Time Relaxed Spatiotemporal Trajectory Joins

Petko Bakalov
University of California, Riverside
pbakalov@cs.ucr.edu

Marios Hadjieleftheriou
Boston University
marioh@cs.bu.edu

Vassilis J. Tsotras
University of California, Riverside
tsotras@cs.ucr.edu

ABSTRACT

Many spatiotemporal applications store moving object data in the form of trajectories. Various recent works have addressed interesting queries on trajectorial data, mainly focusing on range queries and Nearest Neighbor queries. Here we examine another interesting query, the Time Relaxed Spatiotemporal Trajectory Join (TRSTJ) which effectively finds groups of moving objects that have followed similar movements in different times. We first attempt to address the TRSTJ problem using a symbolic representation algorithm, which we have recently proposed for trajectory joins. However we show experimentally that this solution produces false positives that grow rapidly with the increase of the problem size. As a result, it is inefficient for TRSTJ queries as it leads to large query time overhead. In order to improve query performance, we propose two important heuristics that turn the symbolic represenation approach effective for TRSTJ queries. Our first improvement, allows the use of multiple origins when processing strings representing trajectories. The experimental evaluation shows that the multiple-origin approach drastically reduces query performance. We then present a "divide and conquer" approach to further reduce false positives through symbolic class separation. The proposed solutions can be combined together, which leads to even better query performance. We present an experimental study revealing the advantages of using these approaches for solving Time Relaxed Spatiotemporal Trajectory Join queries.

Categories and Subject Descriptors

H.3.1 [**Indexing Methods**]: Information Storage and Retrieval

General Terms

Algorithms, Experimentation, Performance

This work was partially supported by NSF grants IIS 0220148 and IIS 0330481.

GIS'05, November 4, 2005, Bremen, Germany.

Keywords

Trajectory, Join, Indexing

1. INTRODUCTION

Spatiotemporal data is generated in many novel applications like traffic management, mobile communications, RFID systems, surveillance systems, location based services [20], etc. Such applications create large amounts of (historical) data that are gathered for the purpose of further analysis. One common form of spatiotemporal data consists of moving object trajectories. A trajectory corresponds to a sequence of $\langle location/time \rangle$ pairs, storing the location of a given object at various points in time. Many indexing schemes have been proposed to index the spatiotemporal data [12, 16, 24, 5, 8, 9, 7, 17, 6]. Various interesting queries can then be answered over a spatiotemporal archive [18, 26, 28, 23, 6, 21]. Typical queries involve range or nearest neighbor predicates, for example: find all objects that crossed through area A during time interval I, or find the trajectories that were closed to point B at time t. Recently, in [3] we introduced a new class of trajectory-join queries, which identify pairs of trajectories that have similar behavior during a user-specified time interval.

Consider for example two sets of trajectories R and S, and assume we want to find pairs of trajectories that evolved similarly for a time interval with duration δt. The trajectory similarity is defined by their spatial closeness (expressed by a spatial threshold ϵ around each trajectory) which should last at least for an interval with duration δt. In the join definition presented in [3] the user specifies the beginning δt_{begin} and the end δt_{end} of the time interval δt. As a result, the time interval δt is binded with the time domain (time-dependent trajectory join).

In this paper we introduce the Time Relaxed Spatiotemporal Trajectory Join (TRSTJ); here the interval δt can be *anywhere* in the time domain of each trajectory. It is thus a more general version of the trajectory join problem that allows discovering similarities anywhere in the time domain. Consider for example a vehicle trajectory R_i that between 2pm and 3pm on 1/2/04 crossed the Lincoln tunnel and then passed by the Grand Central station. Assume we are interested in all trajectories in S that followed a similar behavior. A trajectory S_j that also crossed Lincoln tunnel and then passed within 50 yards (threshold ϵ) of the Grand Central Station, between 8am and 9am on 3/3/05 would match the join criterion for R_i. Note that the join predicates are fixed on the spatial domain (Lincoln Tunnel, within 50 yards of Grand Central) and on the duration of the time interval (δt

is 1 hour), but are relaxed on the time when the predicate happened (since δt is not anchored on the time domain).

In other applications we may also be interested in limiting the temporal distance between the matching events. Consider a surveillance application where in addition to how long trajectories have matched, it is also required that the corresponding matches occurred within some temporal distance T_d. For example, we want to identify suspects that passed by a meeting area within three hours from each other. In this variation, each δt is still not anchored on the time domain, but the relative time distance between the δts of corresponding trajectories is limited to $T_d = 3$ hours. Allowing the interval δt to move along the time domain increases the complexity of the spatiotemporal trajectory join.

While there has been work on various forms on spatial joins [13, 22, 19], trajectory joins are novel queries that are actually very useful for analyzing spatiotemporal data. Traditional R-tree joins [4, 10, 15] are based on intersections between MBRs while spatiotemporal trajectory join conditions are more complex. Moreover, they are different from traditional similarity queries for time-series data. The typical approach for identifying similar time series results in an expensive evaluation [25, 26] and continuous precomputation of the query result. Trajectory joins also relate to queries in electronic ink databases [2]. Ink databases can also be viewed as spatiotemporal databases where the user searches for sequences of timestamped points in the plane (a trajectory), representing an electronic ink, similar to the input one. Similarity in ink databases however is measured in terms of probability, while in the trajectory joins examined in this paper the join criteria are deterministic (that is, a given trajectory either satisfies or does not satisfy the join criteria).

