CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

Estimation of Travel Time Distribution and Detection of Incidents Based on Automatic Vehicle Classification

V. Anantharam
University of California, Berkeley

California PATH Working Paper
UCB-ITS-PWP-98-12

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department Transportation, Federal Highway Administration.

Report for MOU 223

June 1998

ISSN 1055-1417

ABSTRACT

We study the problem of travel time estimation along a section of a freeway based on data derived from vehicle detectors at multiple locations. We pose the problem as one of pattern recognition. We derive algorithms that aim to recognize patterns which persist between the error-prone upstream detector samples and the error-prone downstream detector samples. We describe how these can allow us to estimate the distribution of the travel time between these detector locations. The most promising algorithm derived in this research is a dynamic programming based algorithm based on sequence matching techniques that would process the data as it arrives and could therefore provide continuously updated travel time estimates in real time.

Keywords: Travel time estimation, pattern matching, sequence matching, dynamic programming.

EXECUTIVE SUMMARY

This report describes the conclusion of project MOU 223 *Estimation of Travel Time Distribution and Detection of Incident Based on Automatic Vehicle Classification* funded by the EECS Department of the University of California, Berkeley by PATH/CALTRANS. The aim of the project was to investigate the relevance of pattern matching techniques to the problem of travel time estimation.

In the course of this project, several potentially useful algorithms from the area of pattern matching were identified as being worthy of further investigation. Most promising among these appears to be a dynamic programming based algorithm that would process the data as it arrives and could therefore provided continuously updated travel time estimates in real time.

Some preliminary investigations of crude versions of this algorithm were done on the I-880 database. However, this database provides limed feature vectors (only absence/presence of a vehicle) and a proper evaluation of these ideas will require the availability of more detailed data sets.

ESTIMATION OF TRAVEL TIME DISTRIBUTION AND DETECTION OF INCIDENTS BASED ON AUTOMATIC VEHICLE CLASSIFICATION

FINAL REPORT

MOU 223

V. ANANTHARAM
Professor
EECS Department
University of California
Berkeley

FINAL REPORT

Introduction

Congestion on the freeways is an increasing problem in California and throughout the nation. It is not an exaggeration to say that it is one of the chief bottlenecks in the efficient functioning of the economy. Traffic management techniques to prevent and to alleviate congestion are therefore important to develop. In particular, rapid detection of incidents on the freeway will help in reducing the time to return traffic to smooth flow after such incidents. Travel time estimation can be used to provide feedback to drivers, enabling them to make informed choices between alternate routes to their destination, thereby increasing the efficiency of the freeway system as a whole.

Existing approaches to traffic travel time estimation and incident detection use detectors of various kinds (typically single loop inductive loops) deployed along the freeway and are generally based on exploiting the correlation between the information provided by successive detectors in some way. There are several such techniques that use macroscopic information such as density, volume or occupancy, as well as microscopic information such as velocity and intervehicular headway, if available (for instance the velocity can be derived from double loop inductive detectors), etc. Good surveys of currently available travel time estimation and incident detection techniques are given in [13], [11]. The most successful among the existing algorithms use pattern recognition algorithms - they look for features derived from the occupancy, volume, and density measurements that are indicative of an incident. The popular *California* algorithms are algorithms of this type, see [13].

Macroscopic traffic quantities are relatively crude characterizations of the traffic. The traffic stream between successive detectors will typically undergo many complex changes. For instance, overtaking and relative distance changes between vehicles at successive detectors is important. Further, on sections of the freeway having entry and exit ramps, the entry and exit of vehicles introduces an additional source of variation in the measured traffic characteristics.

Imagine the traffic stream on a lane, as recorded by human observers who might be prone to error, at two locations along the freeway, see Figure 1 :

The top half of the picture shows what is seen by the upstream observer. the bottom half shows that seen by the downstream observer; the time axis of the bottom half is shifted relative to that of the top half by shift roughly equal to the travel time. Several features that make the travel time estimation problem relatively difficult are indicated in this figure. For instance, one of the 2-axle trucks in the bottom half has not been seen well enough by the observer to enable him to classify it. A new 2-axle truck has entered the traffic stream. perhaps from an adjacent lane. Two new cars appear to have entered, perhaps from entry ramps, and one of the cars appears to have overtaken the pair of 2-axle trucks in

the top figure, by moving to an adjacent lane and returning to its own lane. There are also marked differences in the inter-vehicle spacing between the two traces. However, despite these problems, this is a considerably more detailed view of the traffic stream than would be provided by only macroscopic quantities such as flow, density, and occupancy.

The technology of vehicle detectors has improved to a point where it is now possible to carry out automatic vehicle classification from the measured vehicle-specific patterns at the detector. For instance, one could use laser range-finders. Here the idea would be to place the range finder above each freeway lane. The height versus length profile of the vehicles passing underneath could then be detected by the range-finder, if its response were sufficiently fast. For a vehicle travelling at 30 meters/sec a laser range-finder able to take a sample every 10 ms would be able to take a sample at every 30 cm along the length of the vehicle. This seems quite adequate for a good classification of the vehicle. One could also use optical techniques for automatic feature extraction, as studied in the PATH projects of Malik et al. [10] and Varaiya et al. [14]. Here video cameras are deployed along the freeway, and image processing algorithms are run on the video traces in order to locate the vehicles and to classify them according to some scheme.

In principle, this is just like having human observers prone to error deployed along the freeway, i.e. it is now realistic to pose the problem of travel time estimation and incident detection as being one of designing algorithms that will use information derived from the detectors very similar in nature to that encapsulated in the traces of Figure 1.

We may pose the problem as one of pattern recognition : we wish to recognize patterns which persist between the error-prone upstream and downstream samples, which will allow us to estimate the distribution of the travel time between these detector locations. The comments made earlier about the complexity of the transformations involved in going from the top trace to the bottom trace in Figure 1 show that this is a rather novel problem of pattern recognition, relative to those encountered in the published academic literature. Further, there are additional constraints that make the problem unique - the information produced by the detectors arrives in a continuous stream, and the algorithms will have to work *in real time*, i.e. they should be able to process the data as it arrives and should be able to provide a running estimate of the travel time that adapts and tracks the continuously changing information.

In this project we have explored how one might adapt the pattern matching techniques that are currently used in text searching applications and in molecular biology applications to address these novel problems of estimation of travel time and of incident detection.

Guiding theme of the research

The main observations guiding our research were :

(1) The California algorithms are found to be among the most satisfactory of the currently available algorithms for travel time estimation and incident detection [13], [11]. They are based on pattern recognition algorithms that use macroscopic data such as occupancy, volume, and density measurements and look for features based on these measurements, which are indicative of an incident.

