A Framework for Trajectory Segmentation by Stable Criteria

Sander P. A. Alewijnse
TU Eindhoven
s.p.a.alewijnse@student.tue.nl

Kevin Buchin
TU Eindhoven
k.a.buchin@tue.nl

Maike Buchin
Ruhr-University Bochum
maike.buchin@rub.de

Andrea Kölzsch
Max Planck Institute for Ornithology
akoelzsch@orn.mpg.de

Helmut Kruckenberg
European White-Fronted Goose Research Programme

Michel A. Westenberg
TU Eindhoven
m.a.westenberg@tue.nl

ABSTRACT

We present an algorithmic framework for criteria-based segmentation of trajectories that can efficiently process a large class of criteria. Criteria-based segmentation is the problem of subdividing a trajectory into a small number of parts such that each part satisfies a global criterion. Our framework can handle criteria that are stable, in the sense that these do not change their validity along the trajectory very often. This includes both increasing and decreasing monotone criteria. Our framework takes $O(n \log n)$ time for preprocessing and computation, where n is the number of data points. It surpasses the two previous algorithmic frameworks on criteria-based segmentation, which could only handle decreasing monotone criteria, or had a quadratic running time, respectively. Furthermore, we develop an efficient data structure for interactive parameter selection, and provide mechanisms to improve the exact position of break points in the segmentation. We demonstrate and evaluate our framework by performing case studies on real-world data sets.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations*

General Terms

Algorithms, Theory

Keywords

Trajectory, Computational Geometry, Segmentation

SIGSPATIAL'14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA

ACM 978-1-4503-3131-9/14/11 ...$15.00
http://dx.doi.org/10.1145/2666310.2666415.

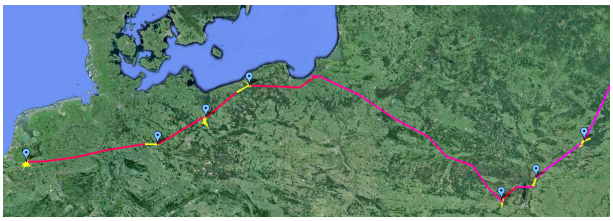


Figure 1: Segmentation of geese data. Red/pink indicates migration flight, yellow stopovers. Blue markers indicate end of a stopover.

1. INTRODUCTION

Movement tracking devices are currently widely available for all kinds of applications. In ecology they are used for tracking animals, and analysing their movement behavior [4]. For instance, tracking technology has been used to detect stopover places of migrating geese [3]. A trajectory is recorded as a series of timestamped locations, measured at regular or irregular intervals. Currently, the amount of recorded data is rapidly increasing, and methods are needed for processing and analyzing it.

We study the following important analysis task: finding a *segmentation* of a trajectory. A segmentation is a partitioning of a trajectory into pieces, which are called *segments*. There are several ways to segment a trajectory. Previous work focussed on finding a semantic annotation of the trajectory [9], profile-based segmentation [7] and *criteria*-based segmentation [2]. In this paper we will focus on latter type of segmentation. In the criteria-based setting, segments are chosen such that the movement inside each segment is homogenous in the sense that it fulfills a given criterion (e.g., on speed). The goal is to find a minimal number of segments under this condition, which is equivalent to a maximal average length of segments. We assume that segments start and end at data points of the trajectories (and not points interpolated inbetween[1]). See Fig. 1 for an example. Segmentation can be combined with classification of the segments. For this, the segmentation is done based on multiple criteria, each of them defining a class, e.g., corresponding to a movement state [3].

[1]We consider what is called discrete segmentation in contrast to continuous segmentation in some previous work.

Buchin *et al.* developed a framework that computes a segmentation given a *decreasing (monotone)*[2] criterion [2]: criteria with the property that if they hold on a certain segment, they also hold on every subsegment of that segment. Many simple criteria concerning homogeneity of location, speed and heading are in this class. Also disjunctions and conjunctions of those simple criteria belong to this class of criteria. In this framework, a segmentation can generally be computed in $O(n \log n)$ time, where n is the number of data points. This also holds for the *continuous segmentation problem*, in which the trajectories are interpolated linearly between data points and segments may also start and end between data points. The framework was used for segmenting animal tracks by movement states of migrating geese [3].

There are many meaningful criteria that are not decreasing, such as minimum duration, but these criteria are harder to handle. Aronov *et al.* [1] presented a simple general algorithm using the *start-stop matrix* that runs in $\Theta(n^2)$ time. Due to the high running time, this algorithm is not suitable for trajectories with a large number of data points.

In this paper we present a criterion-based segmentation framework that runs in $O(n \log n)$ time, with n the number of data points, which allows for:

- Handling a broad class of criteria: the *stable* criteria, which are criteria that do not change their validity along the trajectory very often. This includes decreasing and increasing criteria and combinations of them.
- Segmentation by movement states, with rules governing state transitions and additional optimization goals to determine the exact point of transitions.
- Interactive parameter selection, guided by the stability of the resulting segmentations.

Note that the combination of a decreasing and increasing criteria, e.g., minimum duration and maximum speed, are neither increasing nor decreasing, but they are stable and can therefore be handled in our framework. We demonstrate our framework in a case study on movement data of migrating white-fronted geese[3].

2. FRAMEWORK

In this section we describe our conceptual framework for criteria-based trajectory segmentation, deferring algorithmic details to later sections. Throughout the paper we assume that a trajectory τ is given by a sequence of n triples (x_i, y_i, t_i), where (x_i, y_i) is the location of a moving entity (e.g., an animal) at time t_i. We denote the timestamped locations of τ also by $\tau(i) = (x_i, y_i)$. We treat a trajectory as the sequence of timestamped locations, hence a subtrajectory can only start and end at recorded time stamps. A subtrajectory of τ starting at time t_i and ending at (and including) time t_j is denoted by $\tau[i, j]$. A *segmentation* of a trajectory is a partition of a trajectory τ in subtrajectories (called *segments*) such that these segments have disjoint interiors and cover the whole trajectory τ. We use k to denote the number of segments and s_i to denote the end point of the ith segment; a segmentation is then given by $\tau[s_0, s_1]$, $\tau[s_1, s_2]$, $\ldots$, $\tau[s_{k-1}, s_k]$, with $0 = s_0 < s_1 < \cdots < s_k = n$.

Our segmentation approach is based on *criteria*. Formally, a criterion C is a function that maps a subtrajectory (candidate segment) to true or false. We denote the value of C for candidate segment $\tau[i, j]$ by $C(i, j)$. Intuitively, a criterion conditions the movement on the segment, e.g., by requiring a maximum speed and/or turning angle. A candidate segment that satisfies C is called a *valid segment*, and a segmentation consisting only of valid segments is called a *valid segmentation*. The goal of *criterion-based* segmentation is to find a valid segmentation of τ with a minimal number of segments, given a trajectory τ and a criterion C. Such a segmentation is called *optimal*.