Instead, here we have user specific time intervals δt and T_d, which render the problem of trajectory joins more amendable to solutions that use specialized index structures. The brute force solution that compares every single trajectory in the first dataset with the trajectories in the second is exponential in terms of both time complexity and number of I/O operations. Such an approach would not be practical for large trajectory archives.

To avoid accessing and examining all trajectory combinations, in [3] we proposed using a specialized trajectory representation. Using a "symbolic" representation of trajectories, we grouped related trajectories based on their representation. We used this technique for the simple (time-dependent) trajectory join and showed that it can reduce the number of trajectory pairs that need to be compared. Nevertheless, it generates false positives, which means that a verification step is needed. Since this algorithm worked well for the time-dependent join, we considered if it could be applied to the TRSTJ problem as well. However in the TRSTJ case, because of the absence of binding in the temporal domain, it is necessary to compare all possible trajectory segments with length δt (instead of comparing only those restricted by δt_{begin} and δt_{end} as for the simpler time-dependent join case). Unfortunately, as we also verify experimentally (section 3.4), the number of false positives grows exponentially as the size of the problem increases, thus greatly affecting the query performance.

We thus need new approaches to solve the large number of false positives introduced in the TRSTJ problem. In particular, we present two new heuristics that can drastically improve the TRSTJ join performance. One cause for creating false positives is due to the incorrect assignment of a trajectory to a group. The idea behind the first heuristic (multiple-origin evaluation, section 4) is to improve the grouping of related trajectories and in this way reduce the number of false positives. The idea behind the second heuristic is to divide the problem into smaller problems using a "class separation join" (section 5) that can then be solved independently. It is also possible to combine these two heuristics and achieve even bigger improvement in terms of join performance for the TRSTJ query.

The rest of the paper is organized as follows: Section 2 provides the problem definition, while Section 3 presents an overview of the basic symbolic representation algorithm as well as experimental results confirming its ineffectiveness for the TRSTJ query. Sections 4 and 5 describe the proposed improvements, including experimental evaluations that reveal their drastic effect in join performance. Finally, Section 6 concludes the paper.

2. PROBLEM DEFINITION

For simplicity we assume that a moving object trajectory is defined as a sequence of location/time instant pairs. Other trajectory representations should be easily reduced to this general form by interpolation. More formally:

DEFINITION 1. *A trajectory X is a sequence of $\langle$location /time$\rangle$ pairs: $\{\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$, where $x_i \in \mathbb{R}^d, t_i \in \mathbb{N}$*

Given trajectory sets R and S, the TRSTJ query returns all trajectory pairs (R_i, S_j) where R_i and S_j have been spatially close (i.e., no further than threshold ϵ) to each other continuously for at least some given time interval δt. Again, there is no restriction where this time interval δt appears in the trajectory lifetime (i.e., it is not anchored in the time domain). For the formal definition of the TRSTJ query we need the notion of trajectory 'segment' and 'match':

DEFINITION 2. *Given some trajectory X, a trajectory segment X' with length δt starting at time t_0, is a sequence of $\langle$location/time$\rangle$ pairs which is a sub subsequence of trajectory X and includes all time instants in $\{t_0, \ldots, t_0 + \delta t\}$*

DEFINITION 3. *Given spatial threshold ϵ and time interval of length δt, two trajectories R_i and S_j* match *if there exist segments R'_i in R_i and S'_i in S_j with length δt and starting times t_r and t_s respectively, such that, for every time instance t_i between 0 and δt the spatial distance between trajectory R at time instant $t_r + t_i$ and trajectory S at time $t_s + t_i$ is no more than the threshold ϵ.*

Hence, two trajectories match if there exists a time interval with the same length δt such that the distance between the physical locations of the two trajectories during these intervals is no more than the spatial threshold ϵ. In this paper we consider Euclidean distance as the metric of spatial closeness.

DEFINITION 4. *The trajectory segments during the time intervals - $(t_r, t_r + \delta t)$ for trajectory R_i and $(t_s, t_s + \delta t)$ for S_j are called matching parts.*

Figure 1 shows the matching parts (shown in bold) for two trajectories. We are now ready to define the Time Relaxed Spatiotemporal Trajectory Join.

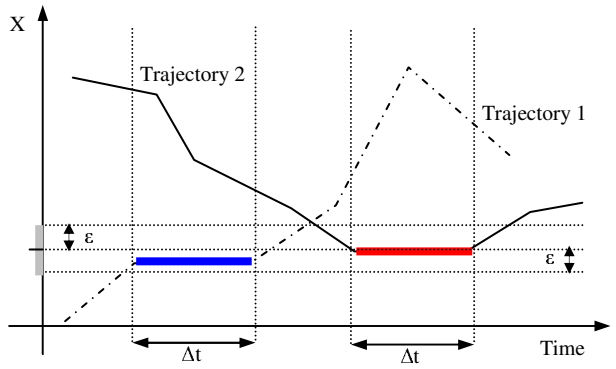

Figure 1: Time Relaxed Spatiotemporal Trajectory Join: the matching parts between the two trajectories are shown.