(2) The technology of vehicle detectors has improved to a point where it is now possible to carry out automatic vehicle classification from the measured vehicle-specific patterns at the detector. Thus, much better information can be expected from the individual detectors, more akin to that provided by a possibly error prone human observer than that provided just by macroscopic variables. Whatever technology is used for detection and classification of vehicles, we expect to see a stream of feature vectors at each detector. The number of features is likely to be somewhere between 10 to 100.

(3) There has a been a deep development of approximate pattern matching algorithms motivated by applications such as molecular biology, text searching, and web browsing. In the molecular biology applications, researchers mostly study DNA or RNA strings and protein strings. The former can be thought of as strings of features drawn from a set of size 4 (the four nucleotides) and the latter as strings of features drawn from a set of 20 (the 20 amino acids). The questions of interest in this area are very similar to the questions of interest to us : molecular biologist compare different DNA strings (or different protein strings) to one another and look for similarities, which indicate a common origin and common role for the corresponding part of the string in the overall functioning of the organism. In the text searching and web browsing applications, the information available in the text or on the web is treated as a huge database. The model is that one poses a query, in the form of a pattern, and the algorithms work to locate the places in the text (or on the web) where this pattern appears. Such pattern matching seems also to be a natural ingredient in the problem of interest to us : subpatterns of the traffic stream at an upstream detector are likely to persist for a while and show up in somewhat distorted form as subpatterns of the traffic stream at the downstream detector; examples of such patterns might include truck convoys, vehicles driven by relatively cautious drivers who maintain their relative position without attempting to overtake, etc.

Our overall aim has been to leverage the developments in pattern matching algorithms for molecular biology and text searching and apply them to the traffic area.

The big picture

Imagine a stretch of freeway, as shown in Figure 2. We consider two detectors on the same lane of the freeway, separated by a distance L, as indicated in the figure : the downstream detector (U) and the upstream detector (D). No assumptions are made about the section of the freeway between the detectors - in particular, there may be several lanes on the freeway, the number of which may change over the section between the detectors, and there may be entrance and exit ramps between the detectors.

For our discussion we assume that each of the detectors provides features from the

observed traffic stream in the form of a string of symbols drawn from a finite set. Suppose, for instance, that each detector is a laser range finder, sampling the distance to the top of the vehicle (or the roadway, if there is no vehicle currently present) at some sampling rate, say 100 Hz. The measured heights would be discretized. For instance, if they are discretized at a resolution of 1 cm, and the heights of vehicles can be as much as 6 meters, a generous upper bound, then each sample is from a set of size 600. We may think of this data as being locally processed, resulting in a string of symbols of the form :

$$\ldots\ g\ g\ g\ A\ g\ g\ g\ g\ g\ g\ g\ g\ D\ g\ g\ g\ g\ g\ X\ g\ g\ A\ g\ g\ g\ g\ E\ g\ \ldots$$

Here "g" is a symbol denoting a gap (no vehicle), the contiguous intervals of gaps have varying lengths, and A, D, X, E, etc. denote different vehicle types. The local processing of the laser detector samples basically consists of identifying a vehicle type from a local collection of samples. We would use a single symbol to represent a vehicle, because the samples are assumed to come at a fixed rate in time, and the number of samples of a vehicle seen by a stationary detector would depend on the velocity of the vehicle. To clarify, what we mean is that even if, say, 6 samples result at the detector while the vehicle passes underneath, we first classify the vehicle, as A, say and write $A\ g\ g\ g\ g\ g$ for the corresponding portion of the output of the stream from the detector. We could, write $A\ A\ g\ g\ g\ g$ or $A\ A\ A\ g\ g\ g$ etc., of course, but the number of A's we write should be fixed a priori and not depend on the number of samples taken by the detector of the vehicle A, the reason being that the number of such samples will depend on the speed of the vehicle as it passes under the detector. The length of the vehicle is, of course, incorporated in the notion of the vehicle type. so we are not really losing this information. If we wanted to, we could even incorporate the measured velocity of the vehicle in the notion of vehicle type, but then the scoring matrices we use (see below) will have to be carefully chosen to not overly penalize the matching of a vehicle with a given velocity with the same vehicle with a different velocity.

The ouput of the upstream detector is just another such symbol string.

If we were using video data, we might proceed as follows : after running an image processing algorithm on the image seen by a video detectors, the location of vehicles is determined, and a vehicle is classified as belonging one of a set of types. Of course, there is the possibility of misidentification. Examples of vehicle classification include the Highway Performance Monitoring System(HPMS) of the Federal Highway Administration, which classifies vehicles into 13 categories, [9], or e.g., the automatic vehicle classification scheme used by the Oklahoma Turnpike Authority in a 1990 test of automatic vehicle classification [4], which used a classification into 8 classes :

1. Automobile, motorcycle or 2-axle four-tired trucks.
2. Class I vehicle towing 1-axle trailer
3. Class I vehicle towing 2-axle trailer
4. 2-axle bus or 2-axle 6-tired truck

5. 3-axle bus or single/combination 3-axle truck
6. 4-axle combination truck
7. 5-axle combination truck
8. 6 or more-axle combination truck.

This scheme used inductance detector loops and photoelectric sensors emitting infrared light beams for automatic vehicle classification, which was compared to visual classifications by a human observer, correctly classifying 95% of 1736 observed vehicles. This is only for illustrative purposes, and any other classification scheme that is considered appropriate could be used. Again, suppose the symbol "g" is used to represent a gap, i.e. the lack of a vehicle. Then, after such image processing and vehicle classification, a symbol string would result from the downstream detector of the form :

$$\ldots\ g\ g\ g\ A\ g\ g\ A\ g\ g\ g\ D\ g\ g\ D\ g\ g\ g\ A\ g\ X\ g\ g\ A\ g\ g\ g\ g\ E\ g\ \ldots$$

where the contiguous intervals of gaps have varying lengths, and A, D, X, E, etc. denote different vehicle types, as before. The ouput of the upstream detector is ' just another such symbol string.

Data from inductive loop detectors could be locally processed using a Karhunen-Loeve approach, such as that being followed by the UC Irvine project of Ritchie et al. [12], and used earlier by Kühne, [8]. This works as follows : Different vehicles have characteristic metallic mass distribution along their length. This results in different inductive loop detuning curves when the vehicle passes over an inductive loop detector. This difference can be used for vehicle classification. The inductive loop detuning curves of vehicles are approximately decomposed as linear combinations of eigenfunctions. These eigenfunctions are precomputed based on the detuning curve signatures of samples of vehicles of the different kinds. This provides a finite number of feature vectors associated to each detected vehicle. The idea suggested and used by [8] and being used by Ritchie et al. [12] is to simultaneously consider the string of feature vectors observed by different detectors along the freeway and to compare them, so as to arrive at travel time estimates. In [8], measurements were made on the A4 Cologne-Aachen freeway in Germany. The approach taken is to compute correlation functions for different hypothetical time shifts for the string of feature vectors at neighbouring detectors. However, one may proceed as follows : one may bin the coefficients of the Karhunen-Loeve projection. This results in a feature vector that is one of a finite set of possibilities (perhaps of the order of a 1000 possibilities, if 3 eigenvectors are used for the K-L expansion, and each of the coefficients is binned using one of 10 possibilities). From the downstream detector we might get the symbol string

$$\ldots\ g\ g\ g\ A\ g\ g\ g\ g\ g\ g\ g\ g\ B\ g\ g\ g\ X\ g\ g\ E\ g\ g\ g\ g\ g\ g\ \ldots$$

where again "g" denotes a gap and symbols like A, B, X, E, etc. represent certain (binned) combinations of feature vectors. The same comments we made earlier to explain why we write the symbol corresponding to a vehicle only once apply here also. Once again, the upstream detector also gives rise to a similar symbol string.