Not all criteria can be handled equally. Previous work focused on the class of *decreasing* criteria [2]. A decreasing criterion has the property that if it holds on a certain candidate segment, it also holds on every subsegment of that segment. There are numerous examples of this kind. For instance, criteria that bound the range of a trajectory attribute (such as speed and heading) are decreasing. For these criteria, the greedy segmentation strategy will yield an optimal result [2]. This strategy starts at the beginning of the trajectory, and makes the first segment as long as possible according to the criterion. The remainder of the trajectory is then segmented in the same way, making each segment as long as possible.

Another class are the *increasing (monotone)* criteria. An increasing criterion has the property that if it holds on a certain segment, it also holds on all segments that contain that segment. Important examples are the minimum length and duration criterion that place lower bounds on length and duration of a segment, respectively. Segmentation based on increasing criteria only is not meaningful: either the whole trajectory is "segmented" in one segment or the trajectory cannot be segmented. However, Boolean combinations of increasing and decreasing criteria can yield meaningful results, e.g., minimum duration and staying in an area. For a combination of increasing and decreasing criteria, the greedy strategy does *not* always yield an optimal result, however.

Consider the trajectory $\tau[0, 5]$ in Fig. 2. It is a regularly sampled track ($t_i = i$ for all $i = 0, \ldots, 5$) of an object that is moving with constant acceleration ($\tau(i) - \tau(i-1) = i$ for all $i = 1, \ldots, 5$) starting at $\tau(0) = (0, 0)$. Assume that the criterion is a conjunction of a duration criterion D (increasing) and a bounded speed range criterion S (decreasing). Criterion D requires a minimum segment duration of 3 and criterion S allows for a maximum speed range of 4, where the speed is estimated by forward-differentiation.

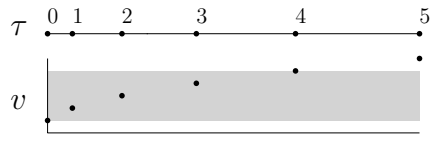


Figure 2: Trajectory τ and speed v along τ. The gray box has height 4 and indicates that $\tau[0, 4]$ satisfies S.

In this case, the greedy strategy picks $\tau[0, 4]$ as first segment, since this is the longest segment starting at t_0 that satisfies $D \wedge S$. There is no segment starting at t_4 that satisfies D, so the greedy strategy will fail to find a segmentation. However, there is a segmentation $\tau[0, 2], \tau[2, 5]$ (optimal) which satisfies $D \wedge S$.

Our framework can handle Boolean combinations of increasing and decreasing criterion. In fact, it can handle all *stable* criteria. These criteria do not change their validity

[2]These criteria are called monotone in [2].

[3]see also the video: `http://youtu.be/5JaRbYgykNg`

along the trajectory very often. More formally: Consider all candidate segments ending at i ordered by increasing length. Let $v(i)$ denote the number of times the validity changes along this order. A criterion is stable if and only if $\sum_{i=0}^{n} v(i) = O(n)$.

Our framework allows for classification of the segments as well, when the classes can be described by criteria. The algorithm then segments by the disjunction of the criteria for all possible classes (e.g., movement states). As a result, the algorithm computes a labeled classification of subtrajectories of the given trajectory. The concept of movement states allows for a broad range of additional rules on the segmentation. First of all, rules can be defined for breaking ties. The original segmentation problem only minimizes the number of segments, and has no preferences among segmentations with equal segment count. However, in practice there are segmentations that are better than others, despite having equal segment count, for example depending on the exact location of segment boundaries. Those rules can be formalized using movement states. Secondly, we can add restrictions on the state transitions by enforcing rules of the form "in every segmentation movement state A can only be followed by movement state B, C or D". The framework also allows for penalization of certain state transitions.

Outliers (e.g., resulting from GPS noise) can also be handled efficiently by our framework. Also, there are outliers that can be ignored, for instance very short stops during a long flight. There are multiple levels at which we can handle outliers. Previous work showed that a constant number of outliers per segment can be ignored by a criterion [3], without changing the monotonicity of the criterion. However, it is more desirable to allow for a certain percentage of outliers in each segment [1] (resulting in a non-monotone criterion). Our framework approximates such a criterion of the form "C except for a fraction f of the points" by:

$$(C \text{ except for 1 point } \wedge \text{ number of points } \geq 1/f) \vee$$
$$(C \text{ except for 2 points } \wedge \text{ number of points } \geq 2/f) \vee \ldots$$

Outliers can also be handled at a higher level in the segmentation framework. Consider the problem of stopping or starting at outliers. To limit the number of consecutive outliers we could for instance put segments starting or ending at outliers in separate states respectively and forbid the transition "ending at outlier" to "starting at outlier".

The framework segments trajectories in $O(n \log n)$ time. First, each criterion is transformed into a compressed start-stop matrix (see Sections 3 and 4). The segmentation algorithm is discussed in Section 5. Moreover, we can put extra rules based on the movement states of the segments, as is described in Section 6. Choosing the segmentation criterion parameters is crucial but also difficult in practice, because it requires domain knowledge. To fine-tune the parameters, the segmentation needs to be computed multiple times, each time using a slightly different setting of parameters. To aid this process, we use an interactive process which is supported by the *stability diagram* introduced in Section 7.

3. COMPRESSED START-STOP MATRIX

Aronov *et al.* [1] presented several algorithms for segmentation. Mostly they focussed on *continuous* segmentation, i.e., segmenting also at interpolated points, for which they showed NP-hardness. For (discrete) segmentation, they gave a simple algorithm which works for any computable criterion. The first step of this method is to compute the *start-stop matrix*. The storage of this matrix takes $\Theta(n^2)$ space. The second step is the computation of the actual segmentation from this matrix using a simple dynamic programming algorithm in $\Theta(n^2)$ time.

Our approach consists of two steps with similar goals. First we construct a *compressed start-stop matrix*. This data structure can efficiently test for any candidate segment $\tau[i, j]$ whether it satisfies the criterion. This data structure has size $O(n)$ and can be computed in $O(n \log n)$ time for a broad class of criteria. In the second step the segmentation is computed from this compressed start-stop matrix in $O(n \log n)$ time. Hence, our framework takes $O(n \log n)$ time in total instead of $\Theta(n^2)$, making the framework practical to use even on long trajectories. In this section we introduce the compressed start-stop matrix. In Section 4 we discuss how to compute it for several criteria. The algorithm that computes the optimal segmentation, given a compressed start-stop matrix as input, is given in Section 5.

3.1 Start-stop matrix

A *start-stop matrix* stores the relation between a trajectory τ and a criterion C. Consider the parameter space of the set of subtrajectories of τ. For any candidate segment $\tau[i, j]$, the start parameter i is associated with the column index and the stop parameter j with the row index of the matrix. So a matrix entry (i, j) in the upper left triangle $(i \leq j)$ represents a candidate segment. Each of those is assigned a value $C(i, j)$, which is true if the criterion C is satisfied by the candidate segment and false otherwise. A candidate segment is called part of the *free space* if it satisfies C and it is part of the *forbidden space* otherwise.

