Trajic: An Effective Compression System for Trajectory Data

Aiden Nibali and Zhen He

Abstract—The need to store vast amounts of trajectory data becomes more problematic as GPS-based tracking devices become increasingly prevalent. There are two commonly used approaches for compressing trajectory data. The first is the line generalisation approach which aims to fit the trajectory using a series of line segments. The second is to store the initial data point and then store the remaining data points as a sequence of successive deltas. The line generalisation approach is only effective when given a large error margin, and existing delta compression algorithms do not permit lossy compression. Consequently there is an uncovered gap in which users expect a good compression ratio by giving away only a small error margin. This paper fills this gap by extending the delta compression approach to allow users to trade a small maximum error margin for large improvements to the compression ratio. In addition, alternative techniques are extensively studied for the following two key components of any delta-based approach: predicting the value of the next data point and encoding leading zeros. We propose a new trajectory compression system called Trajic based on the results of the study. Experimental results show that Trajic produces 1.5 times smaller compressed data than a straight-forward delta compression algorithm for lossless compression and produces 9.4 times smaller compressed data than a state-of-the-art line generalisation algorithm when using a small maximum error bound of 1 metre.

Index Terms—Trajectory compression, spatial databases

1 Introduction

The need to store vast amounts of trajectory data becomes more problematic as GPS-based tracking devices become increasingly prevalent. A trajectory comprises of a sequence of timestamped sample points, each of which contains a time, latitude and longitude. Using this representation a fleet of one thousand cars each recording a sample per second requires close to 2 GB of storage space per day. Databases which archive a backlog of historical GPS data quickly become encumbered by such vast amounts of data. To minimise the impact of this problem, data compression algorithms can be applied to reduce storage requirements. Two contrasting state-of-the-art approaches for compressing trajectory data are line generalisation and delta compression.

There are a large number of algorithms that take the line generalisation approach to compressing trajectories [4], [11], [10], [15]. Line generalisation represents a trajectory using a series of linear segments created by joining selected trajectory points. Compression is achieved by discarding unselected trajectory points. A maximum error threshold is enforced by ensuring that the points discarded are within range of the line segments covering the points. The main problem with this approach is that it can only achieve a good compression ratio when successive data points are close to being linear. However, real-life trajectories can contain a lot of curvature and noise, sometimes accompanied by low sampling rates which amplify the negative effect of curves. The only way to improve compression in these situations is for the user to specify a large error margin (such as over 20 meters), but many users are unwilling to forgo such large amounts of precision.

Delta compression achieves lossless compression by storing the difference between successive data points in a trajectory rather than the points themselves. Since these deltas are typically small, they may be encoded to save space. In contrast to line generalisation, delta compression is still able to achieve reasonable compression ratios despite any noise or curvature because it only requires successive points to be close to each other rather than to lie within a straight line. Furthermore, line generalisation achieves no compression for a point that lies only a little outside the maximum error bound, whereas delta compression is able to smoothly degrade its compression ratio if points become further apart.

Figure 1 illustrates the tradeoff between compression ratio and maximum error for line generalisation, delta compression, and line generalisation enhanced by delta compression. As we can see on the graph, delta compression (the single point labeled as Delta) has a zero error but a relatively poor compression ratio. In contrast line generalisation can achieve a good compression ratio but only after the maximum error is relatively high. Even a combination of line generalisation and delta compression does not show any improvement in the compression ratio until the maximum error is relatively high. Therefore from the figure we can see that there is currently an unfilled gap in which the user is willing to tolerate only a small error in exchange for large improvements in the compression ratio.

We have designed an algorithm called "Trajic" which can fill the above mentioned gap. Figure 1 shows Trajic can achieve a good compression ratio in situations where the user is willing to give up only a small amount of precision. In fact, even at zero error Trajic is able to achieve a relatively good compression ratio. Trajic has two core components: a

• *Aiden Nibali and Zhen He are with the Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, VIC, Australia.*
E-mail: dismaldenizen@gmail.com, z.he@latrobe.edu.au

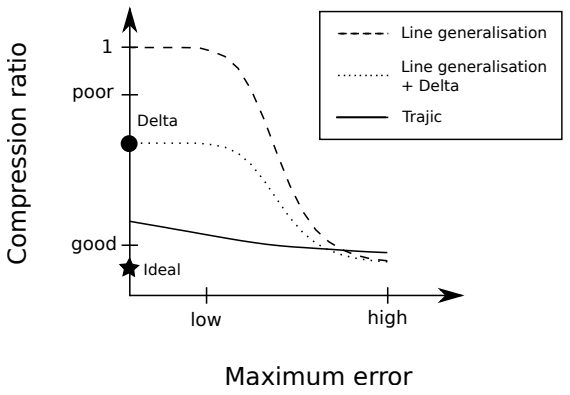


Figure 1. An abstract view of compression algorithms with respect to a theoretical ideal (produced based on results shown in Figure 15)

predictor and a method for encoding residuals. The predictor guesses the value of the next data point and then generates small residuals representing the difference between the predicted and actual values. The residual encoding scheme attempts to use as few bits as possible to encode the number of leading zeros in the residuals, and is able to achieve lossy compression by discarding a number of least significant bits. In Trajic we limit the number of bits discarded to ensure that the maximum user defined error margin is not violated.

Our Trajic compression algorithm is developed by extensively studying alternative algorithms for predictors and leading zero encoding schemes. A comparison was made between five different prediction algorithms ranging from the simple "constant predictor" (used in existing delta compression) to the more conceptually complicated cubic spline predictor. It was found that a simple and fast predictor, the "temporally-aware linear predictor", results in the most accurate predictions. Due to these results, the Trajic system utilises this predictor.

We found different existing leading zero encoding schemes to be optimal for different actual residual lengths (the length of a residual without counting leading zeros). Furthermore, we discovered that in real data sets residual lengths have a skewed distribution. This led us to develop a novel method superior to existing fixed schemes for encoding leading zeros. Our encoding method adapts to the distribution of residual lengths by incorporating a simple pre-pass frequency analysis step.

Finally, Trajic has linear run-time complexity ($O(N)$) which is the same as simple delta compression. In contrast line generalisation algorithms range in complexity from $O(N \log(N))$ to $O(N^2)$.

Extensive experiments using real data sets show that Trajic can produce 1.5 times smaller compressed data than a straightforward delta compression algorithm (used in systems such as TrajStore [5]) for lossless compression and 9.4 times smaller compressed data than a state-of-the-art line generalisation algorithm when using a 1 metre error bound.

In summary, this paper makes the following three main contributions.

- We identify that neither of the existing methods of line generalisation or delta compression, nor a combination of the two, allow users to trade a small amount of accuracy for large gains in the compression ratio. This is a result of a systematic analysis and experimentation of existing state-of-the-art line generalisation algorithms and existing delta compression algorithms.
- We develop the Trajic system to fill this important gap in the existing literature. Within the Trajic system our main claim to novelty is the creation of a very effective and novel residual encoding scheme. A related contribution is making the delta compression scheme span the spectrum between lossless and lossy compression.
- We conduct extensive experiments demonstrating the effectiveness of Trajic when applied to real-world data sets. We systematically compare the different algorithm across the three important metrics of compression/decompression time, compression ratio and error margin.

This paper begins by briefly discussing existing related work and their shortcomings in Section 2. Section 3 describes the Trajic system, starting with the predictor then continuing with the generation of residuals, leading zero encoding and finally the method for storing residuals. Section 4 presents the experimental results and finally Section 5 concludes the paper.