Now consider time t. At this time we choose to think of the string of length n of symbols seen at the upstream detector

$$P = [x_{t-n+1}\ x_{t-n+2}\ \ldots x_t\ ]$$

as being a *pattern*. Fix $0 < t_a < t_b$, chosen on the basis of the reasonable assumption that vehicles will not be able to travel between the detectors with a time less than t_a, and that, except under very rare highly congested conditions, it is unlikely that they will take longer than t_b to travel between the detectors. For instance, if L = 1 mile, then, with the reasonable assumption that vehicles do not travel faster than 120 mph and most likely are travelling faster than 5 mph, we could take $t_a = 30$ seconds and $t_b = 720$ seconds. We think of the string of symbols at the downstream detector

$$T = [y_{t-n+t_a+1}, \ldots, y_{t+t_b}]\ ,$$

which is of length $m = t_b - t_a + n$, as being a *text* of length m. We now pose the problem of *determining approximate occurrences of the pattern in the text.*

Note that one could also adopt another approach : at time t one could take the string of length n at the *downstream* detector

$$P = [y_{t-n+1}\ y_{t-n+2}\ \ldots y_t\ ]$$

as being the pattern, and the string of length m at the *upstream* detector

$$T = [x_{t-n-t_b+1}, \ldots, x_{t-t_a}]\ ,$$

as being the text. We will occasionally make comments about both schemes in the course of this discussion, but will focus on the latter scheme. This is because, in the latter scheme (pattern at the downstream detector), pattern matching can, in principle, begin at time $t-n$ and proceed in real time, so that at time t one has already determined which locations in the text give approximate matches to the pattern. In the former scheme (pattern at the upstream detector), we would have to wait till time $t + t_b$ for the entire text to be seen at the downstream detector before being completely sure about the locations of all the approximate pattern matches.

In this project we explore algorithms to solve this approximate pattern matching problem. We believe that this is a central ingredient in the overall problem of travel time estimation and incident detection. This is because, with a good approximate pattern matching algorithm, we expect that the locations in the text which will approximately match the pattern will tend to congregate around the true travel time, with perhaps a few outliers. The overall travel time and incident detection algorithm would use the output of the pattern matching algorithms to determine when to react - the better these algorithms are, the lower the false alarm rate of the overall scheme will be.

Exact pattern matching

We do not, of course, expect to see *exact* matches of the pattern in the text, except by chance. Nevertheless, the conceptual framework behind approximate pattern matching

algorithms is akin to that used in exact pattern matching algorithms, and, further, some good approximate matching algorithms use exact matching algorithms as an ingredient. Therefore, we discuss exact pattern matching algorithms in this section.

Let P be a *pattern* of length n and T a *text* of length m. We wish to find all *exact* occurrences of the pattern in the text. We say that P occurs in T at location i if

$$[x_1 \ldots x_n] = [y_i \ldots y_{i+n-1}] \ .$$

The *naive algorithm* is to slide P along T one step at a time. Each step takes n comparisons. There is a total of $m-n+1$ steps. This results in an algorithm of complexity $\Theta(mn)$.

Much better algorithms are known for this problem. For each location i in P, $1 \leq i \leq n$, let $P[1 \ldots i]$ denote the prefix of P of length i. The *Knuth Morris Pratt algorithm* (KMP algorithm) [7] works by first preprocessing the pattern P in the following way : For each location i in P, $1 \leq i \leq n$, and each symbol $x \in \Sigma$, define $s(i,x)$ to be the length of the longest proper suffix of $P[1 \ldots i]$ which exactly matches a prefix of P and such that $P[s(i,x)+1] = x$. If there is no such proper prefix of $p[1 \ldots i]$, set $s(i,x) = 0$ for $x = P[1]$ and $s(i,x) = -1$ for $x \neq P[1]$. These numbers $s(i,x)$, $1 \leq i \leq n$, $x \in \Sigma$ can be determined by preprocessing P in time $\Theta(n)$.

We slide the pattern P along T searching for matches, as in the naive algorithm, except for the following differences : Suppose that we are at location j of the text, and have determined that $P[1 \ldots i] = T[j \ldots j+i-1]$. If $i = n$ we have found a match. We examine $T[j+n]$, call it x, and shift the pattern to the right by $n - s(n,x)$. The point is that we now know for sure, unless $s(n,x) = -1$, that $P[1 \ldots s(n,x)+1] = T[j+n-s(n,x) \ldots j+n]$, so we only need to make additional comparisons from position $j+n+1$ onwards of T. If $s(n,x) = -1$, then again we only need to make additional comparisons from position $j+n+1$ onwards of T. If $i < n$ we know that P does not match T at location j. We examine $T[j+i]$, call it x, and shift the pattern to the right by $i - s(i,x)$. As in the previous case, we can argue that we only need to make additional comparisons from position $j+i+1$ onwards of T. The upshot is that each symbol of the text T needs to be involved in exactly one comparison. This results in an algorithm of complexity $\Theta(m)$. If one includes the time required to preprocess P, the overall complexity of the KMP algorithm is $\Theta(m+n)$.

The *Boyer Moore algorithm* [2] is potentially even better, in that often it works without even needing to involve every location of the text in a comparison, although, of course, any deterministic algorithm in the worst case (over all choices of P and T) will have to examine every location of T. This algorithm also works by sliding P over T from left to right, but the comparisons are now done from right to left. The amount by which to shift the pattern to the right in each step is the maximum of that given by two heuristics : the *bad character rule* and the *good suffix rule*.

To apply the rules we first preprocess P to determine certain quantities. For a symbol $x \in \Sigma$ and a location in the pattern $1 \leq i \leq n$, let $r(i,x)$ denote the location of the rightmost occurrence of x in $P[1 \ldots i]$, namely $P[r(x)] = x$ and $P[j] \neq x$ for all $r(x) < j \leq i$. If x does not appear in $P[1 \ldots i]$, set $r(i,x) = 0$. This quantity is required for

the bad character rule. Next, for any $1 \leq i < n$, consider the suffix $P[i+1 \ldots n]$ of P. Define $t(x)$ so that $P[t(x) \ldots t(x) + n - i - 1]$ is the rightmost copy in P of this string, for which either $t(x) = 1$ or $P[t(x) - 1] \neq P[i]$. If there is no such copy, set $t(x) = 0$. This quantity is required for the good suffix rule. Also required for this rule is the quantity $l(i)$ defined as the smallest value $1 \leq l(i) \leq n - i - 1$ such that $P[i + 1 + l(i) \ldots n] = P[1 \ldots n - i - l(i)]$, assuming such a value exists. If such a value does not exist, set $l = n - i$.

