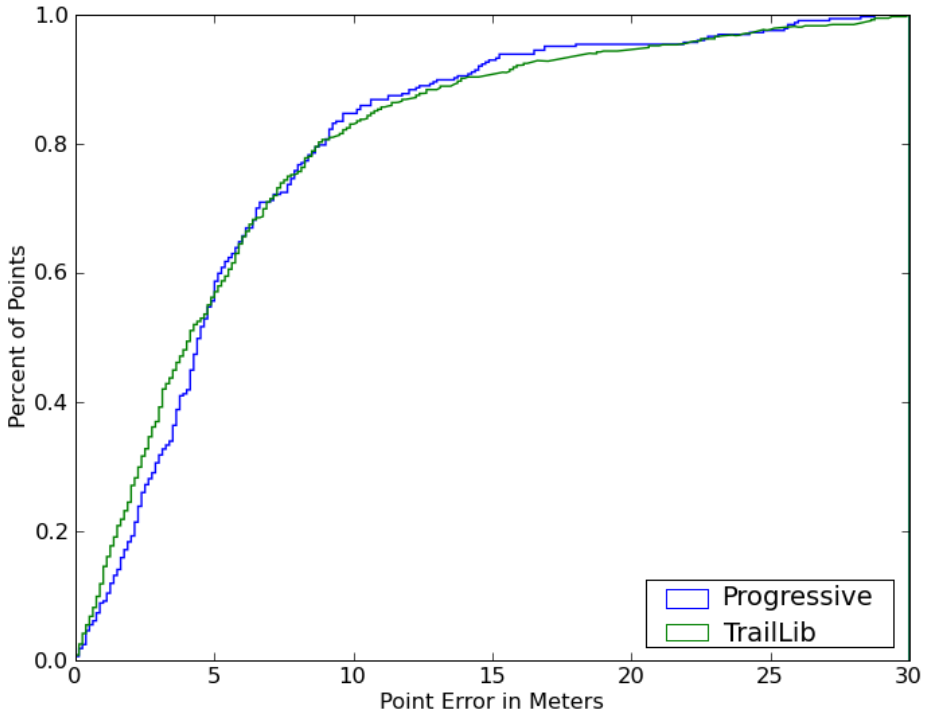



Figure 4.5: Cumulative histogram accuracy comparison

The errors edited out of the source tracklogs in every case were the result of traveling outside the test area, usually as a result of a wrong turn, and once as a result of traveling through the drive-through window of a fast food restaurant for an ice cream cone. These errors were corrected by removing the data points from the location the tracklog left the testing area until it returned to the testing area. As no data was available in these areas for comparison, including them would have introduced additional error into the numerical tests.

Metro street centerline data for the designated area was extracted from a publicly available county dataset [4]. This centerline data serves as a baseline during comparison of the progressive algorithm output, and the output of the Trail Libraries [7] algorithm.

When running the progressive algorithm, I used a variance distance of 50 meters. A variance distance of less then 50 produced several tracks on most streets, and a variance distance of more then 50 combined several of the smaller blocks.

For the Trail Libraries algorithm, I found a value of 80 meters as a reduction threshold to produce the best values for the street data collected. This value stops just short of combining small blocks into one street.

As a comparison metric, I used the Point Distance to Baseline Network, calculated as the distance between each point in the computed network and the nearest points and lines in the baseline network. This distance is either the point distance to a point within the baseline network or the distance to a segment within the baseline network, selecting the closest distance.

While this metric is not perfect, it does give a simple measure of the differences between the output of each algorithm. During my tests, I found the difference between the algorithms to be insignificant. Figure 4.5 contains the cumulative histogram plot of both algorithms. The histogram is computed as a normalized count of Point Distance to Network of both algorithms, and can be used to determine which percentage of the points were within each distance from baseline network.

## 4.4 Speed

As both algorithms are non-deterministic, I ran the comparison between algorithms 10 times, and averaged the results. The tests were run on a 2.2 Ghz Intel Core 2 Duo with 2 GB of RAM. While actual run times will vary between machines, these tests demonstrate the difference in processing speed between the algorithms. The Factor (F) column of the table in Table 4.1 shows that the difference between the algorithms is increasing as well. Such a growth rate will soon render the Trail Libraries algorithm infeasible of networks of any significant size.