A segmentation of τ into a sequence of segments $\tau[s_0, s_1]$, $\ldots$, $\tau[s_{k-1}, s_k]$ corresponds to a staircase in the start-stop matrix. Consecutive segments $\tau[s_i, s_{i+1}], \tau[s_{i+1}, s_{i+2}]$ share a trajectory point. This means that the row index of matrix entry corresponding to $\tau[s_{i+1}, s_{i+2}]$ is equal to the column index of the matrix entry corresponding to $\tau[s_i, s_{i+1}]$. Hence the matrix entries corresponding to the segments $\tau[s_i, s_{i+1}]$ together with the entries (s_i, s_i) on the main diagonal form a staircase. Furthermore, a segmentation is valid if and only if all non-diagonal entries of the staircase lie in the free space. See Fig. 3 for an example of a valid segmentation.

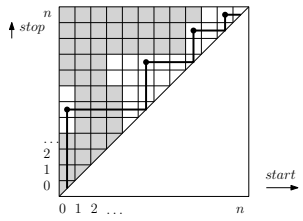


Figure 3: A start-stop matrix and an optimal segmentation into four segments. The free-space is white, the forbidden space is gray. The segmentation is valid, because the four segments (indicated by dots) lie in the free-space.

3.2 Compressing the start-stop matrix

For many criteria the start-stop matrix can be compressed significantly by applying run-length encoding to each row.

Run-length encoding is a simple form of data compression in which *runs* are stored in a compressed form [8]. A run is a sequence of consecutive values that are equal. In our case, we have runs of *true* values and runs of *false* values, which we call *blocks* and *gaps*, respectively. Runs are stored as pair of *value* and *count*. We call this row-wise run-length encoded start-stop matrix the *compressed start-stop matrix*.

Consider the start-stop matrix for a decreasing criterion. The property "if C is satisfied by a certain segment, it is also satisfied by every subsegment" implies that all matrix entries to the right of a true value must be true. Recall that a matrix entry (i', j) right of a matrix entry (i, j) corresponds to $\tau[i', j]$ being a subsegment of $\tau[i, j]$. A row of a decreasing start-stop matrix hence consists of at most two runs: an optional gap followed by a block. An example of a start-stop matrix for a decreasing criterion is shown in Fig. 4(a).

In a similar way the start-stop matrix for an increasing criterion can be compressed. The increasing property implies that all matrix entries to the left of a true value must be true. A row of such a start-stop matrix hence consists of an optional block followed by a gap. An example is shown in Fig. 4(b).

Using the compressed instead of the uncompressed start-stop matrix for decreasing and increasing criteria reduces the storage to $O(n)$. In Section 4 we show that this compressed start-stop matrix can be computed in $O(n \log n)$ for many decreasing and increasing criteria.

Our algorithm is not limited to decreasing and increasing criteria. In fact, it can handle any criterion that has a compressed start-stop matrix with $O(n)$ blocks, which is the case for all stable criteria. Therefore, our framework can compute the optimal segmentation in $O(n \log n)$ time for any stable criterion given its compressed start-stop matrix. Note that the same holds for start-stop matrices that can be compressed to linear size using run-length encoding on the columns; simply reverse the trajectory.

3.3 Combining and transforming compressed start-stop matrices

Basic criteria can be combined to get compound criteria, which can be more effective at segmenting trajectories than basic criteria, as has been demonstrated in [3]. There are two ways to combine criteria: the *conjunction* and *disjunction*.

Given two stable criteria C and C' and their compressed start-stop matrices, the compressed start-stop matrix of $C_1 \wedge C_2$ can be computed efficiently. The key observation is that $C_1 \wedge C_2$ is satisfied by a candidate segment $\tau[i, j]$ if and only if C_1 and C_2 are satisfied. Hence the free space of the start-stop matrix of $C_1 \wedge C_2$ equals the *intersection* of the free space of the start-stop matrices of C_1 and C_2. The run-length encoded form of this intersection can computed per row taking in total $O(n)$ time. Similarly, the compressed start-stop matrix of $C_1 \vee C_2$ equals the *union* of the free space of the start-stop matrices of C_1 and C_2. This union can also be computed in $O(n)$ time. There are several properties concerning combinations of increasing and decreasing criteria.

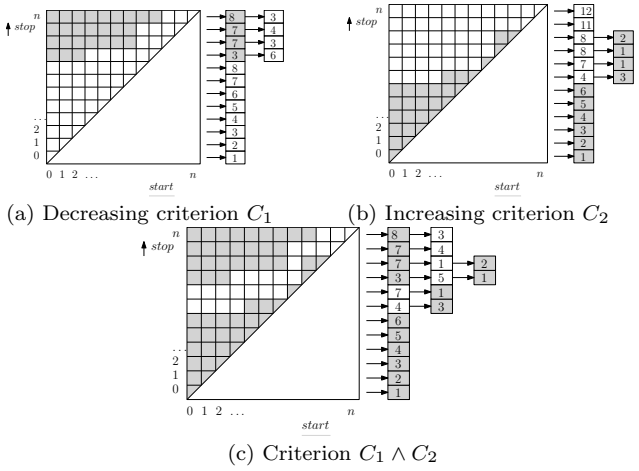
THEOREM 1. *[2, Theorem 15] A combination of conjunctions and disjunctions of decreasing criteria is a decreasing criterion.*

THEOREM 2. *A combination of conjunctions and disjunctions of increasing criteria is an increasing criterion.*

PROOF. Let C and C' be increasing criteria. We will show that the conjunction $C \wedge C'$ and the disjunction $C \vee C'$ are increasing. First we consider the conjunction. Let τ' be a subtrajectory of τ. Assume that $C \wedge C'$ is satisfied by τ'. Let τ'' be a supertrajectory of τ'. Criterion C is satisfied by τ'', because it holds for τ'. The same holds for C'. This implies that $C \wedge C'$ is satisfied by τ''. Hence $C \wedge C'$ is also increasing.

Now we consider the disjunction. Let τ' be a subtrajectory of τ. Assume that $C \vee C'$ is satisfied by τ'. Let τ'' be a supertrajectory of τ'. Without loss of generality assume that criterion C is satisfied by τ'. Then, C is also satisfied by τ''. Hence $C \vee C'$ is satisfied by τ''. This proves that $C \vee C'$ is increasing. □

Note that the conjunction (or disjunction) of a decreasing combined with an increasing criterion is not decreasing or increasing. An example is given in Fig. 4.



(a) Decreasing criterion C_1 (b) Increasing criterion C_2

(c) Criterion $C_1 \wedge C_2$

Figure 4: Two start-stop matrices and their conjunction.