Consider first the bad character rule. Suppose P is currently aligned with T at location j of T, i.e. $P[1 \ldots n]$ is currently being compared to $T[j \ldots j + n - 1]$. Suppose first that there is an exact match. Then the bad character rule does not require a shift of the pattern to the right. If there is a mismatch, suppose that $P[i+1 \ldots n] = T[j+i \ldots j+n-1]$ and that $P[i] \neq T[j + i - 1]$ (recall that we are making comparisons from the right of the pattern to its left). Let x denote $T[j + i - 1]$. The bad character rule requires us to shift the pattern to the right by $i - r(i, x)$ before making the next comparison. This is natural. To see this, if $r(i, x) > 0$, by shifting in this way, we are making the least possible shift that could possibly result in a match, because we already know that $T[j + i - 1] = x$, so there better be an x matching it, if there is to be a match. If $r(i, x) = 0$, again we are making the least possible shift that could possibly result in a match, because we have to slide the pattern all the way over the x sitting at location $j + i - 1$ of T in order to potentially get a match.

Consider next the good suffix rule. Again suppose P is currently aligned with T at location j of T, i.e. $P[1 \ldots n]$ is currently being compared to $T[j \ldots j + n - 1]$. Suppose first that there is an exact match. Then $t(x)$ is necessarily 1 for any x. The good character rule requires us to shift the pattern to the right by l (note that l is always ≥ 1, so we make some progress). This is natural, because we would have to shift at least as much as to get a proper prefix of P matching a proper suffix of P in this case, and if there is no such prefix and suffix, we would have to shift n steps to the right and start over. If there is a mismatch, suppose that $P[i+1 \ldots n] = T[j+i \ldots j+n-1]$ and that $P[i] \neq T[j+i-1]$. Let x denote $T[j + i - 1]$. Then the good suffix rule says that if $t(x) > 1$, we should shift the pattern to the right by $i + 1 - t(x)$ steps, while if $t(x) = 0$, we should shift the pattern to the right by $i + l$ steps. Once again, it is not hard to convince oneself that such a shift is necessary before there could possibly be a match.

The Boyer Moore algorithm [2] shifts the pattern at each stage by the maximum of the shifts prescribed by the bad character rule and the good suffix rule. Typically, many characters of the text do not even need to be examined, leading to a sublinear time algorithm. The worst case complexity of the algorithm is $\Theta(m)$.

The *Karp Rabin algorithm* [6] is a randomized algorithm for exact pattern matching, based on the concept of *fingerprinting*. Let d denote $\mid \Sigma \mid$. We interpret the pattern P and the portions $T[j \ldots j + n - 1]$, $1 \leq j \leq m - n + 1$ of the text as integers by first enumerating the symbols in some order as $\{0, 1, \ldots, d - 1\}$ and then interpreting a string $[x_1 \ldots x_n]$ as $\sum_{i=1}^{n} d^{n-i} x_i$. We pick a random prime p from the primes of size at most nm^2 and take the residue modulo p of the numbers resulting from the pattern and from the portions of the text of length n starting from each j, $1 \leq j \leq m - n + 1$. Call these numbers π_p and $\tau_p(j)$, $1 \leq j \leq m - n + 1$, respectively. The subscript is to remind us of the choice of the prime.

Note that these numbers are in the range $\{0, 1, \ldots, p-1\}$, because they are residues modulo p. Further the numbers $\tau_p(j)$, $1 \leq j \leq m-n+1$, can be easily computed iteratively : to get $\tau_p(j+1)$ from $\tau_p(j)$ one takes the residue modulo p of $p\tau_p(j) + T[j+n]$, where, with some abuse of notation, $T[j+n]$ denotes the numerical value in the range $\{0, 1, \ldots, p-1\}$ assigned to the symbol $T[j+n]$. Now, if the pattern matches the text starting from location j, we must have $\pi_p = \tau_p(j)$. So we can simply check whether this condition holds, for each $1 \leq j \leq m-n+1$. There is a chance of an erroneous declaration of a match when no match exists, while if there is a match we will definitely detect it. The probability of a false match anywhere along the text can be shown to be $O(1/m)$ when the prime is chosen randomly from primes less than nm^2. The overall complexity of the algorithm is $\Theta(m)$.

One can also devise matching algorithms based on a data structure called a *suffix tree*. Given a string, say the text $T = [y_1 \ldots y_m]$, for concreteness, a suffix tree for it is a rooted directed tree $\mathcal{T}$ with exactly m leaves, labeled $1, \ldots, m$ and with the properties : each internal node other than the root has at least 2 children; each edge is labeled with a nonempty substring; no two edges out of a node can have labels beginning with the same symbol; the concatenation of the labels on the edges along the unique path from the root to the leaf labeled j spells out the string $T[j \ldots m]\$$. Here \$ is a special symbol not belonging to the symbol set Σ from which the symbols of T are drawn. A suffix tree exists for any string, for example, the suffix tree of the string $[a\ b\ a\ b\ b\ ]$ over the alphabet $\{a, b\}$ is shown in Figure 3.

Algorithms are known that can construct the suffix tree of a string of length m in time $O(m)$. The pattern matching problem of finding all the locations in a text T of length m which match a pattern P of length n can be solved by first constructing a suffix tree $\mathcal{T}$ for T. Then one starts at the root and works one's way down the tree using the edge labels, to match P to the extent possible (for any symbol string there is a unique way to go down the suffix tree, starting from the root, till at some point it may happen that one cannot proceed). If it is impossible to match the entire pattern P by going down the suffix tree $\mathcal{T}$ in this way, this means that the pattern does not appear anywhere in the test. If it is possible to match the pattern entirely to starting from the root of the suffix tree and ending possibly in the middle of an edge, then consider all the labels of the leaves of $\mathcal{T}$ in the subtree that lies below the node that ends the path along which the pattern P was matched. These are precisely the indices of the locations in T which match P !

The overall complexity oo this algorithm is $\Theta(n+m)$, including the $\Theta(m)$ time to construct the suffix tree of T, and the $\Theta(n)$ time to attempt to match P along this suffix tree.

Approximate pattern matching

The exact pattern matching problem was discussed in the preceding section only because it forms an ingredient of one of the suggested algorithms for approximate pattern matching which will be described below. In the traffic travel time estimation problem under consideration, it is clearly too much to expect the existence of exact matches between the

pattern at the downstream detector and the text at the upstream detector (or vice versa).