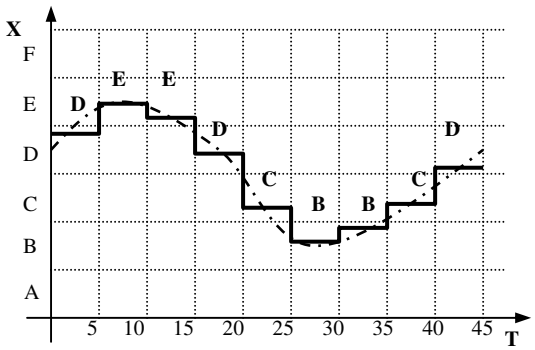

Figure 2: An object trajectory with its PAA representation; the string representation is: DEEDCBBCD.

DEFINITION 5. *Given two sets of object trajectories R and S, threshold ϵ and time interval δt, the result of the TRSTJ operation is the set of pairs (R_i, S_j), such that: (i) $R_i \in R$ and $S_j \in S$, and (ii) there exists a match between R_i and S_j with time duration δt.*

One interesting variation of the TRSTJ is to add the extra restriction that the relative matching between two trajectories should occur within some time distance. That is, the match in the second trajectory should occur within a given time interval after the match occurrence in the first trajectory. In this case not all matching pairs satisfy the join condition. With small extensions, the algorithms presented for the general TRSTJ problem, can be adapted to answer the restricted version of the join as well. For space limitations, the discussion is concentrated on the general TRSTJ problem but we present some experimental results showing the applicability of our approaches to that problem as well.

It should be noted that the trajectory join examined in [3] requires that the matching starting times are the same (i.e., $t_r = t_s$); it is thus a special case of the Time Relaxed Spatiotemporal Trajectory Join. One approach to solve TRSTJ queries is to use the symbolic representation approach we introduced in [3]. This is described in summary in the next section. However, in terms of query processing, TRSTJ needs to compare many more trajectory segments. For example within a trajectory with length 100 time instants, there can be 90 segments with length $\delta t = 10$ but only one segment with length $\delta t = 10$ starting at position $t_{start} = 20$. As we will show, the larger number of examined segments create an even larger number of false positives, which make this solution ineffective for large trajectory archives.

3. USING THE BASIC SYMBOLIC REPRESENTATION ALGORITHM

In order to process the trajectories for the purpose of efficient similarity search we need a lower-bounding distance function that can be computed efficiently for arbitrary time-intervals between pairs of trajectory segments, without having to access the raw trajectory data. This can be done by using an *approximate representation* of the trajectories only for the purpose of computing relevant lower-bounding distances between the trajectory segments. This way, a large volume of the exact trajectory representations that do not qualify for threshold ϵ can be pruned safely, by referring only to the small, approximate dataset. Because we work with approximated data, we need a post filtering step to eliminate the false alarms produced by the approximations. Since the function is lower-bounding we have only false positives but not false negatives. We can specify approximation accuracy and in this way we can adjust the number of false positives introduced in the intermediate result.

3.1 Symbolic Representations for Trajectories

The symbolic representation we proposed in [3] for trajectories is based on the Piecewise Aggregate Approximation (PAA) technique introduced in [11, 27, 14] that transforms time-series data into a string. For ease of exposition, here we present the basic concepts behind symbolic representations for 1-dimensional trajectories. The extension to multidimensional data is straightforward.

PAA accepts as input a trajectory of length n of $\langle location/time \rangle$ pairs and produces as output an approximation of reduced size, say m ($m << n$). The input sequence is divided into m equi-sized "frames" along the temporal domain and the spatial values contained in each frame are replaced by the average of these values. More formally:

DEFINITION 6. *Given trajectory $X = \{< x_1, t_1 >, \ldots, < x_n, t_n >\}$ of length n and a target length $m \ll n$, PAA produces an approximate trajectory $\bar{X} = \{< \bar{x}_1, t_1 >, \ldots, < \bar{x}_m, t_m >\}$ where the spatial values contained inside each time frame $[\frac{n}{m}(i-1), \frac{n}{m}i], 1 \leq i \leq m$ are replaced by their arithmetic mean:*

$$\bar{x}_i = \frac{m}{n} \sum_{j=\frac{n}{m}(i-1)+1}^{\frac{n}{m}i} x_j$$

The advantage of PAA is that the length of the reduced trajectory m can be chosen at will, thus the accuracy of the resulting approximation can be tuned freely.