The results shown were gathered on the numerical comparison dataset, which demonstrates that the accuracy is similar despite the massive increase in the speed of processing.

The difference in speed and the increasing factor between the two algorithms can be explained with a look at the processing steps taken by each algorithm. Table 4.2 lists the size of the source dataset, the intermediate and final datasets of the Trail Libraries algorithm, and the final size of the Progressive algorithm. The numbers listed are the number of segment points present in the dataset. The size of the source dataset is cumulative, representing the number of points processed when that number of source files is used in the analysis.

The process used by the Trail Libraries algorithm involves combining all source tracklogs and points into a single graph. During this process, all overlapping segment intersections are identified, and an intersection is created there joining all of the appropriate segments. Creating an intersection adds more points to the dataset, as represented by the increasing numbers shown in the Tu column of Table 4.2. This process of creating a single graph can explain the growth of the computation times shown in Table 4.1. The extra points created during the first phase of processing are never reduced or eliminated from the final dataset, resulting in a dataset that is larger then the original every time. These extra points also lengthen the processing times.

The Progressive algorithm uses a process that only adds new points when they add additional information to the final spatial network. The results of this process can be seen in the substantially lower numbers of the resulting dataset size. Even as new data is added, the data set only grows slightly as the data is refined and updated. The advantage of a small dataset is one of the advantages of a stream processing approach, as it simplifies both the processing of new data and reduces the size of the final dataset.

Although no numerical accuracy comparison was performed on the situational handling dataset, the difference in processing time is even more remarkable. The

Progressive algorithm completed in 15.36 seconds, while the Trail Libraries Algorithm required 784.40 seconds to complete.

| Num | P(s) | T(s) | F |
|---|---|---|---|
| 1 | 0.22 | 2.06 | 9.24 |
| 2 | 0.68 | 14.10 | 20.80 |
| 3 | 1.29 | 33.01 | 25.67 |
| 4 | 2.02 | 79.00 | 39.05 |
| 5 | 2.81 | 113.37 | 40.37 |
| 6 | 3.18 | 174.53 | 54.82 |

Table 4.1: Speed comparison of the Progressive Algorithm (P) and the Trail Libraries Algorithm (T) measured in seconds, of an increasing number of tracklogs. The Factor (F) between the algorithm's times is also listed.

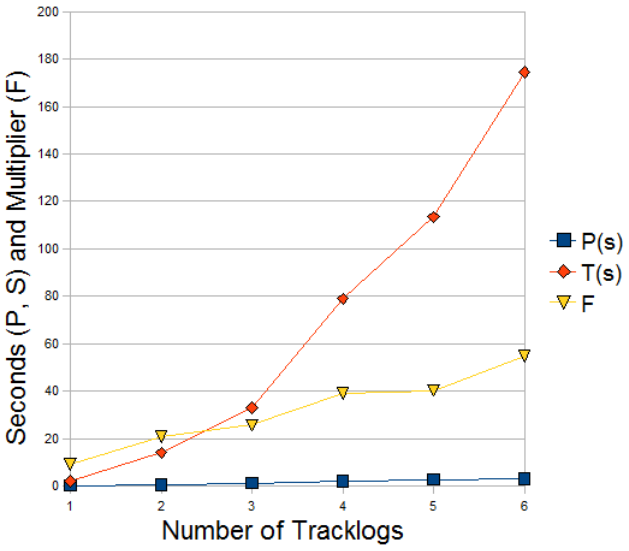



Figure 4.6: Graph of Table 4.1

| Num | S | Tu | Tr | P |
|---|---|---|---|---|
| 1 | 337 | 525 | 407 | 183 |
| 2 | 532 | 1918 | 725 | 220 |
| 3 | 794 | 3844 | 1100 | 240 |
| 4 | 931 | 6417 | 1792 | 250 |
| 5 | 1144 | 7970 | 1982 | 261 |
| 6 | 1216 | 9458 | 2220 | 271 |