Rather, intuition suggests that patterns are likely to match provided that one allows for variation in the sizes of gaps between subpatterns, and for the introduction and/or deletion of other subpatterns between subpatterns of the overall pattern. The preceding sentence is deliberately somewhat vague and suggestive. One can formalize such notions in several ways. One appealing approach is via the concept or *edit distance* between two strings.

Consider given two strings $A = [a_1 \ldots a_k]$ and $B = [b_1 \ldots b_l]$ with symbols drawn from the finite setΣ. We introduce a symbol e for a "gap" in a string. From either one of the strings, say A for concreteness, we can generate strings with symbol set $\Sigma \cup \{e\}$ by introducing gaps. What this means is the the symbols from Σ appear in the same relative order as in the original string A, but may no longer be adjacent to each other. Note that we also allow for gaps at the beginning and/or end of the strings that we generate in this way.

Consider augmenting each of A and B by the introduction of gaps resulting in strings A' and B' respectively of equal length n, place these two sequences one on top of the other, and count the number of mismatches. Consider the smallest such number one could possibly get, over all choice of augmentations A' and B' (the common length of the augmentations is also a free parameter). This is called the edit distance between the strings A and B. This terminology results from an alternative description of this number - we could start with A and consider how to transform it into B step by step, using at each step any one of three operations ; *Insert*, which involves inserting a symbol from Σ at some location in the current string; *Delete*, which involves deleting a symbol from some location in the current string; and *Substitute*, which involves replacing a symbol from some location in the current string by some symbol from Σ. One can verify that the minimum number of such steps that will transform A into B (or, obviously, vice versa), is exactly what we called the edit distance between A and B. For example, the edit distance between the strings [P A T H] and [B A R T], thought of as coming from the symbol set which is the 26 letter alphabet, is 3, as shown by the alignment

$$\begin{array}{c} P\ A\ e\ T\ H \\ B\ A\ R\ T\ e \end{array}$$

which has 3 mismatches, and the observation that it is impossible to create an alignment between these two strings that has fewer than 3 mismatches. A transformation of [P A T H] to [B A R T] by insertions, deletions, and substitutions, in 3 steps is :

$$[P\ A\ T\ H\ ] \mapsto [B\ A\ T\ H\ ] \mapsto [B\ A\ T\ ] \mapsto [B\ A\ R\ T\ ]\ .$$

The edit distance between two strings is a special case of the *weighted edit distance* between the strings. To define this, we need a *scoring function* defined for pairs of symbols drawn from $\Sigma \cup \{e\}$. This is a collection of real numbers $s(x, x'), x, x' \in \Sigma \cup \{e\}$. The choice of these numbers is an important issue, and this choice is likely to considerably influence the performance of the algorithms based on scoring functions that we will propose below.

When these algorithms are implemented for field testing, or simulated for testing against data derived from the field in a laboratory environment, it will be important to allow for experimentally tuning these paramenters to get the best performance for a given pair of detectors.

To define the weighted edit distance, consider augmentations A' and B' of A and B of common length n, exactly as before, and place them one on top of the other, and compute the number $\sum_{i=1}^{n} s(a'_i, b'_i)$. The minimum of this number over all possible choices of augmentations of common length, with the length of the augmentations also a free parameter, is the weighted edit distance between the strings A and B for the particular scoring matrix under consideration.

In the following we prefer to discuss the problem as a maximization problem rather than a minimization problem. These problems are clearly equivalent, involving only a change of sign in the scoring matrix. Thus, we talk about the *similarity* between the two strings A and B as the *maximum* score between augmentations A' and B' of the respective strings, and attempt to determine this quantity, as well as the alignment of the two strings that results in the maximum similarity.

The similarity between two strings A and B can be computed by *dynamic programming*. This is a powerful technique, and, as we will see, this observation leads to algorithms for travel time estimation that can be implemented *in real time*. Given two strings $A = [a_1 \ldots a_k]$ and $B = [b_1 \ldots b_l]$ and indices $1 \leq i \leq k$ and $1 \leq j \leq l$ let $V(i,j)$ denote the similarity between the prefixes $A[1 \ldots i]$ and $B[1 \ldots j]$ of A and B respectively. Then one has

$$V(i,j) = \max[V(i-1,j-1) + s(A[i], B[j]), V(i-1,j) + s(A[i], e), V(i,j-1) + s(e, B[j])]$$

with the initial conditions

$$V(0,j) = \sum_{a=1}^{j} s(e, B[a]) \ , 1 \leq j \leq l \ .$$

and

$$V(i,0) = \sum_{b=1}^{i} s(A[b], e) \ , 1 \leq i \leq k \ .$$

Here, we implicitly assue that $s(e,e) < 0$, as is natural. When computing $V(i,j)$, we retain pointers to the location ($(i-1, j-1)$ or $(i, j-1)$ or $(i-1, j)$) through which the maximum was achieved (if there is a tie, we retain pointers to each of the tying possibilites). By following any path of pointers back from (k,l) to $(0,0)$ we can determine an alignment of the strings A and B that gives rise to the maximum similarity.

In the travel time estimation problem we have two string : the pattern P and the text T, derived from the downstream detector and the upstream detector respectively (or vice versa). If we were to consider the similarity between these two strings, it is like comparing the *entire* string P to the *entire* string T. However, the pattern would typically be of relatively small length compared to the text, and should therefore focus on approximately

matching the pattern to *substrings* of the text. We say that an *approximate occurrence* of a pattern P occurs in the text T if there is a substring $T[i \ldots j]$ of T such that the similarity between P and $T[i \ldots j]$ is at least σ, where σ is some fixed threshold, and where the similarity is defined in such a way that spaces (e's) to the left of the pattern are free. Note that, in practice, σ could also be fine tuned on the field for the pair of detectors under consideration.

It turns out that one can determine the approximate occurrences of P in T using the same dynamic programming recursion considered earlier. The only difference is that we change the initial conditions to

$$V(0, j) = 0 \ , 1 \leq j \leq l \ ,$$

with $V(i, 0), 1 \leq i \leq k$ defined as before. Here we are implicitly assuming that $s(e, x) < 0$ and $s(x, e) < 0$ for all $x \in \Sigma \cup \{e\}$, in addition to $s(e, e) < 0$, which is a natural assumption to make when the problem is formulated as one of maximizing the score. Now, one can show that an approximate occurrence of P occurs at $T[i \ldots j]$ if and only if $V(n, j) \geq \sigma$ and there is a path of pointers from (n, j) to cell $(0, i - 1)$. Note that, in practice we may not be so much interested in determining the exact portion of the text over which the approximate match occurs, as in just knowing that an approximate match has occurred that ends at position j of the text (i.e. we may not be interested in also knowing i). This is an important point for the discussion that follows.