2 Related Work

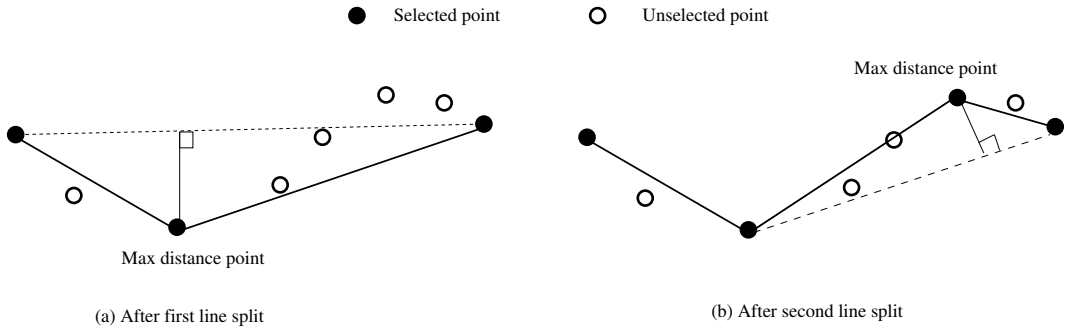
As mentioned in the introduction there are two main approaches to compressing trajectories: line generalisation and delta compression. We will review the existing work in both areas in addition to some other methods that require knowledge of a road network.

2.1 Line generalisation

One of the most commonly utilized trajectory compression schemes is line generalisation [4], [11], [10], [6], [13], [21]. The line generalisation method originated from solving the problem faced by cartographers who wanted to use computers to extract features from detailed data and represent them using simple and readable maps. An important part of representing linear features is to solve the line generalisation problem.

The line generalisation problem can be defined as follows. Given an ordered set of $n+1$ points in the plane, $V_0, V_1, ...V_n$, let C, containing $\overline{V_0V_1}, ..., \overline{V_iVi+1}, ..., \overline{V_{n-1}V_n}$, be a chain with n line segments. The problem is to find a modified C' containing less segments which approximates C within an acceptable error margin, where the error margin is usually defined as the maximum perpendicular distance from C' to C.

One of the most popular line generalisation algorithms is the top-down Douglas-Peucker (DP) algorithm [6], illustrated in Figure 2. At the beginning the entire set of points is approximated using a one line segment connecting the first and last points of the set.



(a) After first line split

(b) After second line split

Figure 2. Example illustrating the Douglas Peucker top-down line generalisation algorithm.

Next the single long line segment is replaced by two shorter line segments in which the end of the first line segment and the start of the second line segment correspond to the point that is the furthest (highest perpendicular distance) from the original single long line segment. This is recursively repeated until there are no points whose maximum perpendicular distance is higher than a user-defined threshold from the line segment covering the point. The original DP algorithm has $O(N^2)$ time complexity, however, various improvements have been proposed, including the one by Hershberger et al. [8], which has a time complexity of $O(N \log(N))$.

Another approach for solving the line generalisation problem is the opening window approach [11]. The opening window approach is an incremental algorithm that starts with a window that contains the first three points of the trajectory and then progressively "opens" the window until a single line segment can no longer represent all of the contained points accurately enough. Once this happens, a single line segment from the first point of the window to the second last point in the window is used to approximate the current window and the second last point in the current window becomes the start of the next window. This is repeated until the entire set of points have been connected by line segments. The opening window algorithm has $O(N^2)$ worst case time complexity.

When line generalisation methods are applied to achieve trajectory compression, the time dimension must be incorporated into the error metric. One way of incorporating time is to use the synchronised euclidean distance (SED) [18]. SED measures the distance from the original point to the approximated point adjusted to the same timestamp. Another metric is the Meratnia-By time-distance ratio [11] where spatial and temporal information are both used separately to decide whether a point is kept or discarded. For the temporal component, the ratio of the time taken to travel the original versus approximated trajectory is used. For the spatial component, the position of the original point is compared to its approximated position within the compressed trajectory.

The main limitation with the line generalisation approach stems from the fact that it uses straight lines to approximate trajectories. Therefore, it works best when trajectories are mostly linear. But in real life there are a lot of non-linear segments in trajectories due to turning, noise and low sampling rates. Line generalisation requires very large error margins to achieve a good compression ratio in these scenarios.

2.2 Delta compression

The standard simple delta compression algorithm stores a delta for each timestamp in the trajectory. The delta d_i for the i^{th} timestamp equals p_i - p_{i-1} where p_i and p_{i-1} are the trajectory point values at timestamp i and $i-1$ respectively. The above is typically performed separately for each of the three components of time, longitude and latitude of the trajectory.

Existing literature in trajectory compression has almost exclusively focused on line generalisation methods, hence delta compression has been largely ignored in the past. However, the TrajStore [5] state-of-the-art trajectory database system only employs the standard simple delta compression method defined above. This simple method suffers from the following three shortcomings: it 1) employs a constant prediction policy; 2) uses a static leading zero encoding scheme; and 3) only performs lossless compression. The first shortcoming means TrajStore's delta compression scheme essentially predicts the next data point to be exactly the same as the current point and stores the difference as a delta. In contrast, this paper explores a number of different prediction models and conclude a temporally-aware linear predictor gives smaller deltas and hence better compression ratios. The second shortcoming of TrajStore's delta compression scheme is that it uses a static scheme to encode leading zeros. In contrast, this paper analyzes a number of different possible leading zero encoding schemes and concludes that our novel dynamic scheme gives the best performance. Finally, TrajStore's delta compression scheme only does lossless compression. In contrast our Trajic performs both lossless and lossy compressing and hence can directly compete against the popular line generalisation methods which are all lossy compression methods.

In addition to delta compression, TrajStore also compresses trajectories by clustering similar trajectories together and storing only one representative trajectory per cluster. This is a complementary technique that may also be used in conjunction with our Trajic algorithm.

2.3 Other methods

There are other systems which use knowledge of predefined road networks and tracks to minimise

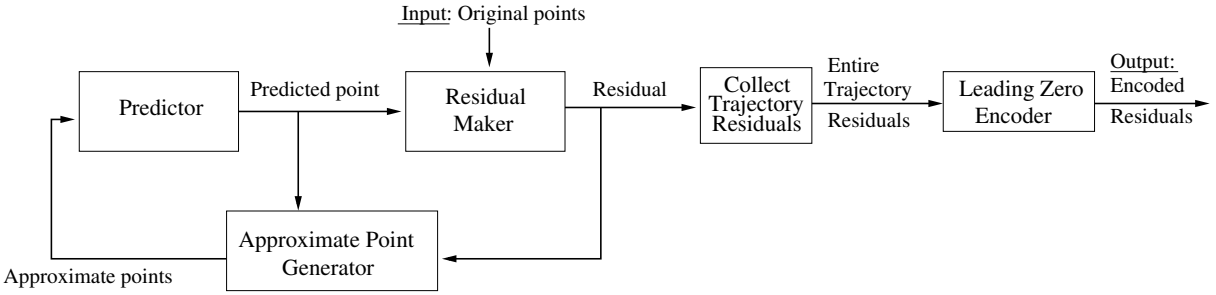


Figure 3. The system architecture used to generate the encoded residuals (compressed data).

the space required when storing trajectories [3], [2]. However, such systems are not very useful when the network data is not available, or the moving objects are not confined to well-defined tracks. The Trajic compression scheme does not require such additional network information to compress trajectories.

Mahrsi et al. [7] proposed a one pass sampling based method for trajectory compression which selectively decides if individual points should be discarded based on whether the point can be predicted within a user defined error. They claim this method is more computationally efficient compared to the line generalisation methods but at the same time allows an upper error bound to be set for compression error. In the paper they propose the use of a predictor which is the same as the temporally-aware linear predictor (See Section 3.2.3) that we found provides best prediction performance. However, Mahrsi et al. use the predictor to decide if the next sample point should be dropped or not. The next sample point is dropped if the predicted point is within a user-defined maximum error margin of the actual point. In contrast, we use the predictor to create residuals (difference between actual and predicted value). Therefore, the system proposed by Mahrsi et al. can be thought of as using the temporally-aware linear predictor to create a one pass line generalisation algorithm, whereas Trajic uses the temporally-aware linear predictor in the context of delta compression. In addition we perform a comprehensive study comparing five different predictors whereas Mahrsi et al. do not compare their predictor against any other predictor.