Criteria can also be transformed by applying *negation*. Given a stable criterion and its compressed start-stop matrix the compressed start-stop matrix of the negation of the criterion can be computed efficiently: Change the forbidden space to free space and vice versa. This can be done in $O(n)$ time. Applying negation to a decreasing or increasing criterion "reverses" these properties, in the following sense.

THEOREM 3. *The negation of a decreasing criterion is an increasing criterion and vice versa.*

PROOF. Let τ be a trajectory. Let C be a decreasing criterion. Assume for the purpose of contradiction that $\neg C$ is not increasing. This means that there is a candidate segment $\tau[i, j]$ that does not satisfy C, for which a supersegment $\tau[i', j']$ exists with $i' \leq i$ and $j' \geq j$ that satisfies C. However, validity of $\tau[i', j']$ implies validity of $\tau[i, j]$ by the decreasing monotonicity of C. This is a contradiction. The proof of the other direction is analogous. □

4. COMPUTING THE COMPRESSED START-STOP MATRIX

The compressed start-stop matrix for a decreasing criterion can be computed using the algorithm *ComputeLongest-*

Valid. Given a trajectory data set τ and a decreasing criterion C the algorithm computes for every trajectory index j the smallest index i for which $\tau[i,j]$ satisfies the criterion. This index is stored in LV_j. Given LV_j it is straightforward to compute the actual compressed start-stop matrix in $O(n)$ time. For *increasing* criteria the compressed start-stop matrix can be computed using the same algorithm. Simply replace $\neg C$ by C in *ComputeLongestValid* and negate the resulting compressed start-stop matrix. The correctness of this method is a direct consequence of Theorem 3.

Algorithm *ComputeLongestValid*(C, τ)
1. $i \leftarrow n$;
2. Initialize empty $\mathcal{D}_C$;
3. **for** $j \leftarrow n$ **to** 0
4. **do while** $i \geq 0 \wedge \mathcal{D}_C.SegmentIsValid$
5. **do** $i \leftarrow i - 1$;
6. $\mathcal{D}_C.Extend(i)$;
7. $LV_j \leftarrow i + 1$
8. $j \leftarrow j - 1$;
9. $\mathcal{D}_C.Shorten(j)$;

The algorithm *ComputeLongestValid* computes LV_j by moving two pointers i and j backwards over all n points of the trajectory τ. Both pointers start at the last point of the trajectory. Pointer i is moved backwards until the segment $\tau[i,j]$ is not valid. At that moment we can conclude that LV_j is equal to $i+1$. Then the pointer j is moved one step and the next LV_j is determined in a similar fashion. Note that it is not necessary to reset pointer i to j, because C is decreasing.

Testing whether $\tau[i,j]$ satisfies a criterion is not a straightforward task. Therefore the data structure $\mathcal{D}_C$ is included in the algorithm to keep track of the validity of candidate segment $\tau[i,j]$. The actual form of this data structure depends on the kind of decreasing criterion C that is considered. It allows for three operations. First of all it can be queried for the validity of $\tau[i,j]$ using the *SegmentIsValid* function. Secondly the segment $\tau[i,j]$ can be extended by one point at the start, when i is decreased by 1. Thirdly the segment $\tau[i,j]$ can be shortened by one point at the end, when j is decreased by 1.

The algorithm *ComputeLongestValid* consists of at most $O(n)$ steps in which the interval $\tau[i,j]$ is extended or shortened. The running time of the algorithm depends on the precise data structure that is used for $\mathcal{D}_C$. Let $M_\ell(n)$, $M_r(n)$ and $T(n)$ denote the running times of respectively the *Extend*, *Shorten* and *SegmentIsValid* operations. The running time of *ComputeLongestValid* can then be stated as $O(n(M_\ell(n) + M_r(n) + T(n)))$.

The following paragraphs list some basic decreasing criteria and the data structure $\mathcal{D}_C$ that is used for computing the corresponding compressed start-stop matrices.

Range criterion on attribute. There is a large class of criteria of the form "for all points in the segment attribute a should be within a range of size α". Alternatively such a criterion can be seen as an upper bound of α on the difference between the maximal and the minimal value of attribute a over all points in the segment.

The data structure $\mathcal{D}_C$ keeps track of the minimal and maximal element. For monotone attributes, such as time and traveled distance, this is easy, because they can only increase along the trajectory. The minimal and maximal element of the segment always correspond to respectively the first and last element. Keeping track of those elements when extending or shortening the candidate segment can be done in constant time. Testing the validity is also done in constant time by comparing the difference between minimal and maximal element with α.

For non-monotone attributes, such as speed, data structure $\mathcal{D}_C$ is slightly more complicated. An ordered multiset data structure such as a balanced binary search tree can be used to keep track of all attribute values of the current candidate segment. Extending and shortening the candidate segment correspond to respectively inserting and deleting an attribute value. Testing whether $\tau[i,j]$ is valid consists of a query for the maximal and minimal element in the multiset and comparing their difference with α. Using a balanced binary search tree all three operations take $O(\log n)$ time.

Note that the range criterion on speed should ignore the speed value of one of the endpoints. Otherwise a large change in velocity would make segmentation impossible. The algorithm *ComputeLongestValid* requires only a small change to make this possible. If the starting point of a segment is excluded we simply need to decrease all LV_j values by one. If the stopping point of a segment is excluded, all LV_j need to be shifted one step, that is: $LV_j \leftarrow LV_{j-1}$.

In practice, it can be very useful to allow for a constant number c of outliers that do not need to lie within the range of size α. This does not affect decreasing monotonicity. A similar data structure can be used as before. We will consider the $c+1$ canonical (ending at attribute values that are present) ranges that cover all values except for c outliers. See for example Fig. 5, where two outliers are allowed. Finding those ranges can be done quite efficiently in $O(c \log n)$ time. If the smallest range is less than α the candidate segment is valid.

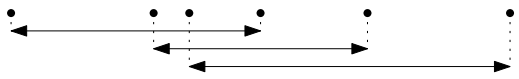


Figure 5: An ordered set of values and its corresponding $c+1$ canonical ranges that cover all but c elements.

Lower bound / Upper bound on attribute. Another class of criteria is of the form "for all points in the segment attribute a should be $\geq \gamma$ (or $\leq \gamma$)". These criteria are especially useful in compound criteria. The same data structure can be used as for the range criterion, but instead of comparing the difference between the extreme points, only the maximal or minimal element is compared with γ. The running times remain the same.

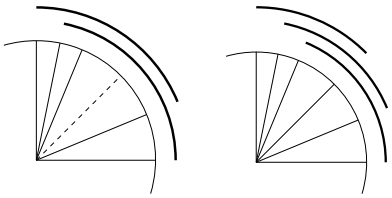
Treating outliers again can be done more efficiently. Testing whether all values except for c outliers are larger than γ is equivalent to comparing the value of the cth order statistic with γ. To allow for fast order statistics queries the binary tree can be augmented into an order statistics tree. This tree can retrieve the cth order statistic in $O(\log n)$ time. Note that the running time is *independent* of c. The running times of the insert and delete operations remain unchanged.