As a second step the PAA approximation can be discretized by using a symbolic representation. In [3] we discretized the PAA approximations by using a uniform space grid and assigning a unique symbol to every partition of the grid. An example is shown in Figure 2. The symbolic representation can formally be defined as follows:

DEFINITION 7. *Given a uniform grid with granularity τ assign an alphabet of symbols $\mathcal{A} = \{\alpha_1, \ldots, \alpha_w\}$ such that $\forall 1 \leq j \leq w : [\tau(j-1), \tau j) \rightarrow \alpha_j$ (every symbol is assigned to a unique interval of the grid). A trajectory X of length n can*

be approximately represented as a string $\tilde{X} = \langle \tilde{x}_1 \cdots \tilde{x}_m \rangle$ *of length* $m \ll n$*, by replacing every value* $\bar{x}_i$ *in the* m*-length PAA approximation of* X *with symbol* $\tilde{x}_i = \alpha_j$ *such that* $\tau(j-1) \leq \bar{x}_i < \tau j$*.*

In the rest, we will refer to this approximation as the *symbolic representation.*

3.2 Symbolic Distance Measures

Having defined a symbolic representation for trajectories we need a distance function that appropriately lower-bounds the given trajectory distance function $\mathcal{D}$. Assume in the rest for simplicity that a Euclidean distance function is used:

$$\mathcal{D}_{\delta t}(X, Y) = \sqrt{\sum_{i \in \delta t} (x_i - y_i)^2}$$

In the simple 1-dimensional case it can be proven that the following distance on the symbolic representations is always a lower-bound of the Euclidean distance:

$$\tilde{\mathcal{D}}_{\delta t}(\tilde{X}, \tilde{Y}) = \sqrt{\frac{n}{k}} \sqrt{\sum_{i \in \delta t} d(\tilde{x}_i, \tilde{y}_i)^2}$$

where $i \in \delta t$ corresponds to all frames completely covered by time-interval δt (i.e., the total number of symbols in the sting representation contained in δt), k is the total number of such frames, and distance d between two alphabet symbols.

3.3 General Trajectory Join Algorithm

We can now proceed with the description of the general trajectory join algorithm. We are given two datasets R and S and we want to evaluate a TRSTJ query with threshold ϵ and time-interval δt. Assume for simplicity and without loss of generality that all trajectories have the same length n and that we approximate them using symbolic representations of length m. Assume that δt covers completely a total of k frames.

Two trajectories R_i and S_j will match if they contain trajectory segments R'_i and S'_j of length δt that are within distance ϵ. Having the lower bound property of the distance function $\tilde{\mathcal{D}}$ we can transform the problem into the symbolic domain and work with the corresponding trajectory strings instead of the raw trajectory data. This is advantageous since the string representations are very compact and much smaller than the actual trajectory data. In the symbolic domain we will produce a set of candidate pairs for the join result that contains all the results, plus false positives; thus a verification step is also required.

3.3.1 Sliding Window Evaluation

To solve the TRSTJ problem in the symbolic domain we need to find pairs of strings that have subsequences with length k for which the corresponding string symbols are not further apart than $\tilde{\epsilon}$, i.e. $d(\tilde{x}_i, \tilde{y}_i) \leq \tilde{\epsilon}$, for $i = 1, ...k$. Here $\tilde{\epsilon}$ is the translation of threshold ϵ in the symbolic domain. This can be accomplished by first generating all subsequences of length k for each string and consecutively use a sliding window algorithm with an appropriately scaled threshold $k\tilde{\epsilon}$. In particular, we order each subsequence according to its $\tilde{\mathcal{D}}$ distance from some origin $\tilde{O}$. The origin can be selected arbitrarily, as long as it is the same for both datasets taking part in the join. For example, we can use as an origin the string that corresponds to the lower left corner of the original space (e.g., $AA\cdots$). Each subsequence can be thought as a point in a k-dimensional space and these points are then placed on a line according to their distance from origin $\tilde{O}$. An example is shown in Figure 3, where subsequences are labeled according to the dataset they belong to (i.e., $\tilde{S}'_1$ is a subsequence from some trajectory in set S).

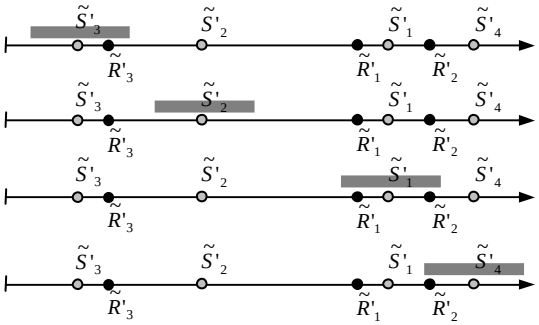

Figure 3: Sliding window algorithm. After ordering the symbolic distances according to origin string $\tilde{O}$ we place the midpoint of the window on every subsequence from dataset $\tilde{S}$.

Table 1: Dataset Characteristics.

N of trajectories	100 000
Size of the universe	1000 × 1000 Km
Simulation length	400 minutes
Initial distribution	Uniform

Since $\tilde{\mathcal{D}}$ is a metric, if two subsequences have origin-relative symbolic distance larger than $k\tilde{\epsilon}$, then there exists at least one frame where the corresponding subsequences have symbols that are farther than $\tilde{\epsilon}$. This means that the actual trajectories lie farther apart than ϵ for at least one time-instant inside δt. On the other hand, the inverse is not true and hence this technique will introduce false alarms.