Muckell et al. [14] proposed SQUISH, a one pass trajectory compression algorithm that is based on the line generalisation approach of discarding data points while minimizing the increase in compression error. They consider a current buffer worth of points and prioritize to keep points within that buffer that are extreme points based on the local estimation of the error. SQUISH improves on line generalisation techniques by incurring much lower execution time due to the fact it takes only a single pass through all the data. However, it still offers similar compression ratios and compression error behavior when compared to line generalisation algorithms such as the Douglas-Peucker (DP) algorithm [6]. Therefore it still does not fill the gap of a good compression ratio and low compression error.

3 THE TRAJIC SYSTEM

3.1 Overview

The Trajic system takes as input the original trajectory consisting of a sequence of sample points $p_1, p_2, p_3, ...$, with each point consisting of a timestamp, latitude and longitude. During compression, Trajic generates an approximate trajectory consisting of the approximate points $p'_1, p'_2, p'_3, ...$, whose error is within user defined bounds. In the case where the user requests lossless compression the approximate trajectory will be identical to the original trajectory. A sequence of residuals $r_1, r_2, r_3, ...$ is incrementally produced by Trajic during the compression process. These residuals are stored and used to reconstruct the trajectory during later decompression.

Figure 3 shows the system architecture including all the components used to generate the residuals from the original points. The system consists of the following four key components: the predictor; the approximate point generator; the residual maker; and the leading zero encoder. The predictor takes the previous sequence of approximate points ($p'_1, p'_2, p'_3, ...p'_{i-1}$) to generate the next predicted point (α). The residual maker stores the difference between the actual original point (p_i) and the corresponding predicted point (α) to create the residual (r_i). The residual (r_i) is fed back to the approximate point generator and combined with the predicted point (α) to generate the i^{th} approximate point (p'_i). The reason that the i^{th} original point (p_i) is different from the approximate point (p'_i) is that the residual (r_i) may contain inaccuracies when using lossy compression. Algorithm 1 summarises this compression process in high-level pseudocode. Decompression follows logically from compression and is described in algorithm 2.

We have engineered our system so that errors do not propagate at all between successive points. This is achieved by creating approximate points, just as the decompressor would, and using these as feedback in the compression process so that inaccuracies may be compensated for. For example, assume we are using the constant predictor and the original values are $p_1 = 3, p_2 = 3$, but due to losses in the residual the approximate value p'_1 is 2 instead. When we run the constant predictor, we will predict the next value to be $\alpha = 2$ (based on p'_1) rather than 3 (based on p_1). Since the next residual generated will be based on the difference between our predicted point ($\alpha = 2$) and the actual point ($p_2 = 3$), we will end up with a larger residual to compensate for the fact we lost precision

earlier on. This effectively prevents the errors from propagating.

As shown in Figure 3, once all of the residuals for a trajectory have been calculated, a leading zero encoding algorithm is used to compact the leading zeros. The reason we collect the entire trajectory before encoding the leading zeros is that our leading zero encoding algorithm uses the statistics of the entire sequence of residuals to find the optimal encoding scheme for a particular trajectory. Each set of residuals (time, latitude and longitude) are encoded separately and then interleaved during writing. These residuals, along with any initial reference points required by the predictor, contain enough data to recreate the entire trajectory.

Query processing. Normally, the delta-based compression techniques including Trajic need to decompress the entire trajectory before query processing can occur. This is because each residual value depends on successive previous residual values. However, partial decompression is possible if we store the actual uncompressed point at regular intervals in the compressed trajectory. This allows decompression to start at the uncompressed points. The line generalisation based compression methods can be partially decompressed by just decompressing the data points around the query interval.

Algorithm 1: High-level compression pseudocode

Input: A sequence of timestamped points
Output: A compressed representation of the trajectory
Function Compress($p_0, p_1, \ldots, p_n$) **begin**
  for $i \leftarrow 0$ **to** n **do**
    $\alpha \leftarrow$ PredictNext($p'_0, p'_1, \ldots, p'_{i-1}$)
    $r_i \leftarrow$ CalculateResidual(α, p_i)
    `// Discard bits from residual for lossy compression`
    $r_i \leftarrow$ DiscardBits(r_i)
    $p'_i \leftarrow$ RestoreResidual(α, r_i)
  return Encode($r_0, r_1, \ldots, r_n$)

Algorithm 2: High-level decompression pseudocode

Input: A compressed representation of the trajectory
Output: A sequence of timestamped points
Function Decompress($data$) **begin**
  $r_0, r_1, \ldots, r_n \leftarrow$ Decode($data$)
  for $i \leftarrow 0$ **to** n **do**
    $\alpha \leftarrow$ PredictNext($p'_0, p'_1, \ldots, p'_{i-1}$)
    $p'_i \leftarrow$ RestoreResidual(α, r_i)
  return $p'_0, p'_1, \ldots, p'_n$

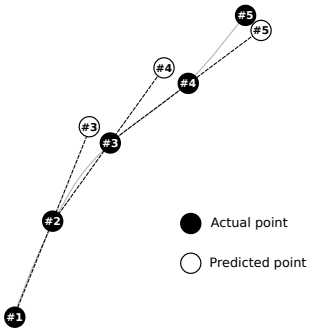


Figure 4. Visual demonstration of basic linear spatial prediction

3.2 Predicting points

In this section, we present a range of predictors, starting with the simple constant predictor and ending with our temporally-aware linear predictor. Although the predictor is a fairly rudimentary part of the Trajic system, it is necessary to select an algorithm which forms a good basis for the novel leading zero encoding scheme proposed in Section 3.3. To design a good predictor, two important factors must be taken into consideration: accuracy and efficiency. An accurate predictor produces predicted points which are close to the actual future points, thus minimizing residual sizes and reducing the final storage space required. An efficient predictor predicts points quickly. Given that a trajectory consists of a sequence of points containing time, latitude and longitude, a predictor is composed of separate functions for predicting the temporal and spatial elements of each point.

To keep the total compressor complexity at $O(N)$, only predictors which run in constant time are considered.

3.2.1 Constant predictor

The constant predictor simply predicts the next point to be the same as the current one. This essentially corresponds to delta compression, as the result is storing residuals between each pair of adjacent points in the trajectory (the "deltas").

Such a predictor is very simple to implement, and works best when samples are taken frequently with small changes between them. Realistically this is not often the case due to the costs associated with measuring and storing high-resolution data on portable GPS devices. Hence constant predictors can perform poorly in real-life data where sample rates may be low.

3.2.2 Basic linear predictor

The basic linear predictor functions by considering the previous two points and calculating the displacement between them. It then adds this displacement to the last point, thus forming a prediction. In general this performs better than the constant predictor because it takes advantage of the tendency of moving objects to have inertia. However, when points are not recorded at a constant sample rate the simple displacement-based algorithm begins to lose some of its accuracy.

For example, if the previous two points were sampled five seconds apart, but the next point was sampled only one second afterwards, this predictor will tend to overestimate the next point by four seconds' worth of displacement.

Figure 4 demonstrates the spatial prediction process of the linear predictor, with actual and approximate points labelled with a number indicating order. The displacement between actual points #1 and #2 is calculated, shown here as a dashed line. This displacement is then added to actual point #2 to find the location of predicted point #3.

3.2.3 Temporally-aware linear predictor

The temporally-aware linear predictor extends the basic linear predictor described previously by taking advantage of the fact that the timestamp of the next point can be made available when predicting its location. That is, we make predictions based on *velocity* rather than displacement, thus eliminating the loss of spatial accuracy caused by a variable sample rate. In other words, using displacement only would result in poor prediction accuracy when the sampling rate has high variance. This is because an object moving at constant velocity would move twice as far if it was sampled at time $t_c + 2\Delta t$ versus $t_c + \Delta t$, where t_c is the time of the current sample.

Timestamps are predicted by assuming a constant sample rate.