Angular range criterion on attribute. The range criterion for *angular* attributes, such as heading and turning angle, is similar to the range criterion for ordinary attributes. The only difference is that the value range is wrapped around. For instance, the heading values $\pi/6$ and

$11\pi/6$ can be covered by a range of size $\pi/3$. If the upper bound on the angular range α is less than π, the approach is similar to the ordinary range criterion. We maintain a minimal and maximal element that differ less than π (modulo 2π) and span all values in between.

However, if the upper bound α is larger than π the circular nature of the angular domain prevents us from maintaining a meaningful maximal and minimal element. In that case we will keep track of the largest gap between consecutive angular values instead. The smallest range that can cover all the attribute values has size $360^\circ - g$, where $g \in [0, 2\pi)$ is the angle of this largest gap. To keep track of this largest gap we use two ordered multiset structures that both store the set of gaps. The sets are ordered by respectively size and angular order. Testing validity corresponds to a query for the largest gap g and comparing $360^\circ - g$ with α. Extending and shortening correspond to respectively splitting and merging a gap, both taking $O(\log n)$ time using the set structures.

The data structure can be extended to allow for a constant number c of outliers. Instead of storing gap-intervals that span exactly two values, we store intervals that span exactly $2 + c$ values. Testing validity remains unchanged. However, the extend and shorten operations have to change. Inserting a value will change c intervals and create one new interval. Deleting a value will change c intervals and delete one. Both operations take $O(c \log n)$ time. An example is given in Fig. 6 for two outliers.



(a) Before insertion (b) After insertion

Figure 6: Inserting of the dashed value in the angular range data structure allowing two outliers. Only the changed intervals are shown.

Disk criterion. A disk criterion has the form "all points in the segment can be covered by a circle with fixed radius r". We use a data structure $\mathcal{D}_C$ to keep track of the *smallest enclosing circle* of the points on the segment. The query corresponds to comparing the radius of the smallest enclosing circle to r.

It is not straightforward to maintain the smallest enclosing circle efficiently in a dynamic setting. The asymptotically fastest technique lifts the problem to convex programming over a half-space intersection [5]. Insertions, deletions and smallest enclosing circle query are guaranteed to take only polylogarithmic time. Using this data structure the algorithm *ComputeLongestValid* has a near-linear running time.

However, the complex data structure described in [5] has high constant factors in the running time. It is much faster to maintain an *approximate* smallest enclosing circle using our method described below. This can cause the algorithm to compute a segmentation that is not exactly optimal with respect to the exact disk criterion, but the differences (if any are present) are mostly insignificant. Note that the approximation ratio can be chosen arbitrary close to one.

The idea of the approximation scheme is to maintain a constant size approximate convex hull and compute its smallest enclosing circle when the structure is queried. The approximate convex hull consists of the (at most $2k$) extreme points in k regularly sampled directions. We will call this the k-approximate convex hull. An example is shown in Fig. 7. Thin lines are drawn through the points on the hull to indicate the directions in the points are extreme. The approximate enclosing circle is not the smallest enclosing circle. There is a point that is not enclosed (indicated by an arrow). However, Theorem 4 states that the two circles cannot differ much.

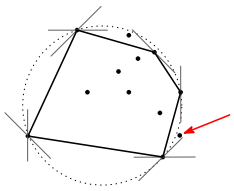


Figure 7: The 4-approximate convex hull of a point set and its smallest enclosing circle.

THEOREM 4. *Given a point set P. Let r and r' be the radii of the smallest enclosing circle of respectively P and its k-approximate convex hull. Then,*

$$1 \geq \frac{r'}{r} \geq 1 - \frac{1}{2\cot(\frac{\pi}{2k})} = 1 - O(\frac{1}{k}).$$

To maintain the k-approximate convex hull we store the point set k times, each point set ordered in a different regularly sampled direction. Using an efficient set structure, insertion and deletion take $O(\log n)$ time per set, so $O(k \log n)$ time in total. Getting the k-approximate convex hull also takes $O(k \log n)$ time. Computing the smallest enclosing disk for this set of points can be done using a randomized incremental algorithm in $O(k)$ [6]

5. COMPUTING THE OPTIMAL SEGMENTATION

First we will discuss an algorithm that computes the optimal segmentation given an uncompressed start-stop matrix [1]. This dynamic programming algorithm is based on the following property:

THEOREM 5. *The optimal sequence of segments for $\tau[0, i]$ (if it exists) consists either of just one segment, or it is equal to an optimal sequence of segments for $\tau[0, j]$ appended with a segment $\tau[j, i]$, where j is an index such $\tau[j, i]$ is valid.*

Theorem 5 allows us to transform the segmentation problem to a shortest path problem in an unweighted directed acyclic graph on n vertices, having the start-stop matrix as adjacency matrix. To find this shortest path from 0 to n the program *SimpleComputeSegmentation* is used. In this dynamic program the optimal segmentation of $\tau[0, i]$ is computed for all $i = 0, \ldots, n$. Instead of storing the complete segmentation for each i, only the starting index of the last segment (*last*) and the total number of segments (*count*) is stored. The actual segmentation can be retrieved from the dynamic programming table in $O(n)$ time.

Algorithm *SimpleComputeSegmentation*(τ, C)
1. $Sg[0].last \leftarrow nil$; $Sg[0].count \leftarrow 0$;
2. **for** $i \leftarrow 1$ **to** n
3. **do** $Sg[i].count \leftarrow \infty$
4. **for** each j for which $\tau[j,i]$ satisfies C
5. **do if** $Sg[j].count + 1 < Sg[i].count$
6. **then** $Sg[i].count \leftarrow Sg[j].count + 1$;
7. $Sg[i].last \leftarrow j$;

For each index i the algorithm loops over all valid candidate segments $\tau[j,i]$, while maintaining the optimum. The optimal segment in this context is the segment $\tau[j,i]$ for which the optimal segmentation of $\tau[0,j]$ consists of the smallest number of segments.

Our approach is similar to *SimpleComputeSegmentation*: The algorithm *ComputeSegmentation* loops over all indices i and for each i it determines the optimal last segment. The difference is that our algorithm finds this optimal segment more efficiently. Instead of processing the valid starting indices one by one, a whole block of consecutive valid indices is processed at once. These blocks correspond to the blocks in the compressed start-stop matrix $\mathcal{S}$ at row i. Processing a block of valid indices is a complex operation. To allow for this operation the table Sg is changed to a balanced binary tree $\mathcal{T}$. A node in $\mathcal{T}$ corresponds to an entry in Sg and has three fields: *last*, *count* and *index*. The tree $\mathcal{T}$ is ordered on *index*.