Table 4.2: Comparison of dataset size between the Source Data (S), the Trail Libraries Algorithm before reduction (Tu) and after reduction (Tr), and the Progressive Algorithm (P)

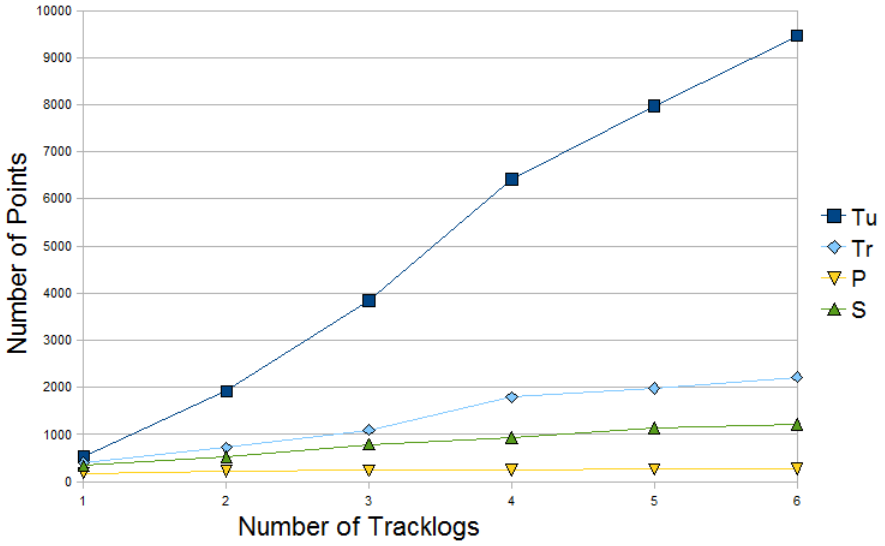



Figure 4.7: Graph of Table 4.2

# Chapter 5

## Analysis

While I failed to prove some of the properties of the Progressive Algorithm that I had originally planned to, my work demonstrates three powerful principles. These are the value of intermediate data, the algorithmic benefits of spatial data limits, and the viability of a stream processing approach for geographic data. I also gained insight into the use of Bayesian updates and the importance of source data accuracy.

### 5.1 Intermediate Data

The algorithm I've constructed relies on the storage of intermediate data that is used in future calculations. Each time I update an existing point, I increase the population counter stored with the point. This small piece of additional data allows me to perform a population weighted average when updating the location of the point. This concept could be applied to the Trail Libraries [7] algorithm to overcome its bias toward newly added data. If each point in their network contained a measure of how many original points had been averaged together previously, then a weighted average could be produced without any significant increase in algorithm complexity.

Population size is not the only intermediate variable that might be useful in calculating geographic networks. With future research into Bayesian [6][10] updates of each point's location, the variance distance can also be stored as an intermediate value. Elevation data is another example of intermediate information that may not

be needed output of the algorithm, but can be used in processing. Elevation data might be useful, for example, in keeping airplane flight paths at different elevations separate.

## 5.2 Spatial Limits

My application of spatial limits on dataset size is perhaps the most valuable insight of this research. While I applied spatial limits as part of my stream processing approach, they could also be applied in a filter process prior to the main computation. This concept is very powerful in the area of geographic data, as there is a natural limit to the number of data points of a particular resolution within a geographic area.

Reducing the number of data points can reduce the effectiveness of the algorithm, but the use of intermediate values provides the ability to overcome this effect.

The benefit demonstrated with the very small data set used in my research shows great promise. As the growth rate of the data set is the factor, the benefits of this approach increase dramatically with system and data set size.

## 5.3 Stream Processing

My work also demonstrates the viability of data stream processing for computation of geographic data. The speed of processing demonstrated in section 4.4 indicates that such processing would be possible in real time on a resource constrained device such as a smart phone or personal GPS device, even with complex datasets. It also demonstrates the possibility of using centralized servers to compute spatial networks from data streamed from remote mobile devices, and computed centrally into a spatial network. An example application of such central processing would be mapping the walking travel paths taken by fans leaving a stadium. Data provided by smart phones in possession of some of the fans can be computed into a spatial network, and then

fed back to the phone to provide efficient walking directions to the user's car based on current flow of fans and the traffic arrangements made for the game.