Algorithm 3: Spatial prediction function for the temporally-aware linear predictor

Input: $pastLocs$: list of locations at which previous sample points were found, $time$: time of the point we wish to find the location of
Output: $nextLoc$: predicted location of the next sample point
Function PredictLocation(pastLocs, time) **begin**
```
// Displacement between last two points
dx ← pastLocs.last − pastLocs.secondLast
// Time interval between last two points
dt ← pastLocs.last.time - pastLocs.secondLast.time
// Calculate velocity
velocity ← dx ÷ dt
// Assume constant velocity to predict the displacement to the next point
ndx ← velocity × (time − pastLocs.last.time)
// Calculate predicted location
nextLoc ← pastLocs.last + ndx
return nextLoc
```

Time	Actual latitude	Linear prediction	Temporally-aware linear prediction
0	0	-	-
5	10	-	-
6	12	20	12
8	16	14	16

Table 1
Temporal awareness has a considerable impact on the accuracy of predicted points when sample rate is not fixed

Table 1 contains a sample set of trajectory points for an object moving with constant velocity but recording positions with a variable sample rate. The basic linear predictor ignores time and assumes that the displacement between the second and third points will be the same as that between the first and second. This results in a predicted latitude of 20, which is a considerable overestimation of the actual latitude, 12. However, the temporally-aware linear predictor calculates the velocity between the first and second points, assumes constant velocity and hence accurately predicts the next latitude to be 12. This example shows the importance of giving time special consideration when forming predictions.

3.2.4 Complex predictors

There are more complex prediction algorithms that attempt to fit smooth curves to a group of sample points, including Lagrange extrapolation and natural cubic spline extrapolation [1]. In order for the entire prediction process to run in linear time it is necessary to only perform these more complicated predictions based on a fixed number of past points.

3.2.5 Comparison of predictors

Each predictor was run over a set of 213 trajectories from Illinois [17] and the average residual length for each spatiotemporal coordinate was recorded in Table 2. Surprisingly, the complex Lagrange extrapolation and spline extrapolation predictors were generally less accurate than the temporally-aware linear predictor. This is most likely due to noise in real data sets causing attempts to fit mathematical functions to yield poor results.

The Trajic system uses the temporally-aware linear predictor since it both produces accurate predictions and makes predictions quickly.

3.3 Generating residuals

A *residual* is typically a small number representing the difference between two similar values. Given either value, the other may be restored by using the residual. Although in our case the two values are the predicted and actual coordinates in the form of double-precision floating-point numbers, the algorithms presented in this section can be easily extended to compress values represented in other formats. Residuals are calculated by XORing the binary representation of these values. This works because the most important parts of a double, the sign and exponent, are stored in the most significant of the 64 bits (Figure 5). Therefore, since

	Constant	Linear	Temporally-aware linear	Lagrange	Cubic spline
Time	40.86	1.81	1.79	6.98	1.81
Latitude	34.70	30.03	29.85	38.83	32.04
Longitude	32.20	28.54	28.41	38.12	31.11

Table 2
The average residual length for each spatiotemporal coordinate produced by different predictors over all trajectories from the Illinois data set

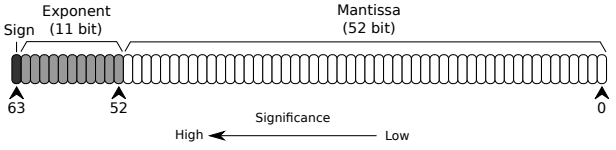


Figure 5. The relationship between bit position and significance in the IEEE 754 binary representation of a double floating-point number is monotonic

	Value	How it's stored
Value 1	27.34212484801	`40 3b 57 95 7e 79 5a 17`
Value 2	27.34212484972	`40 3b 57 95 7e 80 b2 42`
Residual	Val. 1 XOR Val. 2	`00 00 00 00 00 f9 e8 55`

Table 3
Residuals calculated between similar numbers tend to have many leading zeros

similar numbers tend to share a sign, exponent, and a few significant bits of the mantissa, residuals will tend to have many leading zeroes. An example residual calculation is demonstrated in Table 3.

3.3.1 Lossy compression

In this section we introduce lossy compression by using approximate residuals which, when restored, are within a known error margin of the original value. This enables the creation of much smaller residuals through bitshifting, *effectively discarding the D least significant bits from each residual*. Ultimately, smaller residuals require less storage space and are desirable. It is ideal to use a different value of D for spatial and temporal residuals as the range of values and error requirements for each are usually different.

Here we perform error calculations based on the IEEE 754 standard for storing double-precision floating-point numbers, but the formulae presented should apply to other representations of doubles with slight modification.

It is important to ensure that the error induced from the lossy compression stays within the user defined maximum error. The maximum error possible when discarding D bits from a double's mantissa is $error_{max} = (2^{D-52} - 2^{-52}) \times 2^q$, where q is the value of the double's exponent (which in the case of the IEEE 754 standard is the number stored in the 11-bit exponent field minus an offset of 1023). In order to calculate a safe value of D which will not exceed the

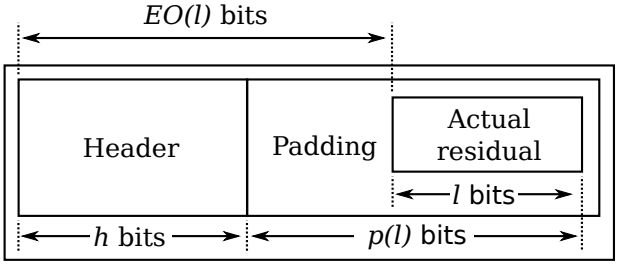


Figure 6. A general encoding scheme with important bit-lengths labelled

specified error bounds for the entire trajectory, it is necessary to know the maximum exponent possible for values in that particular trajectory, q_{max}. By rearranging the formula we get $D = \lfloor \log_2(error_{bound} \times 2^{52-q_{max}} + 1) \rfloor$

q_{max} may be established by determining the maximum absolute value in the trajectory and performing bitwise manipulation to extract the exponent (Algorithm 4).

Algorithm 4: Function for calculating $D from a specified error bound, assuming doubles are stored using the IEEE 754 standard

Input: $error$: user-specified error bound, $maxVal$: maximum value in the trajectory
Output: D: number of bits which are safe to discard from residuals
Function calculateDiscard(error, maxVal) **begin**
    `// Extract exponent value`
    q ← (maxVal >> 52) & 0x7ff − 1023
    `// Calculate safe number of bits to discard`
    D ← $\lfloor \log_2(error \times 2^{52-q} + 1) \rfloor$
    return D

3.4 Compressing residuals

Although each residual is typically a small number, they are still technically 64-bit longs. To actually achieve compression, the leading zeros of the residuals must be compactly encoded.

In general, a number stored with a leading zero encoding scheme has three sections, shown in Figure 6: a header indicating how many bits are to follow, a

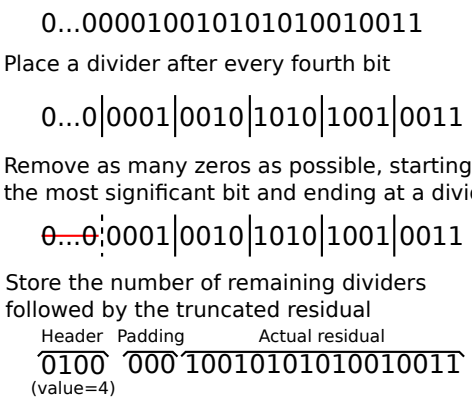


Figure 8. The process for encoding residuals with the half-byte count scheme is typical of all static encoding schemes.

small amount of zeros for padding (when the header indicates more bits than the actual residual length), and the actual data representing the residual value. Let l be the length of the actual residual in bits ($l \in [0, 64]$). Let $p(l)$ be the padded residual length in bits ($p(l) \geq l$). Let h be the length of the header in bits (the header indicates the value of $p(l)$). The aim of such an encoding scheme is to minimise the combined size of the header and padding for each l, which we will call the encoding overhead ($EO(l)$). Using the previously defined terms, the encoding overhead for a given residual length is $EO(l) = h + p(l) - l$. A perfect encoding scheme would always have $EO(l) = 0$, implying a header size of zero ($h = 0$) and no padding ($p(l) = l$). In reality this is not possible as the decoder must have some way of knowing where one number ends and the next begins. So to devise a good encoder one must reduce the header size, h, whilst still keeping the amount of required padding, $p(l) - l$, to a minimum.