Algorithm *ComputeSegmentation*$(\tau, \mathcal{S})$
1. Initialize empty $\mathcal{T}$;
2. Create new node ν_0;
3. $\nu_0.index \leftarrow 0$; $\nu_0.count \leftarrow 0$;
4. $\mathcal{T}.Insert(\nu_0)$
5. **for** $i \leftarrow 1$ **to** n
6. **do** Create new node ν;
7. $\nu.index \leftarrow i$;
8. $\nu.count \leftarrow \infty$;
9. **for** each block b at row i of $\mathcal{S}$
10. **do** $\nu' \leftarrow \mathcal{T}.GetMinimalCount(b)$;
11. **if** $\nu'.count + 1 < \nu.count$
12. **then** $\nu.count \leftarrow \nu'.count + 1$;
13. $\nu.last \leftarrow \nu'.index$;
14. $\mathcal{T}.Insert(\nu)$;

For each row i in the compressed start-stop matrix the query algorithm *GetMinimalCount* is executed c times, where c is the number of blocks at row i. Furthermore, a new element storing the optimal segmentation for $\tau[0,i]$ is created and inserted in $\mathcal{T}$ taking $O(\log n)$ time. Our algorithm runs on stable criteria. Hence the total number of blocks is $O(n)$.

The input of the *GetMinimalCount* query is a block b at row i of the compressed start-stop matrix. Block b covers the indices $b.begin, b.begin+1, \ldots, b.end$, which are starting indices of valid candidate segments ending that end at i.

The goal of the *GetMinimalCount* procedure is to find the candidate segment starting at $j \in [b.begin, b.end]$ for which the optimal segmentation of $\tau[0,j]$ consists of the smallest number of segments. This corresponds to finding the node in $\mathcal{T}$ with minimal *count* over all nodes with $index \in [b.begin, b.end]$.

To find this node one could simply visit all nodes with $index \in [b.begin, b.end]$ and keep track of the one with minimal *count*, but this can be done more efficiently. According to the binary search tree property these nodes with $index \in [b.begin, b.end]$ are located between the paths to $b.begin$ and $b.end$. Figure 8 shows an example.

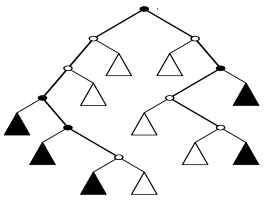


Figure 8: A tree $\mathcal{T}$ and two search paths. The nodes in the white subtrees and the white nodes on the paths are between the two search paths.

A node ν is between the paths to $b.begin$ and $b.end$ if and only if it is
1. on the path to $b.begin$, with $\nu.index \geq b.begin$, or
2. on the path to $b.end$, with $\nu.index \leq b.end$, or
3. in the right or left subtree of a node in respectively set 1 or 2.

There are at most $O(\log n)$ nodes in set 1 and 2, and there can be a linear number of nodes in set 3. However, the number of subtrees in set 3 is at most $O(\log n)$. To speed up the search the balanced binary search tree $\mathcal{T}$ is augmented with the fields $\nu.min_{count}$ and $\nu.argmin_{count}$, which are equal to respectively the minimal value of *count* over all nodes in the subtree rooted at ν and a pointer to the node where this minimum occurs. This enables us to find the node with minimal *count* in a subtree in constant time. The algorithm *GetMinimalCount* can thus loop over all nodes (set 1 and 2) and subtrees (set 3) located between the search paths to $b.begin$ and $b.end$, while maintaining the node with minimal *count*, taking $O(\log n)$ time in total. Hence *ComputeSegmentation* takes $O(n \log n)$ time.

6. SEGMENTATION BY STATES

The most natural way to define a criterion for the segmentation according to behavioral movement states is a disjunction of subcriteria, of which each subcriterion corresponds to a behavioral state:

$$\text{Behavior } 1 \vee \text{Behavior } 2 \vee \cdots \vee \text{Behavior } m.$$

As was described in Section 3.3, segmenting according to such a disjunction can be done by taking the union of the compressed start-stop matrices. However, taking the union of the compressed start-stop matrices makes us loose valuable information: the segmentation algorithm cannot distinguish between different classes of segments.

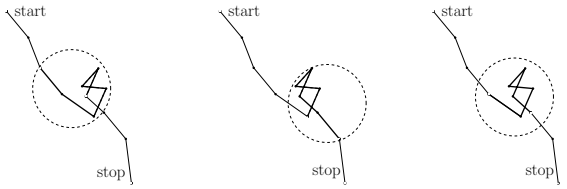
The algorithm *ComputeSegmentation* can be changed in such a way that this information remains. In this extended version the input consists of the m compressed start-stop matrices that correspond to the m behavioral states. The loop of lines 9-13 over all blocks of the compressed start-stop matrix is executed once for each of the m start-stop matrices. Given the m individual start-stop matrices, classification of the segments is possible. To store the classification each node will have one extra field storing the movement state of the last segment. When the optimal starting point of the last segment is changed on line 13 the movement state corresponding to the current compressed start-stop matrix is assigned to this field. The running time of this segmentation/classification algorithm is $O(mn \log n)$.

6.1 Adding optimization goals in case of ties

A segment is put into the first class, whose criterion it satisfies. Intuitively, the order in which the start-stop matrices are handled corresponds to the order of "preference". However, this order does not yield real guarantees. One could also order all optimal segmentations (equal segment count) by the number of segments of class i that they contain (and in case of ties, on the number of segments of class i', etc). Keeping track of the number of segments of each class and doing a lexicographical compare instead of the simple comparison of line 11 takes only $O(m)$ additional time, which does not affect the asymptotic running time.

There is another way of breaking ties that is especially useful when segmenting trajectories of animals that show *pausing* or *stopping* behavior (see Section 8 for an example). This kind of behavior can usually be described by a disk criterion. Segmenting without defining extra rules for breaking ties, can yield strange results biased towards a certain direction (forward or backward). For example, consider the trajectory in Fig. 9. Assume that it is segmented according to a disjunction criterion containing a disk and bounded turning angle criterion ($\leq 60°$). Making the last segment as long as possible would result in the segmentation in Fig. 9(a). Making the last segment as short as possible results in the situation in Fig. 9(b). Both of them are not what we would expect from a good segmentation. They are biased towards respectively the front or the back: the segment that satisfies the disk criterion ends in a strange "limb" at its front- or back-end.

To counteract this biased behavior rules can be defined to break ties based on the segment classes. In Fig. 9, one could make the algorithm pick the last segment as short as possible if it satisfies the bounded turning angle criterion, and pick the last segment as long as possible if it satisfies the disk criterion. This would result in the segmentation in Fig. 9(c), which is preferred over the other two segmentations.



(a) Last segment as long as possible (b) Last segment as short as possible (c) Class dependent tie breaking

Figure 9: Three optimal segmentations. Pausing segment is depicted thicker.

The tie breaking strategy is easy to incorporate in the algorithm. Simply pick the left-most or right-most valid starting index that is reported by *GetMinimalCount*. Note that two different variants of *GetMinimalCount* are needed: one to find the left-most and another to find the right-most valid optimal starting index within the block. To enable this query on $\mathcal{T}$, it needs to be augmented with two different $argmin_{count}$ fields, one to store the left-most and another to store the right most-node where min_{count} occurs. This can be done without affecting the asymptotic running time.