We now illustrate the value of the dynamic programming approach to the problem of weighted edit distance computation by means of an assesement of the number of computations that would be required in what appears to be a typical example of the problem of traffic travel time estimation. Consider the pattern P of length n resulting from a stretch of symbols at the downstream detector. Here n should also be thought of as a design parameter of the overall scheme, which can be fine tuned to best suit the particular stretch of freeway under consideration, but for now we think of it as being fixed. For instance, if one is able to generate a symbol every 1/60 th of a second from the the detector, and one takes observations for a time of 2 minutes, the resulting pattern has length $n = 7200$. Suppose the upstream detector is 1 mile away from the downstream detector. A reasonable assumption is that the speed of the traffic stream is no more than 120 mph, and is at least 5 mph. This translates into a travel time of between 30 seconds and 720 seconds, i.e. of between 1800 and 43200 samples. Consider fixed time t. We compare the pattern

$$P = [x_{t-7219} \ldots x_t]$$

consisting of symbols produced by the downstream detector to the text

$$T = [y_{t-50399} \ldots y_{t-1800}]$$

consisting of symbols produced by the upstream detector. We look for approximate matches of the pattern at locations along the text using a scoring matrix that can be fine tuned to perform well for the particular detector pair under consideration.

To describe how the computation progresses, note that the first symbol of the pattern, i.e. its prefix of length 1, is available at time $t - 7219$. Also, at this time the prefix

$[y_{t-50399} \ldots x_{t-7219}]$ of the text, of length 43201 is available. We may therefore compute all the numbers $V(i,j)$ for $i = 1$ and $1 \leq j \leq 43201$. At any time of type $t-u$, $1800 \leq u \leq 7219$, we may compute the $V(i,j)$ for $i = 7200-u, 1 \leq j \leq 50,400-u$ and for $1 \leq i \leq 7200-u, j = 50,400-u$. For any time of type $t-u$, $0 \leq u \leq 1799$, we may compute the $V(i,j)$ for $i = 7200-u, 1 \leq j \leq 50,400$. This way, all the computations are complete at time t, and, by recording the values of j for which $V(7200,j)$ exceeds the threshold σ we have determined all the locations in the text where some substring that approximately matches the pattern ends. We will of course now have to choose an algorithm to use this data to determine whether to signal than an incident has been detected. Note that the computation has proceeded *in real time*. Note that, at any time, at most $50,400$ evaluations of the update rule of the dynamic programming algorithm need to take place.

We can also have parallel versions of this comparison running simultaneously, e.g. if we want updates every 30 seconds instead of every 2 minutes, while still using a pattern corresponding to observations over a length of time of 2 minutes, we would, at time t, be running comparisons for each of the patterns

$$P_k = [x_{t-1800k-7219} \ldots x_{t-1800k}]$$

for $k = 0,\ 1,\ 2,\ 3$, simultaneously, against the corresponding text

$$T_k = [y_{t-1800k-50399} \ldots y_{t-1800k-1800}]\ ,$$

$k = 0,\ 1,\ 2,\ 3$. Thus, at most about 200,000 evaluations of the update rule of the dynamic programming algorithm need to take place at any given time. This is something that it would seem could be quite easily handled by today's computing technology

We have emphasized the dynamic programming approach because of its obvious real time nature. This approach does, however, result in a $\Theta(nm)$ algorithm. In closing, we would also like to suggest and algorithm known for approximate pattern matching problem that use exact matching as an ingredient, and which we believe could result in much better complexity estimates in typical problems. This algorithm is basically the *Baeza-Yates Perleberg algorithm* (BYP algorithm) [5], [1].

Assume, for convenience, that the scoring matrix is such that exact matches receive score 0, while inexact matches receive a negative score. We wish to determine the locations in the text where the pattern approximately matches with a score of at least σ. Thus σ is a strictly negative number. Let $\mu < 0$ be such that $\mid \mu \mid$ is the smallest of the absolute values of the scores corresponding to inexact matches. Let $k = \mid \frac{\sigma}{\mu} \mid$. Let $r = \lfloor \frac{n}{k+1} \rfloor$. We break the pattern (which, we recall, is of length n) into $k+1$ contiguous subintervals each of size r and whatever is left over; this is possible by the definition of r. We now observe that if the pattern approximately matches a location $T[i, \ldots j]$ in the text with a score of at least σ, then at least one of the $k+1$ subintervals of the pattern of size r must exactly match a corresponding portion of the text. Thus, we may proceed as follows (this is essentially the *Baeza-Yates Perleberg algorithm* (BYP algorithm), [1]) : We first look for locations in the text that *exactly* match one of these $k+1$ subintervals of the pattern of size r. Having found these locations, for each of these, we focus on the portion of the text of length $2n+r$ around this location and carry out an approximate matching algorithm for the entire pattern

against this portion of the text (for this purpose we could use a dynamic programming based algorithm, such as the one considered earlier, if desired). The complexity of each such step is $\Theta(n^2)$, and the number of such steps necessary will depend on how many exact matches are found in the text with the subintervals of the pattern. Since $n << m$, we expect that the overall complexity of this algorithm would typically be less that of dynamic programming for the pattern against the full text.

Preliminary implementations

We implemented a preliminary version of our ideas on the I-880 data. We worked with an undergraduate, Luis Gutierrez, who has since graduated. Luis provided C++ code for the implementations that he ran. The code, together with details of how to use it is available at <<http://www.eecs.berkeley.edu/ ananth/mou223 >>. It is meant to work on both ploop and floop data. Unfortunately, this code appears to be somewhat buggy as of the current moment. The basic idea of what we did in this implementation is described in the following section. The I-880 data is not a very good data set on which to test our ideas, because it does not provide a good set of feature vectors. We therefore feel it is more valuable to await the arrival of better data from projects such as [12] and [14] to test the dynamic programming method.

Basically, we attempted to run a version of the dynamic programming based algorithms to get sets of shift between the upstream and downstream detectors which correspond to strong correlations in the observations - such shifts are potential values for the travel time, as discussed above. We did not attempt to discuss how one could go from these sets of potential travel time values to an actual estimate of the travel time, neither to an actual decision as to whether an incident has occurred or not. To make this aspect of the investigation worthwhile, it will be necessary to work with better data. To try to create more features from the data, we used a "reinforced matching" idea. Roughly what this does is to weight matches that correspond to "long" vehicles in a "reinforced" way that depends on their length. This methodology was also suggested by the visual observations of Ben Coiffman from the I-880 data that long vehicles such as trucks tend to have more stable travel times, [3].

The streams that our estimators use have values of 0 or 1 depending on wether a car was passing through the detector at any give time. The delayed stream is shifted a number of times with respect to the non-delayed stream and a sum is computed for each shift. This sum is stored on an element of a sum array indexed by the shift amount. The increment between successive shifts is usually 1 sample although it can be changed by a parameter; the max and min shifts can also be set by parameters. For each match sum[shift amount] is incremented by a number. If it's a regular estimator this is 1; for a linear estimator this is initially 1 and for each consecutive match a 1 is added; for an exponential estimator this is initially 1 and for each consecutive match the increment doubles (this saturates after some point) If the matches are not consecutive the increments are reset to 1.