3.4.1 Static leading zero encoding

Most of the current popular leading zero encoding schemes may be classified as being "static". That is, the scheme is chosen ahead of time and does not adapt to suit the particular residuals which are to be encoded. These schemes can be thought of as placing "dividers" at regular intervals, then removing as many leading zeros as possible up until a divider. The number of dividers remaining in the residual are then stored in a header before the residual. Figure 8 demonstrates these steps with the half-byte encoding scheme. Using the concept of dividers, each possible value of $p(l)$ (each possible padded length) corresponds to a divider position. Here we describe three such static leading zero encoding schemes.

Bit count. A very basic, yet still popular approach to encoding leading zeros is to use a six-bit header ($h = 6$) to store the length of the actual residual. This corresponds to placing a divider between every single bit in the residual. Since the actual length of the residual, l, is encoded in the header there is no need for padding ($p(l) = l$). Figure 9 contains an example of the bit count encoding scheme. Although having no padding was one of the requirements for an ideal encoding scheme, the constantly large header size results in a suboptimal complete encoding overhead ($EO(l) = h + p(l) - l = 6$).

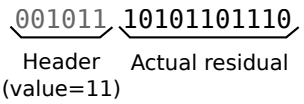


Figure 9. A common encoding scheme uses $h = 6$ bits to store a count of the following data bits. In this example the header contains the number 11, which indicates that the next 11 bits contain the actual residual.

Half-byte count. The half-byte count scheme addresses the large header issue by allowing some padding later on as a trade-off to reduce the header size. More specifically, the header stores the number of half-bytes required to store the actual residual rather than the precise number of bits. This information takes less space to store, resulting in a header size of 4 bits ($h = 4$). However, $p(l) = \lceil l \div 4 \rceil$ as a result of needing to pad residuals to the nearest half-byte. Assuming an even distribution of residual lengths, this scheme yields an average encoding overhead of 5.54 which is an improvement over the bit count scheme. The half-byte encoding process is shown in Figure 8.

Byte count. The byte count scheme is very similar to the half-byte scheme, but counts bytes instead of half-bytes. This results in a smaller header size ($h = 3$), but demonstrates that introducing too much padding has a detrimental impact on the encoding overhead. The average $EO(l)$ is 6.57 for an even distribution of residual lengths.

Analysis. The problem with each of the above three encoding schemes is that they are optimal for different actual residual lengths (l). Figure 7 shows the encoding overhead for varying actual residual lengths for all three encoding schemes. Notice the sawtooth pattern on the graph - this is a visual manifestation of the encoding overhead varying for different residual lengths.

In practice a sequence of residuals will have a variety of different actual lengths. Therefore none of the above three simple encoding schemes will be optimal for all residuals within the sequence of residuals. For example, whilst byte-counting has the highest average number of excess bits, if the majority of residuals have $l = 32$ then it will actually perform best for that particular data. This motivates our *dynamic* scheme for encoding leading zeros.

It is important to note that prior research [16] has shown that representing the compressed data at the byte level results in faster decompression when compared to bit level storage. This is because decompressing data stored at the byte level allows data to be aligned to cache lines and better fits the pipelined architecture of modern CPUs. Decompressing data at the bit level requires more CPU instructions to shift the bits to become byte aligned. In this paper, we emphasise achieving a better compression ratio while

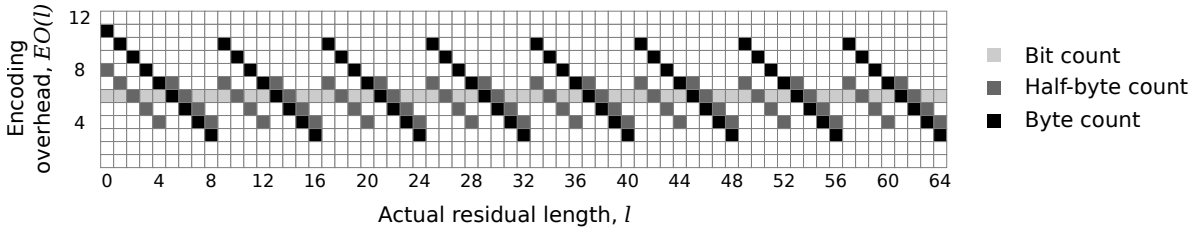


Figure 7. Different encoding schemes perform better for different l values.

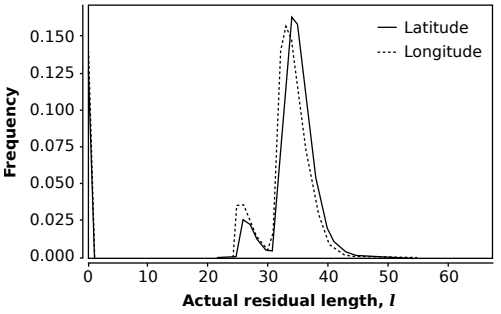


Figure 10. Distribution of latitudinal and longitudinal residuals for the Illinois data set, obtained via the temporally-aware linear predictor

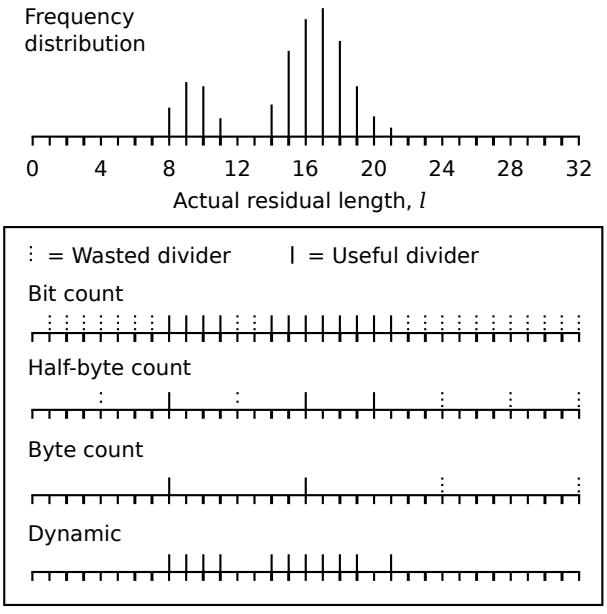


Figure 11. The dynamic encoding scheme uses knowledge of the residual length frequency distribution to avoid unnecessary encoding overhead present in static encoding schemes

incurring small error as opposed to extremely optimised decompression performance. Hence we have adopted a bit level compression algorithm because it gives a better compression ratio. However, for future work, it would be interesting to compare our algorithm to some byte level compression algorithms for all four metrics of decompression time, compression time, compression ratio, and compression error.

3.4.2 Dynamic leading zero encoding

The motivation for our dynamic leading zero encoding scheme is derived from our observation that real trajectories have skew in the frequency distribution of actual residual lengths. See Figure 10 for the frequency distribution of actual residual lengths of a trajectory in the Illinois data set used in our experiments. Notice the clear skew in the distribution.

The idea behind our dynamic leading zero encoding scheme is to use the frequency distribution information to dynamically create an encoding scheme which aims to minimise the encoding overhead for the most frequent residual lengths. That is, we intelligently place dividers where they are needed most, as illustrated by Figure 11. The figure shows that the static encoding schemes, which are oblivious to the frequency distribution of residual lengths, have many wasted dividers (dividers in areas with low value frequency) and/or large padding (long distance from high frequency values to nearest higher divider). In contrast, the dynamic scheme does not have any wasted dividers and does not require many padding bits.