The sliding window algorithm works in two steps. First, we set the length of the window to $2k\tilde{\epsilon}$ and place the midpoint of the window on the first subsequence from dataset S, say $\tilde{S}'_j$ (this would be $\tilde{S}'_3$ in the example of Figure 3). For all subsequences $\tilde{R}'_i$ of dataset R falling inside the window, we report pairs $\langle \tilde{S}'_j, \tilde{R}'_i \rangle$ as possible join candidates. Then, we slide the window and place its midpoint on the next subsequence in $\tilde{S}$ on the 1-dimensional line, and so on. In the second step we load the actual trajectory data for all candidate pairs reported by the sliding window method and verify the results.

Finally, it should be noted, that the symbolic representation can be organized as an index structure so as to minimize accessing relative subsequences efficiently ([3]).

3.4 Experimental results

To investigate the applicability of the original symbolic representation algorithm for solving TRSTJ queries, we performed an experimental evaluation using data generated from a moving object simulator [1]. We simulated the behavior of moving objects using the road system of Illinois. The properties of the generated data are shown in table 1. In the experimental results we report the number of I/O operations since it is the prominent factor in the time spent. CPU computations were much less time consuming.

Algorithm 1 General Trajectory Join Algorithm

Input: Query $\mathcal{Q} = \{R, S, \delta t, \epsilon\}$
Output: Set of pairs (R'_i, S'_j) such that there a match between R_i and S_j with time duration δt

```
1: Set U ← ∅, V ← ∅, P_B ← ∅
2: Find Origin O
3: Compute Approximation Õ
4: for all subsequences R'_i in R do
5:     Compute Approximation R̃'_i
6:     R̃'_i.score ← D̃_δt(R̃'_i, Õ)
7:     U.push(R̃'_i)
8: for all subsequences S'_j in S do
9:     Compute Approximation S̃'_j
10:    S̃'_j.score ← D̃_δt(S̃'_j, Õ)
11:    U.push(S̃'_j)
12: U.sort()
13: i ← 0
14: while i ← U.size do
15:    Entry X = U[i]
16:    if X ∈ R then, FindPairsInWindow(X,i,U,V,ε)
17: while V not empty do
18:    Entry < R'_i, S'_j >= V.top
19:    if (R'_i) ∈ R and (S'_j) ∈ S satisfy the criteria then
20:        P_B.push(R'_i, S'_j)
21: Return P_B
```

Algorithm 2 FindPairsInWindow

Input: X,i,U,V,ϵ

```
1: Compute kε̃
2: j ← i
3: while U[j].score - X.score < kε̃ do
4:    Entry Y = U[j]
5:    if Y ∈ S then, V.push(X, Y)
6:    j − −
7: j ← i
8: while U[j].score - X.score < kε̃ do
9:    Entry Y = U[j]
10:   if Y ∈ S then, V.push(X, Y)
11:   j + +
```

For these experiments we tested the behavior of the symbolic representation algorithm with 100 random queries, each with time interval δt varying from 30 to 100 minutes. (i.e., from 7.5% to 25% of the lifetime of the trajectories). We performed four groups of experiments varying the similarity threshold ϵ from 5Km to 30 Km (i.e., from 0.5% up to 3% of the total space).

For every group we measured the I/O cost for index access and data access assuming unlimited main memory buffer. We also measured the number of candidate pairs reported by the algorithm and the actual number of hits (number of trajectory pairs which indeed satisfy the join criteria). The results are shown on Figure 4 and Figure 5.

Clearly there is a huge disproportion between the number of candidate pairs and the number of pairs in the join result. This disproportion is caused by the very large number of trajectory segments (and thus string subsequences) generated. For example, with 100K trajectories that last for 400 minutes we need to generate about 37M trajectory segments with length 30 minutes and $(37M)^2$ possible pairs between them. We conclude that the number of false positives generated needs to be drastically reduced (in orders of magnitude) before the symbolic representation algorithm can be used for the TRSTJ problem.

Because the join criteria in the TRSTJ queries are more

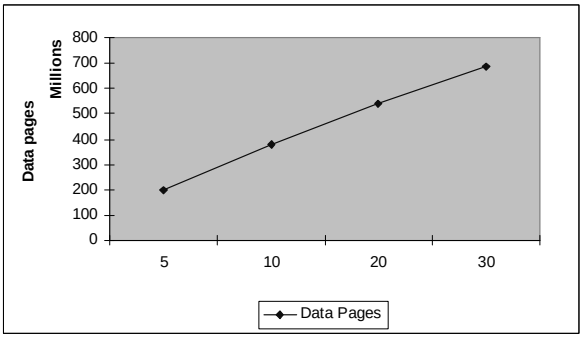

Figure 4: Number of comparisons.