## 5.4 Bayesian Updates

During the planning phases of this research, I had planned to use an application of Bayes rule (described in Section 2.5) to refine the data as additional data was added. Using the Bayesian update process did not return the results expected. Testing and tuning the point update process was difficult due to the complexity of the situational data of the numerical analysis dataset as described in Section 3.4. The high complexity to segment length ratio of the metro data set made it difficult to test and observe the effects without causing unintended consequences. Selecting a dataset with lower situational complexity would aid further research into this process.

I believe that the concept of using Bayesian processes to update has merit, but must be studied further to determine its applicability.

## 5.5 Accuracy of Source Data

The variable accuracy of the source data did surprise me. While most of the data was fairly accurate, a few of the tracklog sections were significantly inaccurate. Dealing with some inaccurate data proved to be more difficult then I had previously believed.

The inaccurate data made it necessary to expand the variance distance, which caused problems elsewhere in the dataset. Using a single variance distance for the entire operation is what led to most of this struggle. Using a form of user feedback to locally adjust the variance distance could avoid this problem and aid in the production of more accurate results.

# Chapter 6

## Conclusion and Future Work

### 6.1 Conclusion

Spatial Networks are valuable information tools and can be computed from tracklogs collected by consumer GPS devices. Updating spatial networks with new data in an efficient manner is an important requirement pushed by the proliferation of GPS devices capable of transmitting data in real time.

I demonstrated a progressive, stream processing approach to updating spatial networks. This method shows significant speed improvements over existing methods. Adding data to the network is significantly faster, and by a factor which increases as the network grows. The increase in speed is partially due to the reduced complexity of the algorithm results.

I demonstrated the utility of spatial limits on data set size. By imposing limits on data set growth, spatial update algorithms can run in linear time.

I failed to prove that accuracy will improve as additional data is added, but did demonstrate that the accuracy matches that of existing methods.

This thesis demonstrates the viability of a stream processing approach to geographic information processing, explores the utility of storing intermediate data to facilitate fast processing, and provides a platform for further study of geographic processing of spatial networks.

## 6.2 Future Work

Further study of the process of updating the variance distance of points in the spatial network could yield improvements to the accuracy of updating spatial networks, as well as enable reporting of accuracy estimates.

Observing the difficulty of choosing the proper variables for an entire network hints at the possibility of using human input for local adjustments to variance distance and other algorithm variables.

The algorithm as demonstrated is greedy as it changes network topology. After a segment is created, it is never combined into another segment even if it is within the variance distance of another segment. Adding some evaluation to the network to enable topological changes to the network would help refine some of the problems caused by this situation.

Study of the learning rate [12] during the point combination process could yield faster converting, future preferring, and other types of convergence.

Storing additional meta data, such as the direction of travel, could yield additional information about the computed network. This additional information can further describe network segments, including direction of travel or average speed.

# Appendix A

## Debugging Tricks

Visualizing geographic data can be difficult, and visualizing the intermediate steps of a geographic stream processing can be even more so. Rather then build my own visualization tools, I output data in the form of KML files, which can be viewed and explored in Google Earth [2]. This dynamic environment allows a much easier way to evaluate geographic data then any tool likely to be built for academic evaluation. The product is available free of charge, and carries no restriction which would prevent its use in academic research.

### A.1 Geographic Data Visualization

KML files easily represent points and paths typical in geographic data. Points have a latitude, longitude, and an elevation, and can be augmented with an icon to represent point type. Other information about a point can be embedded in the name and description fields available for each point. In my work, I displayed the population size of each point in the description field.

I used points to represent both segment points and segment intersections. In the case of intersections, I used the description field to list the identifiers of connected segments. This helped me to visually verify the correctness of the graph I was building.

Segments can be given a name, as well as a line style, which includes color and width. I used the color of segments to separate the output of different algorithms, allowing visual comparison of similar data sets.