The system for encoding leading zeros proposed here dynamically generates a padded length function $p(l)$ based on the frequency distribution of l values in the sequence of residuals. To ensure an optimal function is obtained, a variation of the linear partitioning algorithm [19] is employed. This results in values of $p(l)$ that are close to l for the most frequently occurring l values. Finally, Huffman coding [9] is applied to minimize the header length for most frequently used values of $p(l)$.

Given a residual length frequency distribution we aim to find the padded length function $p(l)$ which produces the lowest average number of encoding overhead bits $EO(l)$ for the trajectory. This is achieved by partitioning the frequencies using a varying number of dividers and defining $p(l)$ as a function which simply rounds l up to the next divider. Since the algorithm aims to minimise the average number of excess bits used to store residuals, the total cost of a particular partitioning must approximate the average amount of encoding overhead. We define the cost of a partitioning as the average encoding overhead of the partitioning which is expressed mathematically as

Algorithm 5: Partitioning Algorithm

```
Function Partition(freqs) begin
    // The maximum number of dividers
    maxDivs ← 32
    // 2D array of minimum cost
    costs ← Array[length(freqs)][maxDivs]
    // 2D array of divider positions
       corresponding to minimum cost
    path = Array[length(freqs)][maxDivs]
    // Boundry case for no dividers
    for i ← 0 to length(freqs) - 1 do
        costs[i][0] ← 0
        for y ← 0 to i - 1 do
            costs[i][0] ← costs[i][0] + (i - y) ×
            freqs[y]
    // Boundry case for segments of
       length zero
    for j ← 1 to maxDivs - 1 do
        costs[0][j] ← 0
    // Populate 'costs' and 'path'
    for i ← 1 to length(freqs) - 1 do
        for j ← 1 to maxDivs - 1 do
            costs[i][j] ← ∞
            for x ← j - 1 to i - 1 do
                cost ← costs[x][j - 1]
                for y ← x + 1 to i - 1 do
                    cost ← cost + (i - y) * freqs[y]
                if cost < costs[i][j] then
                    costs[i][j] ← cost
                    path[i][j] ← x
    // Find the partitioning with
       lowest cost across varying
       numbers of dividers
    dividers ← Null
    minCost ← ∞
    for n ← 2 to maxDivs do
        // Find the position of the last
           divider
        lastDiv ← length(freqs) - 1
        while lastDiv > n do
            if freqs[lastDiv] > 0 then
                break
            else
                −−lastDiv
        cost ← costs[lastDiv][n − 1] + log2(n)
        if cost < minCost then
            minCost ← cost
            dividers ← Array[n]
            dividers[n - 1] ← lastDiv
            for j ← n − 2 downto 0 do
                dividers[j] ← path[dividers[j + 1]][j
                + 1]
    return dividers
```

follows.

$$\text{cost}(d_1, ..., d_n) = \sum_{l=0}^{64} (f_l \times (p(l) - l)) + \log_2(n) \quad (1)$$

where $\{d_1, d_2, ..., d_n\}$ is a set of n dividers, f_l is the frequency with which residuals of length l occurs and

$$p(l) = \begin{cases} d_1 & l \leq d_1 \\ d_2 & d_1 < l \leq d_2 \\ ... & \\ d_n & d_{n-1} < l \leq d_n \\ \infty & l > d_n \end{cases}$$

Figure 12 shows an example of calculating the cost presented in Equation 1. In Equation 1 the header length is computed as $\log_2(n)$, which effectively assumes all dividers are represented using the same code word length. However, this is just an approximation since we actually use Huffman coding (See Section 3.4.3) to implement variable length code words.

By finding the number and arrangement of dividers with the lowest total cost, we find an efficient function $p(l)$ which can be used to compactly encode the set of residuals. This is achieved by adapting the solution to the well-known partition problem [19].

The following is an example to illustrate the partition problem. Suppose four scholars are asked to search through every page of every book in a book shelf for some important information. Further, suppose the scholars are not allowed to rearrange the order of the books and each scholar can only work with a single contiguous sequence of books. How should the books be divided so that the reading workload is most evenly spread amongst the scholars? The problem of finding the optimal divisions is known as the "partition problem".

Our problem of finding the optimal set of dividers can be mapped to the above partition problem as follows. Each residual length maps to a book, the frequency of a residual length maps to the number of pages in that particular book, and finally the number of dividers maps to the number scholars minus 1. Instead of spreading the workload as evenly as possible we are instead interested in minimizing the cost defined in Equation 1.

Although the partition problem is NP-complete, there exists a dynamic programming solution which runs in pseudo-polynomial time [19]. A variation of this particular solution (Algorithm 5) is used to solve our problem of finding an efficient function $p(l)$. The algorithm works by constructing a matrix where row i, column j contains the minimum average encoding overhead for residuals with length $\leq i$ using j dividers. This matrix is labeled as $cost$ in algorithm 5. The algorithm also keep track of the divider positions corresponding to the minimum cost using the $path$ matrix. Specifically, $path[i][j]$ contains the position of the next lowest divider for optimal partitioning with a divider placed after element i and j more dividers available. Populating the first row and column of the $cost$ and $path$ arrays is trivial, and this forms a basis for calculating the rest of the entries. The optimal number of dividers is found by

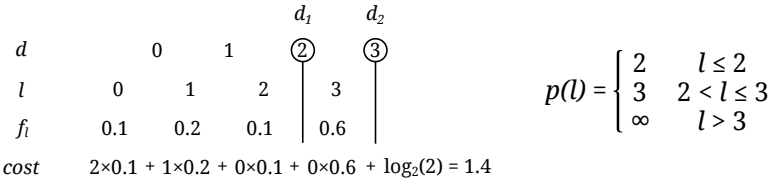


Figure 12. The cost of a particular partitioning is calculated by multiplying each frequency with its distance from the next divider, then adding these products to the base 2 log of the number of dividers.

looking up the average encoding overhead for each number of dividers in the matrix and selecting the best one.

Complexity analysis. Although our partitioning algorithm (Algorithm 5) runs in pseudo-polynomial time in terms of the *maximum residual length*, it still allows Trajic to perform compression in linear time in terms of the number of sample points in each trajectory. This is because the partitioning algorithm only takes the frequency distribution of residual lengths as input and therefore is independent of the number of sample points in the trajectory. This effectively means the partitioning algorithm only imposes a constant overhead on the overall compression algorithm complexity, when complexity is measured in terms of the number sample points in the trajectories.

A more detailed analysis is as follows. The algorithm is dominated by the complexity of the 4 nested for loops. The complexity of this nested for loop only depends on the constants length of the freq array (maximum residual length) and maxDivs which are 64 and 32 in our experiments, respectively. We computed the number of times the inner most part of the loop is executed to be 559705 times. A modern computer can easily do 2 GFLOPS, which works out to be around 0.28 ms to process the entire inner loop. We conducted an experiment, described in Section 4.2.2, which shows that the time taken to execute the partitioning algorithm (place dividers) is only 1% of the total compression time (Figure 14).

3.4.3 Storing the codewords and data using Huffman coding

The previous section focused on finding the optimal dividers to reduce encoding overhead. We can further reduce the encoding overhead by assigning optimal length codewords to each divider based on the frequency distribution of the actual residual lengths. To do this we use Huffman coding [9] to assign the optimal length codewords to dividers as headers.

A Huffman codeword is assigned to each possible divider/output value of $p(l)$, with a divider's frequency (for the purposes of Huffman coding) being the sum of all frequencies for l values between it and the next smallest divider. Once the codebooks for time, latitude and longitude are written, residuals may be stored by writing the appropriate codeword as the header, followed by $p(l)$ bits containing the actual residual data.