An example to clarify what is done in this implementation is also worked out in the document at <<http://www.eecs.berkeley.edu/ ananth/mou223 >>.

References

[1] Baeza-Yates, R. A., and Gonnet, G. H. (1994). Fast string matching with mismatches, *Information and Computation*, Vol. 108, No. 2, pp. 187 -199.

[2] Boyer, R. S., and Moore, J. S. (1977). A Fast String Searching Algorithm. *Communications of the ACM*, Vol. 20, pp. 762 -772.

[3] Cassidy, M. et al. (1997). Automated Travel Time Measurement Using Vehicle Lengths From Loop Detector Speed Traps. *PATH MOU 352.*

[4] Gattis, J. L., and Lee, C. E. (1992). Testing Photoelectric sensor system to classify vehicles. *Journal of Transportation Engineering*, Vol. 118, No. 3, pp. 457 -471.

[5] Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology.* Cambridge University Press, New York.

[6] Karp, R. and Rabin, M. (1987). Efficient Randomized Pattern Matching Algorithms. *IBM Journal of Research and Development*, Vol. 31, pp. 249 -260.

[7] Knuth, D. E., Morris, J. H., and Pratt, V. B. (1977). Fast Pattern Matching in Strings. *SIAM Journal of Computing*, Vol. 6, pp. 323 -350.

[8] Kühne, R. R. (1993). The principle of Section-related traffic variables detection. *Preprint.*

[9] Lu, Y.-J., Hsu, Y.-H., and Maldague, X. (1992). Vehicle Classification using Infrared Image Analysis. *Journal of Transportation engineering*, Vol. 118, No. 2, pp. 223 -240.

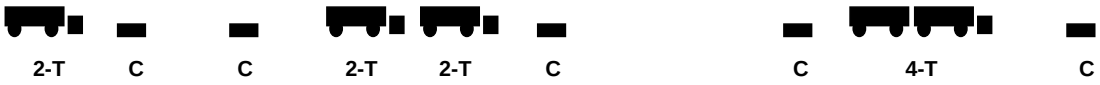
[10] Malik J. et al. (1997). Development of Binocular Stereopis for Vehicle Lateral Control and Obstacle Detection. *PATH MOU 306.*

[11] PATII (1996). Automatic Incident Detection Algorithms. *Survey*, California PATII Project. Available at <<http://www.path.berkeley.edu/~ leap/TTM/Incident_Manage/Detection/aida.html>>.

[12] Ritchie, S. et al. (1997). Section-Related Measures of Traffic System Performance: Field Prototype Implementation. *PATH MOU 336.*

[13] Solomon, M. (1991). A Review of Automatic Incident Detection Techniques. *Technical Report*, Transportation Center, Northwestern University. Available at <<http://ais.its-program.anl.gov/reports/travel.time.software.html>>.

[14] Varaiya P. et al. (1997). Real-Time Algorithms for Travel Time and Origin-Destination Estimates, Incident and Verification. *PATH MOU 353.*



View at detector at one location along freeway

View at detector at another location along freeway

Figure 1: Illustrating the pattern matching problem

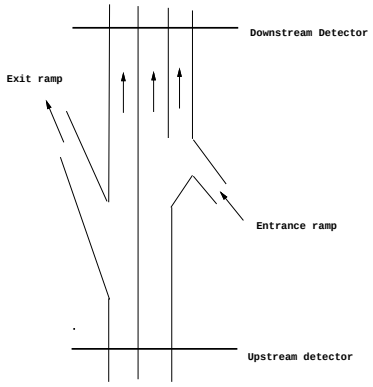


Figure 2: Illustrating a typical section of freeway

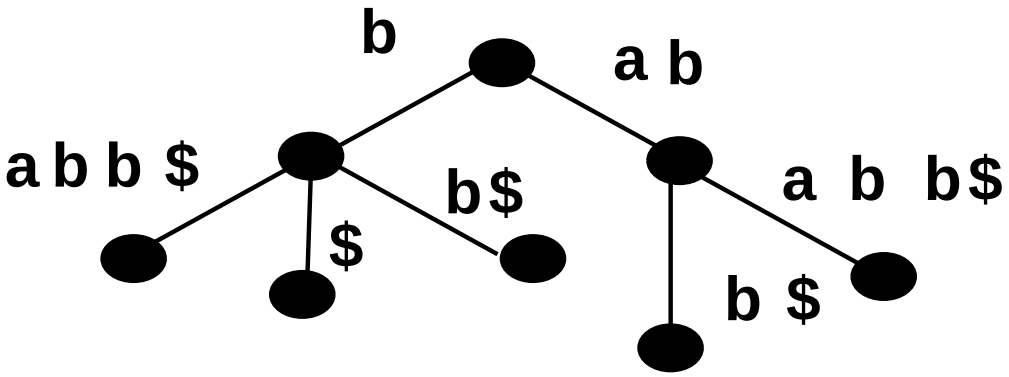


Figure 3: The suffix tree of [a b a b b]

http://www.eecs.berkeley.edu/~ananth/mou223.html

*THE PROGRAM.

A makefile is included in the src directory so to compile just type 'make' this will produce the executable 'est'. To erase the object and executables type 'make clean'. The programs compiles with g++.

For each estimator and for each iteration the program reads a given number of samples from both streams. The streams that the estimator uses have values of 0 or 1 depending on wether a car was passing through the detector at any give time(how these 0,1 streams are generated from floop files is explained later). The delayed stream is shifted a number of times with respect to the non-delayed stream and a sum is computed for each shift. This sum is stored on an element of a sum array indexed by the shift amount. The increment between successive shifts is usually 1 sample although it can be changed by a parameter; the max and min shifts can also be set by parameters. For each match sum[shift ammount] is incremented by a number. If it's a regular estimator this is 1; for a linear estimator this is initially 1 and for each consequtive match a 1 is added; for an exp. estimator this is initially 1 and for each consecutive match the increment doubles(this saturates after some point) If the matches are not consecutive the increments are reset to 1. For example, with the given streams, max shift = 2, min shift = 0, shift increment = 1 and num samples = 4 this happens:

```
delayed stream     : <most recent sample> 0 1 1 0 1 1 0 0 0 1 1 0<oldest sample>
non-delayed stream: <most recent sample> 1 1 0 1 1 0 1 1 0 0 0 1<oldest sample>
                                 sum array for          sum array for
iteration 1:(assume x= 0)              regular               linear
                                sr[0 | 1 | 2 | 3]   sl[ 0  | 1 | 2 |  3]
delayed stream : 0 1 1 0
non-dlyd stream: 0 0 0 1 x x x      0                    0

delayed stream :   0 1 1 0
non-dlyd stream: 0 0 0 1 x x x         1                    1

delayed stream :     0 1 1 0
non-dlyd stream: 0 0 0 1 x x x            1                  1

delayed stream :       0 1 1 0
non-dlyd stream: 0 0 0 1 x x x               0                   0

iteration 2

delayed stream : 1 1 0 0           1                   1
non-dlyd stream: 1 0 1 1 0 0 0

delayed stream :   1 1 0 0
```