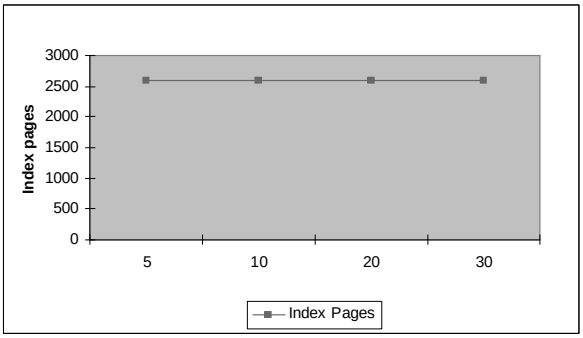

Figure 5: Number of actual matches.

relaxed, an increase of the threshold ϵ increases dramatically the number of generated candidate pairs.

This drawback of the original symbolic representation algorithm is also depicted in Figure 6 and in Figure 7 where the data access cost dominates the index access cost again in orders of magnitude. The number of index pages accessed is constant and does not depend on the threshold ϵ. This is because the creation of trajectory subsequences depends on k, i.e., the query-specified δt. To compute the subsequence scores the whole index structure has to be read. However, the increased number of false positives leads to verification tests that need to be performed on the "raw" trajectory data (for each candidate pair we read the trajectory data from the storage).

The conclusion from these experiments is that the basic symbolic representation algorithm as described in [3] is *not* suitable for the TRSTJ problem. A better solution is needed. In the following section we present two modifica-

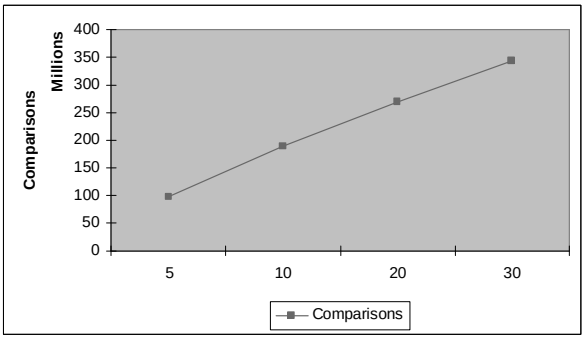

Figure 6: Data disc pages accessed.

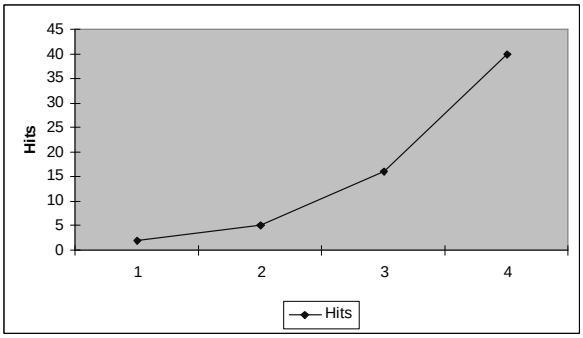

Figure 7: Index disc pages accessed.

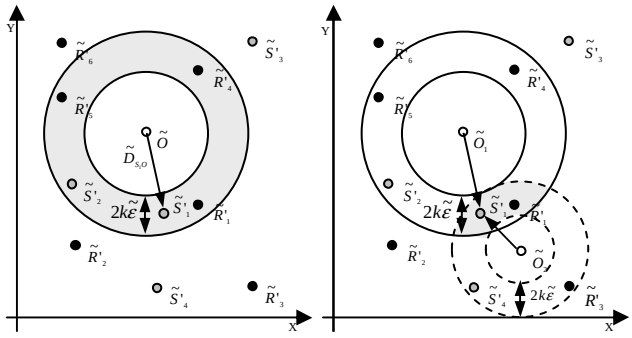

Figure 8: Graphical interpretation of the Multiple Origin Sliding Window Evaluation (MOSWE) algorithm

tions that lead into big reductions over the number of false positives generated.

4. MULTIPLE ORIGIN SLIDING WINDOW EVALUATION

The first modification we propose is based on a simple idea which however was very effective in reducing the number of false positives. In the original symbolic representation algorithm we use just a single origin $\tilde{O}$ and ordered the trajectory subsequences $\tilde{X}'$, produced by the trajectories from both sets $\tilde{R}$ and $\tilde{S}$, according to their distance to this origin $\tilde{\mathcal{D}}_{\delta t}(\tilde{O}, \tilde{X}')$. Instead we suggest to use multiple origins $\tilde{O}_0, \ldots, \tilde{O}_j, \ldots \tilde{O}_q$. Below we describe the Multiple Origin Sliding Window Evaluation (MOSWE) algorithm and verify its dramatic reduction of false positives through an experimental evaluation. Moreover, we propose an efficient method to access the trajectory data from the disk and compute their distances in an on-line fashion.

4.1 The MOSWE algorithm

We associate to every subsequence $\tilde{X}'$, a set of q scores $(w_0, \ldots, w_j, \ldots w_q)$, where score w_j represents the distance $\tilde{\mathcal{D}}_{\delta t}(\tilde{O}_j, \tilde{X}')$, between subsequence $\tilde{X}'$ and the corresponding origin $\tilde{O}_j$.