Consider a set of four-bit residuals where the frequency distribution of l values is found to be $f_o = 0.1,\ f_1 = 0.4,\ f_2 = 0.1,\ f_3 = 0.1,\ f_4 = 0.3$. Assume the following definition for function $p(l)$:

$$p(l) = \begin{cases} 1 & l \leq 1 \\ 3 & 1 < l \leq 3 \\ 4 & 3 < l \leq 4 \end{cases}$$

Since there are three possible output values of $p(l)$, three Huffman codewords are created. Observe that the most frequently used value of $p(l)$ has the smallest codeword length:

$p(l)$	frequency	codeword	h
1	$f_0 + f_1 = 0.5$	0	1
3	$f_2 + f_3 = 0.2$	10	2
4	$f_3 = 0.3$	11	2

Now, to store a 2-bit residual, a header containing "10" is written out followed by 3 bits containing the 2-bit residual padded with a leading zero.

4 Experiments

4.1 Experimental setup

4.1.1 Implementation and hardware

The Trajic system and other existing compression schemes were implemented and tested via C++11 code compiled with GCC version 4.7.2 using the -O3 optimization flag. Results were produced on an AMD Phenom II X4 965 processor clocked at 3.4 GHz with 4 gigabytes of RAM, running Ubuntu Linux.

4.1.2 Algorithms implemented

Delta compression. A simple delta compression algorithm was implemented with a bit-count leading zero encoding scheme. This is the delta compression method used in TrajStore [5].

TD-SED. TD-SED is a top-down line generalisation algorithm based on the Douglas-Peucker algorithm [6] described in Section 2.1. It utilises the synchronised Euclidean distance (SED) as an error metric to incorporate the time dimension (as described in Section 2.1). TD-SED was used in our experiments to represent the line generalisation approach to trajectory compression. The reason we chose to compare against the Douglas-Peucker algorithm instead of others was due to the thorough set of experiments conducted by Muckell et al. [13]. Muckell et al. [13] compared seven different state-of-the-art trajectory compression algorithms using several real data sets. The results showed that Douglas-Peucker offered the best performance in terms of both low execution time and low synchronized Euclidean distance error.

Data set	Trajectories	Avg sample rate (approx)	Avg samples per trajectory
GeoLife	17,621	1 per 3 secs	1343
Illinois	213	1 per 1 sec	1671

Table 5
Statistics for the data sets used

TD-SED + Delta. TD-SED and delta compression were combined by first applying the line generalisation method to remove points, then storing the remaining points as a sequence of deltas.

Trajic. The Trajic system described in this paper was implemented with the temporally-aware linear predictor and dynamic leading zero encoding.

4.1.3 Data sets

The two main data sets used for the experiments were a set of trajectories recorded by the University of Illinois [17] and data from the Microsoft GeoLife project [12]. Table 5 provides a summary of these data sets.

The GeoLife data contains 17,621 trajectories taken with a variety of GPS devices, and the set includes many modes of transport including walking, cycling and driving. Over 48,000 hours of data taken from April 2007 to October 2011 is contained in the complete data set.

There are 213 trajectories recorded by the University of Illinois used in these experiments. This data corresponds to the movements of two people over a 6 month period.

Since both data sets provide positions in terms of a longitude and latitude, the haversine formula [20] was used to convert distances into kilometers.

4.2 Results

Comparisons for Trajic were made with simple delta compression, the TD-SED line generalisation method and TD-SED + Delta (a combination of TD-SED and delta compression).

4.2.1 Compression ratio

Compression ratio was measured by dividing the total size of compressed data by the amount of space which would be required if each sample point were to be stored in 24 bytes (a 64-bit double for each of time, latitude and longitude). Hence a smaller compression ratio is desirable, as it corresponds to greater savings in storage space. Reasonably strict error bounds of 1 second temporally and 1 metre spatially (approximately 1 metre) were used for the lossy Trajic compression (Table 4).

It is evident that the storage gains from Trajic are significant, with as much as 93% of space being saved with minimal impact on data integrity (less than 1 metre of maximum error). Both the lossless and a lossy version of the Trajic compression system achieve

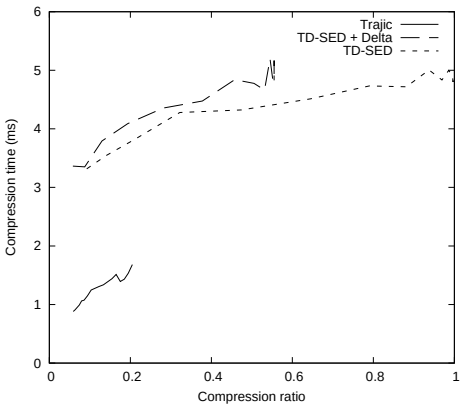


Figure 13. Compression time was plotted against compression ratio to compare the behaviours of trajectory compression algorithms

significantly better compression ratios than line generalisation and delta compression for small to medium error bounds. In particular, with 1 metre of error Trajic produces 5.3 times smaller compressed data than TD-SED + Delta. This shows that Trajic is able to fill the gap where users expect high compression whilst giving away only a small amount of precision, as was expected.

4.2.2 Running time

In order to test the time performance of the compression algorithm presented in this paper, the average time taken to compress/decompress a trajectory was measured (Table 4). Once again, the error bounds for lossy compression were set to 1 second temporally and 1 metre spatially. For both Trajic and TD-SED, it was found that allowing for greater maximum error resulted in a faster decompression time. This was expected as when either system stores less data in compressed form, it takes less time to restore the trajectory.

Compression ratio versus compression time. We used 100 points from the GeoLife data set to perform a micro benchmark comparing compression time with ratio for the Trajic, TD-SED and TD-SED + Delta algorithms. Figure 13 was created by varying the error bounds from 0 to 100 metres and calculating the average compression time and ratio for each bound. The graph shows that better compression speed is achieved at better compression ratios (larger error bounds) for each of the algorithms tested. This is due to the nature of these algorithms: lower compression ratios are achieved by dropping parts of the data, which means that there is less data to process. Observe that Trajic is able to compress trajectories much quicker than the TD-SED variants for all ratios in the graph.

Trajic compression time breakdown. Additional timers were added to the compression code as a way of breaking down compression time for various stages of the process. We selected the largest trajectory in the GeoLife data set for this experiment to better distinguish linear time procedures from large constant

Algorithm	Compression ratio		Compression time (ms)		Decompression time (ms)	
	GeoLife	Illinois	GeoLife	Illinois	GeoLife	Illinois
Delta	0.55	0.39	0.7	0.3	0.8	0.4
Trajic (lossless)	0.37	0.18	5.2	2.8	0.6	0.2
TD-SED (1m error)	0.66	0.90	4.5	2.5	0.6	0.4
TD-SED + Delta (1m error)	0.37	0.36	4.3	2.5	0.6	0.3
Trajic (1m error)	0.07	0.08	1.9	1.2	0.3	0.2
TD-SED (31m error)	0.12	0.75	3.4	2.5	0.1	0.4
TD-SED + Delta (31m error)	0.07	0.31	3.0	2.6	0.1	0.3

Table 4
The compression ratios and average compression/decompression times per trajectory

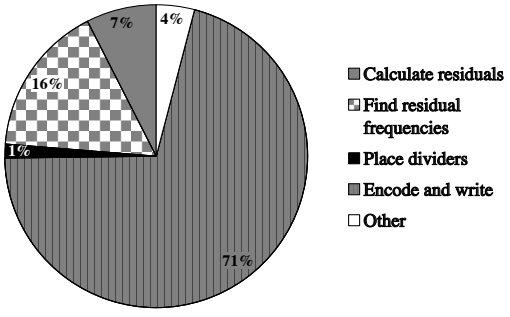


Figure 14. The compression time breakdown was measured for the largest trajectory in the GeoLife data set

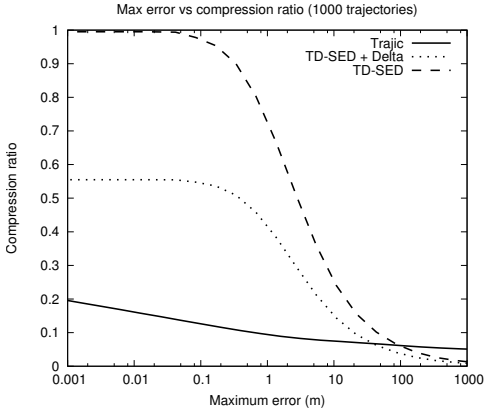


Figure 15. Compression ratio was plotted against spatial error bound to compare Trajic with TD-SED (line generalisation) and TD-SED enhanced by delta compression over the GeoLife data set

time procedures. Figure 14 shows that compression time is dominated by linear time operations such as calculating residuals (this includes predicting), finding residual frequencies, encoding, and writing to disk. A key observation is that the pseudo-polynomial time algorithm for placing dividers (partitioning algorithm) occupies only 1% of the total compression time. This result supports our earlier observation that although the algorithm has a high time complexity with respect to its constant inputs, it is acceptable due to its independence from trajectory size.