6.2 Follow relations between movement states

In practice, transitions between movement states are not occurring at random, they are often well-structured. The transitions can be modeled by a (weighted) graph on m vertices. We will present a method to enforce those transition relations on a segmentation.

This method requires the computation of m segmentation trees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$, instead of one (as described in Section 5). Each tree $\mathcal{T}_c$ stores the optimal segmentation for the subtrajectory $\tau[0, i]$ that ends in movement state c for all $i = 1, \ldots, n$.

Consider the computation of the optimal segmentation for $\tau[0, i+1]$ ending at state c. Assume that c is only preceded by states $p_1, p_2, \ldots, p_d$ according to the transition graph. As before, the start-stop matrix of state c determines which segments (ending at $i+1$) are valid. Recall that in the unrestricted setting the starting index of the last segment is chosen such that it started at the index j for which the optimal segmentation of $\tau[0, j]$ has minimal segment count. This index can be computed efficiently using the *GetMinimalCount* queries. In the restricted case this definition of best segment changes: the last segment starts at the index j for which the optimal segmentation of $\tau[0, j]$ ending at p_1, or p_2, or $\ldots$, has minimal segment count. To compute this index we execute the *GetMinimalCount* query on $\mathcal{T}_{p_1}, \mathcal{T}_{p_2}, \ldots, \mathcal{T}_{p_d}$.

Enforcing hard follow restrictions can result in situations where no valid segmentation exists. An alternative approach is to penalize less likely transitions, and to minimize the total penalty of the segmentation. Our framework supports this approach too.

7. INTERACTIVE PARAMETER SELECTION

Most criteria that we have discussed involve parameters, such as the upper bound α on the angular range of the heading and the radius r of the covering disk, for example. Tuning these parameters is complex, and there is no well-defined ground truth. Hence, an interactive setting is needed. This interactive process can be guided by the stability of the parameter values. A value is *unstable* if a small change to its value can result in a large change in the segmentation, i.e., in a change of the number of segments. To measure the stability of a parameter value we run the segmentation algorithm multiple times with different parameter values and count the number of segments for each of the resulting segmentations. The more stable value ranges correspond to the "flat" parts of the step function, that is, the parts with the least variation in number of segments (see Section 8 for an example).

To compute the stability of parameter values, we need to compute the segmentation of the same trajectory multiple times using the same criterion, but with different parameter values. For decreasing criteria, information can be reused in the different runs. For this purpose we use the double-and-search method described by Buchin *et al.* in [2], but instead of testing validity for segments directly, we query a data structure that is constructed once and used in all runs with different parameter values. For most decreasing criteria this yields a significant speed up of the double-and-search method. The running time is reduced from $O(n \log n)$ to $O(k \log n)$ time, where k is the number of segments.

The data structure is basically a table that stores certain criterion-dependent information for all segments of length $1, 2, 4, 8, \ldots$. For instance, for the range criterion on speed,

the maximal and minimal speed is stored, and for the (approximate) disk criterion the extreme points in all k directions are stored. The data structure can efficiently (in $O(\log n)$ time) test the validity of any segment given a certain parameter value. In case of the range criterion it can check whether a certain segment is valid by retrieving the maximum and minimum value from the tables and testing whether their difference is small enough. A naive use of the retrieve query, i.e., in each step of the double-and-search method, yields a running time of $O(k \log^2 n)$. However, by reusing information during the search, we can reduce this running time to $O(k \log n)$.

8. CASE STUDY

We assess the performance of our framework by analyzing two trajectories of migrating white-fronted geese [10] during their spring migration. The goal is to segment this data set into migration flight and stopovers (including wintering, breeding, and moult). Fig. 10 shows the segmentation as computed by our framework.

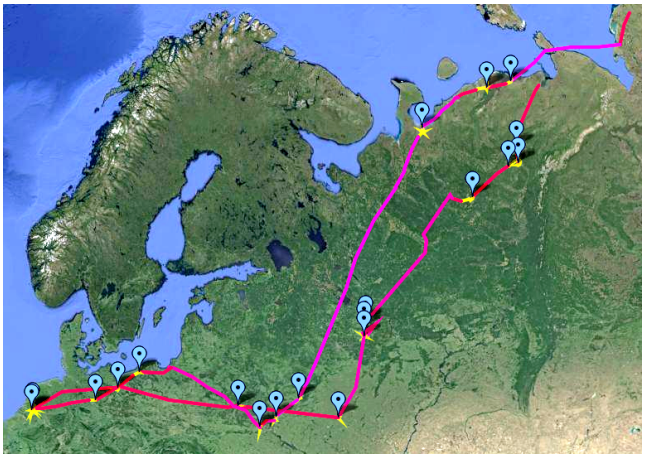


Figure 10: Trajectories of two migrating geese. Red/pink segments are flight, yellow segments are stopovers. Blue markers indicate end of a stopover.

The data was previously analyzed and manually segmented by domain experts [10]. A stopover segment is characterized by its limited variation in location. Therefore, the disk criterion is used for stopovers. Note that a stopover is not simply a stop, but a resting place, where a goose rests, flies (short stretches) and feeds for at least 48 hours [10] before moving on. During flight geese maintain the same heading for long stretches. This clearly differs from the seemingly undirected motion that can be observed at stopovers. To deal with this, we use the angular range criterion for heading for flight. On this data set, a disk with radius 30 km and an angular range of 1.7 radians yield the best segmentation result, that is, the segmentation that is most similar to the domain-expert's segmentation by hand.

Finding suitable parameters was an interactive process. Our search was aided by stability diagrams (see Section 7). The stability diagram of Fig. 11 was very helpful in our search for a good angular range bound. The diagram clearly shows a stable region between 1 and 2.5 radians. We picked three different values in this stable region: 1, 1.7 and 2.5 and

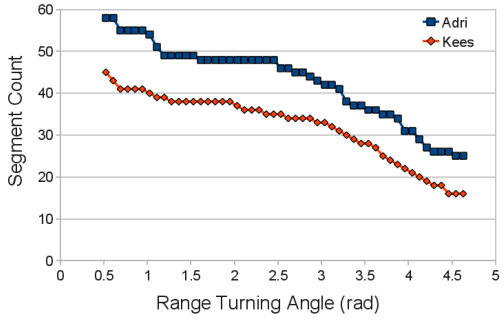
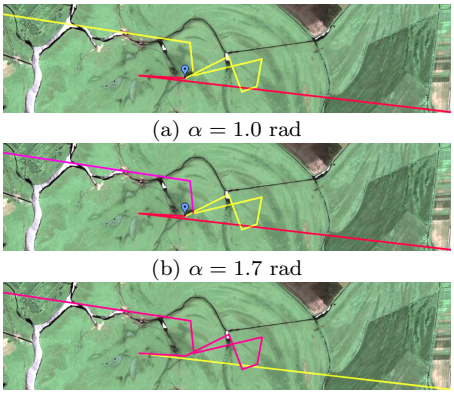


Figure 11: Stability diagram of the turning angle.

segmented according to those parameter settings. There is a trade-off between large stopovers that can cover parts of flight segments and small stopovers that leave some of the stop points uncovered that will be either incorrectly labeled as flight, or that will form extra stopovers. The middle value 1.7 proved to be a suitable compromise. A typical example motivating our choice is shown in Fig. 12.