Before starting the sliding window evaluation, we sort the trajectory subsequences $\tilde{X}'$ using these q scores, in order of $w_0, w_1, ...$, etc. That is, if there is a group of trajectory subsequences having the same value of w_0, they are further sorted on their w_1 score and so on.

In order for a pair of subsequences to be reported as candidate pair by the sliding window algorithm, their corresponding difference in *each* score w_j should be less than the threshold distance $k\tilde{\epsilon}$. That is, these subsequences are considered "close" together if seen from *every* origin $\tilde{O}_j$. The number of reported candidate pairs will be much smaller and thus fewer I/O operations are needed to retrieve the raw trajectory data in the verification step of every reported candidate pair.

Figure 8 shows a graphical representation of the multiple origins approach. For simplicity, we have a 2-dimensional symbolic space which means that our strings consist of two symbols. The single origin symbolic representation algorithm is shown in Figure 8.a, during the processing of string $\tilde{S}'_1$. The distance between the origin approximation $\tilde{O}$ and string $\tilde{S}'_1$ is $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}}$ (this is also the score for this string). String $\tilde{S}'_1$ will be combined in a candidate pair with every string from the opposite set that is within distance $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}} \pm k\tilde{\epsilon}$ from the origin (subsequences from R trajectories that have score $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}} \pm k\tilde{\epsilon}$). Such strings are represented by points that lie in the shaded "ring" formed by the circles centered in $\tilde{O}$ with radiuses $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}} + k\tilde{\epsilon}$ and $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}} - k\tilde{\epsilon}$ respectively. In this example the reported candidate pairs will be $< \tilde{S}'_1, \tilde{R}'_1 >$, $< \tilde{S}'_1, \tilde{R}'_4 >$ and $< \tilde{S}'_1, \tilde{R}'_5 >$, even though strings $\tilde{R}'_4$ and $\tilde{R}'_5$ lie on distance larger than $k\tilde{\epsilon}$ from $\tilde{S}'_1$. Thus the pairs $< \tilde{S}'_1, \tilde{R}'_4 >$ and $< \tilde{S}'_1, \tilde{R}'_5 >$ are false positives which will be rejected during the verification step after accessing the raw trajectory data.

The new technique (MOSWE) appears in Figure8.b, for the same 2-dimensional strings, again when processing string $\tilde{S}'_1$. The number of origins is set to $q = 2$. The distances from string $\tilde{S}'_1$ to the origin approximations $\tilde{O}_1$ and $\tilde{O}_2$ are $w_1 = \tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}_1}$ and $w_2 = \tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}_2}$ respectively. String $\tilde{S}'_1$ will now be combined in a candidate pair with every string from the opposite set R that have distance to the origin $\tilde{O}_1$ in the range $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}_1} \pm k\tilde{\epsilon}$ and distance to the origin $\tilde{O}_2$ in the range $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}_2} \pm k\tilde{\epsilon}$. The graphical representation of this is an area bounded by the arcs of four circles: two centered in $\tilde{O}_1$ with radiuses $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}_1} + k\tilde{\epsilon}$ and $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}_1} - k\tilde{\epsilon}$ (shown with solid line), and two centered in $\tilde{O}_2$ with radiuses $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}_2} + k\tilde{\epsilon}$ and $\tilde{\mathcal{D}}_{\tilde{S}'_1\tilde{O}_2} - k\tilde{\epsilon}$ (shown with dashed line). The intersection of the two "rings" is shown shaded. Its size is clearly smaller thus reducing the number of reported candidate pairs.

In order for the sliding window algorithm to work efficiently, it needs a fast way to compute the subsequence distances from an origin. In the time-constrained join [3] we proposed an index structure that can identify quickly the subsequences corresponding to trajectory segments that intersect with the query specified time-interval δt. In this structure we store, for all trajectories, the symbols that correspond to the same frame. Along with every trajectory symbol we also store the identifier of the trajectory that this symbol belongs to. Figure 9 shows an example, where one page size is assumed to be able to store a maximum of three symbols along with their respective trajectory identifiers. The pages in every frame are stored sequentially on disk. Moreover, the first page of a frame directly follows the last page of the previous frame. To speed up the retrieval of the first page for the frame corresponding to the beginning of the query time-interval δt, the heads of the frame page lists are indexed by a B^+-tree.

However, for the TRSTJ problem trajectory segments can be anywhere in the time-domain (since the query-specified δt is not anchored on the time axis). We thus need a new