4.2.3 Overall performance

Considering compression ratio and decompression time simultaneously it becomes obvious that Trajic compression is fast as well as effective, and should not have a detrimental impact on the speed of a larger database system. The line generalisation system can only match Trajic's effectiveness if large error bounds are permitted (31 meters as opposed to 1 metre).

Furthermore, multiple scatter plots were constructed to compare compression/decompression time with trajectory length. These confirmed the time complexity analysis, revealing a linear time complexity for Trajic.

4.2.4 Error bounds

The relationship between specified error bounds and compression ratio was tested by keeping the temporal error constant whilst the varying the spatial error. The results across 1000 trajectories are shown in Figure 15. Note that the x-axis uses a log to base 10 scale. This graph clearly illustrates the ability of Trajic to fill the gap left by existing algorithms which is achieving a good (low) compression ratio while sacrificing small error margins. In particular, at the extremely small maximum error of 0.001 meters, Trajic produces 5 times smaller compressed data than TD-SED and 2.8 times smaller compressed data compared to TD-SED + Delta.

Figure 15 shows an interesting property: $compression\ ratio \propto \log(error\ bound)$ for small error bounds in the Trajic system. This is because the number of discarded bits (referred to as D earlier) is proportional to the logarithm of the error bound. While the number of discarded bits is less than most of the residual lengths (l values), discarding an extra bit effectively means removing one bit from the storage space required for each of the majority of residuals. Therefore, compression ratio is proportional to the logarithm of the user-specified error bound until the number of discarded bits exceeds the bulk of residual lengths, as at this stage most residuals are zero and discarding further bits will not reduce their storage requirements.

In contrast, the line generalisation algorithm de-

grades very quickly as maximum error is reduced, with Trajic becoming the superior system for errors less than approximately 25 meters. At less than 0.2 meters it becomes impossible for line generalisation to improve the compression ratio any further.

5 Conclusion

Trajic is an effective, efficient and flexible compression system for the reduction of storage space required for trajectory data. It takes advantage of the inertia of objects to predict where future points in a trajectory will lie, and then stores the residuals between the actual and predicted points using a novel dynamic leading zero encoding scheme. Both the compressor and decompressor run in linear time, making the system feasible for database applications. In addition, the compressor works for any reasonable user-specified error bound, achieving good ratios for both lossless and varying levels of lossy compression. Trajic fills a gap existing amongst current systems, as it is able to achieve better compression ratios than delta and line generalisation for low maximum error bounds.

The compressor proposed here is independent in that it can compress and decompress trajectories without any additional requirements, such as knowledge of a road map or historical data. Consequently it is not difficult to integrate the Trajic system into a full trajectory database ensemble. For example, the delta compression step described in TrajStore [5] can be replaced with the system proposed here, whilst still maintaining the functionality of the rest of the system as described in the paper; including the clustering algorithm for greater compression across multiple similar trajectories.

An interesting direction for future work is to measure the combined effects of Trajic and an extra layer of compression via the clustering algorithm of TrajStore. A challenging aspect of this is to do the combined compression while limiting the maximum error to be within a user defined threshold.

The source code for a working implementation of Trajic may be found at https://github.com/dismaldenizen/trajic.

References

[1] R. H. Bartels, J. C. Beatty, and B. A. Barsky. *An introduction to splines for use in computer graphics & geometric modeling*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[2] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *VLDB*, pages 853–864, 2005.

[3] H. Cao and O. Wolfson. Nonmaterialized motion information in transport networks. In *Proceedings of the 10th International Conference on Database Theory (ICDT)*, pages 173–188, Berlin, Heidelberg, 2005. Springer-Verlag.

[4] M. Chen, M. Xu, and P. Fränti. Compression of gps trajectories. In *Data Compression Conference (DCC)*, pages 62–71, 2012.

[5] P. Cudré-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *Proceedings of the 26th International Conference on Data Engineering, (ICDE)*, pages 109–120, 2010.

[6] D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a line or its caricature. *The Canadian Cartographer*, 10(2):112 –122, 1973.

[7] M. K. El Mahrsi, C. Potier, G. Hébrail, and F. Rossi. Spatiotemporal sampling for trajectory streams. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 1627–1628, New York, NY, USA, 2010. ACM.

[8] J. Hershberger and J. Snoeyink. Speeding up the Douglas-Peucker line-simplification algorithm. In *Proc. 5th Intl. Symp. on Spatial Data Handling*, pages 134–143, 1992.

[9] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098 –1101, sept. 1952.

[10] R. Lange, F. Dürr, and K. Rothermel. Efficient real-time trajectory tracking. *VLDB Journal*, 20(5):671–694, 2011.

[11] N. Meratnia and R. A. de By. Spatiotemporal compression techniques for moving point objects. In *Advances in Database Technology (EDBT)*, pages 765–782, 2004.

[12] Microsoft. GeoLife GPS Trajectories. http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/, 2011.

[13] J. Muckell, J.-H. Hwang, C. T. Lawson, and S. S. Ravi. Algorithms for compressing gps trajectory data: an empirical evaluation. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 402–405, New York, NY, USA, 2010. ACM.

[14] J. Muckell, J.-H. Hwang, V. Patil, C. T. Lawson, F. Ping, and S. S. Ravi. SQUISH: An Online Approach for GPS Trajectory Compression. In *Proceedings of the 2Nd International Conference on Computing for Geospatial Research &Amp; Applications*, COM.Geo '11, pages 13:1–13:8, New York, NY, USA, 2011. ACM.

[15] N. Meratnia and R. de By. A New Perspective on Trajectory Compression Techniques, 2003.

[16] T. Neumann and G. Weikum. Rdf-3x: A risc-style engine for rdf. *Proc. VLDB Endow.*, 1(1):647–659, Aug. 2008.

[17] U. of Illinois at Chicago. Real Trajectory Data. http://www.cs.uic.edu/~boxu/mp2p/gps_data.html, 2006.

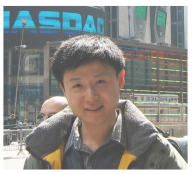
[18] M. Potamias, K. Patroumpas, and T. Sellis. Sampling trajectory streams with spatiotemporal criteria. In *Proceedings of the 18th International Conference on Scientific and Statistical Database Management*, SSDBM '06, pages 275–284, Washington, DC, USA, 2006. IEEE Computer Society.

[19] S. Skiena. *The Algorithm Design Manual*. Springer, 2nd edition, 2008.

[20] W. M. Smart. *Textbook on Spherical Astronomy*. Cambridge University Press, sixth edition, 1977.

[21] Z. Xie, H. Wang, and L. Wu. The improved douglas-peucker algorithm based on the contour character. In *Geoinformatics, 2011 19th International Conference on*, pages 1 –5, june 2011.

Aiden Nibali is an undergraduate scholarship student in Computer Science and Electronic Engineering at La Trobe University Australia. His interests include big data analytics, data compression, audio processing and programming language design.



Zhen He is a senior lecturer in the department of computer science in La Trobe University Australia. He obtained his undergraduate and PhD degrees in computer science from the Australian National University. His research interests include big data analytics, moving object databases, relational database optimization and parallel databases.