non-dlyd stream: 1 0 1 1 0 0 0 2 2

delayed stream : 1 1 0 0
non-dlyd stream: 1 0 1 1 0 0 0 3(+1+1) 4(+1+2)

delayed stream : 1 1 0 0
non-dlyd stream: 1 0 1 1 0 0 0 1(+1) 1(+1)

iteration 3

delayed stream : 0 1 1 0
non-dlyd stream: 1 1 0 1 1 0 1 2(+1) 2(+1)

delayed stream : 0 1 1 0
non-dlyd stream: 1 1 0 1 1 0 1 3(+1) 3(+1)

delayed stream : 0 1 1 0
non-dlyd stream: 1 1 0 1 1 0 1 5(+1+1) 7(+1+2)

delayed stream : 0 1 1 0
non-dlyd stream: 1 1 0 1 1 0 1 2(+1) 2(+1)

after each iteration the program prints to the output file the indexes of the elements of s >= max_val_in_s*threshold. So if threshold was 1.0 then the linear estimator will output:

iteration 1:

0 1
0 2

iteration 2:

1 2

iteration 3:

2 2

the number in the first colum is the iteration number - 1.

Also, there's a way to reset the sum array to 0 after X iterations. Reseting the sum array periodically "should" allow the estimator to track variations in delay. The problem with this is that it can't be done too often; i don't know how often it can be done.

*THE DATA I USED TO GET THESE RESULTS:

I did not get any ploop files from Petty and none of the people that replied to the e-mail i sent 3 months ago asking for the format of the raw data files seemed to know either. The raw data contains the time pulses in 60ths of seconds but Petty's program only considers output periods larger than 1 second so the data you get has samples spaced 1 second appart. One of the reasons the estimators do worst with the floop files is that there's less data than with the ploop files(those have samples 1/60th second apart); at least i get better results with the ploop files i got from you(the problem is that they don't include congestion).

I used floop files for lane 4 extracted from data for 3/11/93 and i only tried detectors 2,11,18, and 6 because on the southbound direction there are times(at about 8:30am or about 31000 seconds after midnight) where the speed drops to ~10MPH. The estimation is done from 8:00 - 9:00 in 3 minute intervals(180 samples per iteration) and the sum array is reset after every 10th iteration(30 minutes). All floop files are text files so you can look at them using emacs or some other editor.

There are two types of floop files on the stuff that i'm sending you. The first type contain speed of cars going through the detector. These files are named floop<number>.{n,s}s4 ; you can open floop11.ss4 and you'll see a drop in the speed at about 8:30 or 8:40.

The second type is named floop<number>.{n,s}c4 where number is a detector number and n = northbound s = southbound. The first column contains the number of seconds after midnight and the second column contains the number of pulses during each 1 second interval. You sometimes see two consecuitve 0.5 's because(i believe) the pulse started on 1 interval and ended on the next. As i said before, the estimators work on streams of 1's and 0's so i read the floop files and output a 1 if the value is non-zero and a zero otherwise.
So the following :

28800 0.5 (1 rising edge)
28801 0.5 (1 falling edge)
28802 0
28803 0
28804 2.5 (3 rising 2 falling edges)
28805 0.5 (1 falling edge)

gets turned into: 1 1 0 0 1 1 and fed to the estimator.

If i knew how to get data with 1/60th second resolution then within each interval there would be only one 0.5(for a rising or fallling edge) so the stuff would look like :

0.5
0

0
0.5
0
0
0
0.5
0
0.5

and would get turned into 1 1 1 1 0 0 0 1 1 1. This would provide much more data for the estimator to work on and the quality of the estimates may improve.

*HOW TO RUN THE PROGRAM:

The name of the executable is 'est' and is located in the src directory. This name can be changed in the makefile if you want.

The program requires a setup file. If no setup file is given as an argument the program tries to open 'setup.txt' and if not present it exits.
The format for this file is as follows:
(<foo> = foo is required; [foo]= foo is optional; 'foo' = 'foo' is a keyword)

-one or more <param> = <value> pairs.

-at most one of the following "<space><command string for each detector><space>" (including the quotes)

-'start'

-for each estimator:
<'rxy'||'rxylin'||'rxyexp'> <'ploop'||'floop'> <upfilename>
<downfilename> <min> <max>
<inc> <num after which to reset> <num of samples to read>
[output file name]
'end'

<param> can be 'iter', 'threshold', 'filename'

'iter' = number of times to run each estimator

'threshold' = report values >= threshold*maxval so to find out the maximum values after each estimation set 'threshold' = 1.0.

'filename' = if an estimator doesn't specify an output file the output will be stored in 'filename'

for the estimators:

'rxy' is the contant increment estimator(for each match increment sum by 1)

'rxylin' is the linear increment estimator(for each consecutive match increment sum by previous increment+1)

'rxyexp' is the exponential increment estimator(for each consecutive match increment sum by double previous increment)

'ploop' specifies an input file in ploop format

'floop' specifies an input file in floop format

upfilename = the delayed file

downfilename = the non-delayed file

min = minimum delay to consider

max = maximum delay to consider

inc = shift interval between estimations

num after which to reset = zero sum array after this many iterations

num of samples to read = read this many samples during each iteration

output file name = store the results in this file

look at the setup files in the floop directory if you want to see examples:

'rsetup.txt' = setup file for regular estimator
'lsetup.txt' = setup file for linear estimator
'esetup.txt' = setup file for exp. estimator

The data is stored in :

rxy for regular estimator
rxylin for linear estimator
rxyexp for exp. estimator

The naming conventions for the output files is:

[r|l|e]_<non_delayed_stream>_<delayed_stream>.[n|s]

so e_2_11.n = the results produced by running the program on the stream generated from detectors 2 and 11 for the northbound direction assuming that stream from

detector 11 is delayed with respect to stream from detector 2.
With the included setup files the following files get produced:

{r,l,e}_2_11.n	{r,l,e}_11_2.s
{r,l,e}_6_18.n	{r,l,e}_6_18.s

I think the files can still be viewed using the xgraph -P -nl <filename> command (i'm not sure if its 'p' or 'P') but the last time i did this was a while ago. Now i just look at the output files on a text editor to see what the output looks like because i don't have xgraph.

The output files are text and they have the following format:

a string with the detector number(they're numbered from 0 - N with 0 being the first detector listed in the setup file)

one or more of the following lines:
<iteration - 1> n where n is such that s[n] >= threshold*max_val_of_s[]

and repeat for each detector.