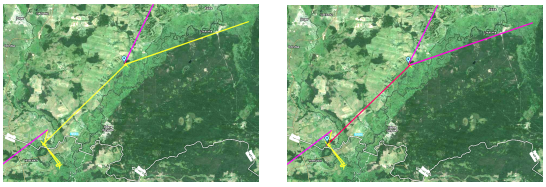
(a) $\alpha = 1.0$ rad

(b) $\alpha = 1.7$ rad

(a) $\alpha = 2.5$ rad

Figure 12: Segmentations for varying heading range bound α.

A similar analysis was done to find a suitable disk radius. The stability diagram indicated several stable regions. We tried one value from each of the most stable regions: 10 km, 30km and 40 km. We preferred 30km, because it resulted in stopovers of size similar to those in the manual segmentation. A radius of 40km mislabeled numerous migration flight segments as stopovers. A radius of 10km was good at detecting stops, but not at finding stopovers. For this 10km radius the manually labeled stopover segments were split in stops (labeled stopover) and non-migration flight (labeled migration flight) (see Fig. 13).

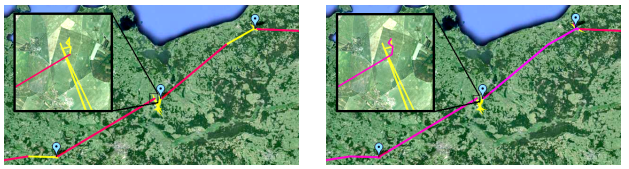


(a) r =30 km (b) r =10 km

Figure 13: Segmentations for varying radius r.

The criteria as discussed above are successfully detecting stopovers, but are mislabeling some short stops (< 48h) as stopover. That is why we placed a minimal duration criterion (48h) in conjunction with the disk criterion. This resulted in the short stops being labeled migration flight. Furthermore, we noticed numerous artificial splits in the stretches of migration flight, because geese do not maintain their heading on all data points in the migration flight (according to manual segmentation). Geese can fly in a completely different direction for a very small period of time after which they change back to the previous heading (zigzag). Allowing a constant number of outliers per segment would help the segmentation, since flight segments are then merged. However, a constant number of outliers causes small flight segments to cover a significant part of the stopovers, so many even that complete stopovers can be missed by the algorithm. Instead, we allow a number of outliers proportional to the number of points in the segment (using the method described in Section 2). In this specific case we have chosen an outlier percentage of 20%. Allowing this percentage of outliers effectively reduces the number of consecutive flight segments. Most of them are merged, which is preferred.

Just optimizing with respect to the number of segments, does not restrict the chosen breakpoints between segments. Thus, we added the rule that flight segments must be as short as possible and stopovers as long as possible using the method described in Section 6.1. This tie breaking method performs very well especially compared to the alternative: making flight as long as possible and stopovers as short as possible. See Fig. 14 for part of one of the segmentations.



(a) Long stops, short flight (b) Short stops, long flight

Figure 14: Segmentations with different tie breaking rules.

We conclude that our segmentation framework proved to be useful in practice. In contrast to previous approaches [3], our framework offers mechanisms for optimizing the breakpoints, choosing parameter values, and handling outliers in a more consistent way, which are effective in practice. The resulting segmentations are very close to the manual segmentation. Our labeling agrees with the manual labeling on respectively 96.3 and 92.6% of the total points.

9. CONCLUSIONS

We have introduced a framework for criteria-based segmentation that can efficiently (in $O(n \log n)$ time) handle a more powerful class of criteria than previous algorithms. It allows for segmentation by movement states and in contrast to previous methods we can add a broad range of state-based rules governing state transitions and additional optimization goals to fine tune the exact point of transitions. We have also introduced interactive parameter selection.

Our segmentation framework proved to be useful in practice and yielded segmentations similar to manual ones. Hence they can substitute these, where a manual segmentation is not possible, e.g., due to the size of data to be analyzed. In this context the interactive approach has a large potential. Incorporating advanced statistical methods into interactive parameter selection could guide the user even more.

On the algorithmic side, future work could focus on the more advanced tie breaking and state transition rules, especially with focus on outlier handling. Furthermore the results on efficient queries for different parameter values in decreasing criteria might be generalized to stable criteria.

Acknowledgments

The authors would like to thank the Vogelschutz-Komitee e.V. (VsK Hamburg), Alterra Wageningen-UR and the Dutch Society of Goosecatchers for the financial and technical support in the catching and tracking of the geese. K. Buchin is supported by the Netherlands Organisation for Scientific Research (NWO) under project no. 612.001.207.

10. REFERENCES

[1] B. Aronov, A. Driemel, M. J. van Kreveld, M. Löffler, and F. Staals. Segmentation of trajectories for non-monotone criteria. In *Proc. 24th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 1897–1911, 2013.

[2] M. Buchin, A. Driemel, M. van Kreveld, and V. Sacristan. Segmenting trajectories: A framework and algorithms using spatiotemporal criteria. *Journal of Spatial Information Science*, 3:33–63, 2011.

[3] M. Buchin, H. Kruckenberg, and A. Kölzsch. Segmenting trajectories based on movement states. In *Proc. 15th Internat. Sympos. Spatial Data Handling (SDH)*, pages 15–25. Springer-Verlag, 2012.

[4] F. Cagnacci, L. Boitani, R. Powell, and M. Boyce. Animal ecology meets gps-based radiotelemetry: a perfect storm of opportunities and challenges. *Philos Trans R Soc Lond B Biol Sci*, 365(1550):2157–62, 2010.

[5] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. In *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 1196–1202. ACM, 2006.

[6] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[7] S. Dodge, R. Weibel, and E. Forootan. Revealing the physics of movement: comparing the similarity of movement characteristics of different types of moving objects. *Computers, Environment and Urban Systems*, 33(6):419–434, November 2009.

[8] R. C. Gonzalez and R. E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[9] J. Harguess and J. K. Aggarwal. Semantic labeling of track events using time series segmentation and shape analysis. In *Proc. 16th IEEE Internat. Conf. Image Processing*, pages 4261–4264. IEEE, 2009.

[10] R. E. van Wijk, A. Kölzsch, H. Kruckenberg, B. S. Ebbinge, G. J. D. M. Müskens, and B. A. Nolet. Individually tracked geese follow peaks of temperature acceleration during spring migration. *Oikos*, 121(5):655–664, 2012.