Google Earth contains two methods for viewing KML files. The first is to load the file into Google Earth directly. A copy of the data is stored within its database, making this a great method for taking snapshots of output data. After import, the data can be renamed to assist in identification. The second method is called a Network Link. While network links typically point to an online service, you can also point them at a file on the local system. By pointing the network link at the current output file of the system under development, the user can easily reload the current data in the file. Network links can be refreshed manually, automatically when the current view moves, or with a timer. I used the view change method, which allowed me to adjust the view slightly when I wanted the data reloaded from disk. Using the timer method sometimes caused file conflicts if Google Earth tried to reload the data while it was being written.

Distance between points is often an important aspect of geographic data, and Google Earth contains two ways to approximate distances on datasets. Visible on the bottom left corner of the screen is a scale bar that adjusts according to the current zoom level. Zooming in on the distance to be measured allows use of the scale tool as a rudimentary ruler for quick distance checks. A more accurate measurement may be taken by using the Ruler tool, available in the tool bar or the Tools menu.

Another useful feature of KML files is the ability to embed a timestamp into the points in the file. When timestamp elements are present, a time slider control is displayed, allowing the user to filter the data to a specific time period, and even scrub the control back and forth to see prior or subsequent data. While the timestamps of my collected data were not useful for debugging, I output the last updated timestamp for each data point. This allowed me to follow the progress of the stream processing

algorithm by scrubbing the control back and forth. It also allowed me to pinpoint the most recently updated points, which was invaluable for debugging errors in the process. The time granularity of the scrubbing control is limited to seconds. My algorithm processed points fast enough that the timestamp of each point was nearly identical. To make it easier to scrub through the points, I kept an artificial clock, which I advanced one second each time a point was updated. The timestamps were only used to show progression in time, and so the false time values did not cause any problems for evaluation.

## A.2 Stream Processing

Viewing the output is very useful for development and debugging, but it is sometimes helpful or necessary to follow the processing of each data point as it flows through the processing code. To allow such analysis, I used the step debugger available in combination with the KML output features described above. Because I was interested in inspecting the processing of data in a small geographical area, I created code to support a 'debug window' defined by a latitude and longitude bounding box. When the algorithm processed a point within the debug window, it output a debug message and also dumped the current data to the KML file linked to Google Earth. By setting a breakpoint on the debug message that only ran when within the window, I was able to fast forward the processing to the point I desired to inspect. I also set a flag upon entering the window, so that the pauses for debugging continued past the window. Using my debugger, I could clear the flag to continue fast forwarding. I also set a limit on the number of tracklogs that must be processed before entering the window, making it easy to fast forward to the 5th or 6th time the window was entered.

Each time the breakpoint was hit, I could refresh the data in Google Earth, inspect the current data, and then step through the algorithm one line at a time. This allowed me to verify the correct processing of the data points under inspection. It

also allowed me to gather the values of intermediate variables. By copying the values of intermediate variables and pasting them into the search box of Google Earth, I was able to easily visualize the data used during the processing of each point as well as the final result.

Without these debugging visualization tools, building and testing the algorithm described in this paper would have been nearly impossible.

# Bibliography


[1] ActiveTrails - http://www.activetrails.com/.

[2] Google Earth - http://earth.google.com/.

[3] OpenStreetMap - http://www.openstreetmap.com.

[4] Utah County GIS Data - http://ims2.co.utah.ut.us/website/download1/data1.cfm.

[5] D. Knuth. The art of computer programming. Vol. 2: Seminumerical algorithms. Third Edition. *Reading*, page 232, 1981.

[6] R. Michalski, J. Carbonell, and T. Mitchell. *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann, 1986.

[7] S. Morris, A. Morris, and K. Barnard. Digital trail libraries. *Digital Libraries, 2004. Proceedings of the 2004 Joint ACM/IEEE Conference on*, pages 63–71, 2004.

[8] T. Peucker. A theory of the cartographic line. *International Yearbook of Cartography*, 16:134–143, 1976.

[9] S. Rogers, P. Langley, and C. Wilson. Mining GPS data to augment road models. *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 104–113, 1999.

[10] S. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1995.

[11] S. Stigler. Who discovered Bayess theorem. *American Statistician*, 37:290–296, 1983.

[12] C. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8(3):279–292, 1992.

[13] B. Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3):419–420, 1962.