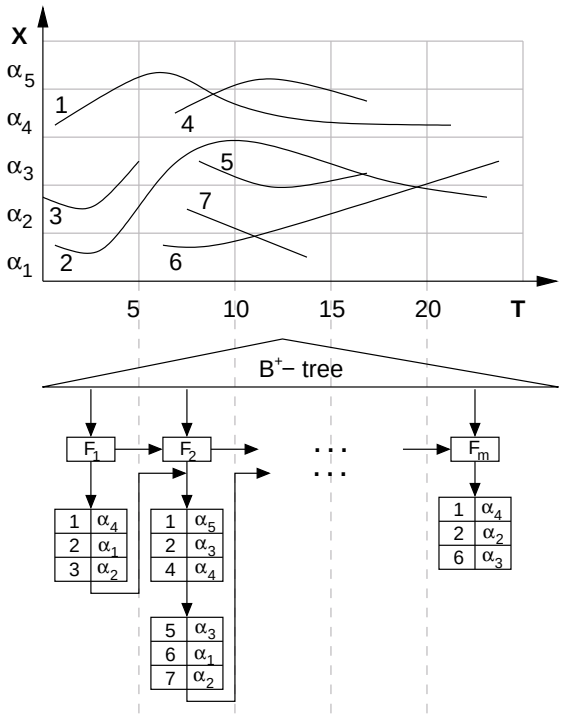

Figure 9: Indexing the symbolic representations.

access algorithm so as to produce efficiently the scores for *all* trajectory subsequences that have symbolic representation of length k . To do so, for every trajectory X_i we create a main memory list L_i with fixed length k. This list maintains the last k symbols seen from the corresponding trajectory i.e., it serves as a window of length k that slides in frame order over the symbolic representation of the trajectory. We start reading the data pages from the B^+-tree sequentially in frame order, starting with the first page of the first time frame, reading all pages of that frame and then continuing with the next frame. If in the current frame we find a symbol for trajectory X_i and list L_i has less than k symbols, we add it at the end of list L_i. If L_i has already k symbols, we first delete its first element (earliest symbol) before adding the new symbol from X_i. At any time frame, the score of the subsequence currently in each list is computed from the origins $\tilde{O}_0, \ldots, \tilde{O}_j, \ldots \tilde{O}_q$. As a result, all trajectory segment scores will be computed on the fly by reading every data page only once.

If the main memory is not enough to store the lists for all trajectories the segment scores can be computed on several passes. For example, if there is space for maintaining the list structure of v trajectories, the first pass computes the scores for the subsequences of the first v trajectories (trajectories with identifier in the range $1, \ldots, v$). During the second phase the scores for the subsequences of the next v trajectories are computed and so on. Since the symbolic representations within each frame are ordered by trajectory identifier, we can maintain the last page accessed in given pass for every time-frame and in the next pass we simply continue with the next page in that frame. This will allow accessing each data page only once (even though some non-sequential accesses will be introduced). If the number of subsequence scores cannot be kept in main-memory, external sorting algorithms can be utilized so as to order them before the sliding window algorithm emanates.

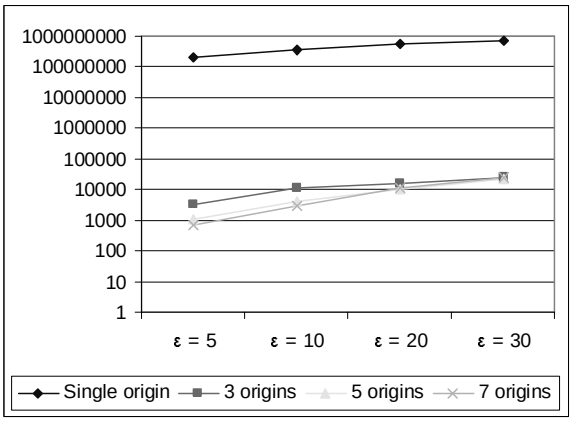

Figure 10: Number of I/O operations using different number of origins.

4.2 Experimental results

We proceed with an experimental comparison of the MOS WE approach with the basic symbolic join algorithm for the TRSTJ problem. Using the same dataset we run 100 randomly generated queries each with query interval length varying from 30 to 100 minutes. We run four sets of experiments with the number of origins varying from 1 (single origin sliding window evaluation) to 7. For every experiment, we run queries with different values of the threshold ϵ (varied from 5 to 30 Km). The results are shown in Figure 10 (the x-axis corresponds to the threshold value while the y-axis represents the number of I/Os).

The scale is logarithmic and shows clearly the orders of magnitude reduction that MOSWE offers against the single origin algorithm. Another interesting observation is that there is not substantial performance improvement when increasing the number of origins above three. That is, the MOSWE approach works well even with few origins, which makes it a very practical solution (less score bookkeeping etc.). Two origins worked well for the 2-dimensional data we used. Figure 11 shows the number of comparisons for the different number of origins as well as the number of hits (number of pairs which indeed satisfy the join criteria).

5. SYMBOLIC CLASS SEPARATION JOIN

Our second heuristic partitions subsequences in smaller groups or *classes* based on their spatial properties and solves the problem independently for each group. Each class is associated with a spatial area. Moreover, areas are non-overlapping and their union covers the whole spatial domain. Subsequences are assigned to the class in whose area the corresponding trajectory crossed during the first frame of the subsequence. The choice of frame is set arbitrarily; any of the k frames could be used for this matter.

This division heuristic is efficient for our purposes because we can decide whether it is possible to have a match between two trajectory subsequences by looking at the classes to which they belong. If the distance between the corresponding class areas is larger than the threshold ϵ then these trajectory subsequences cannot have a match (since there is at least one time instance for which the distance between

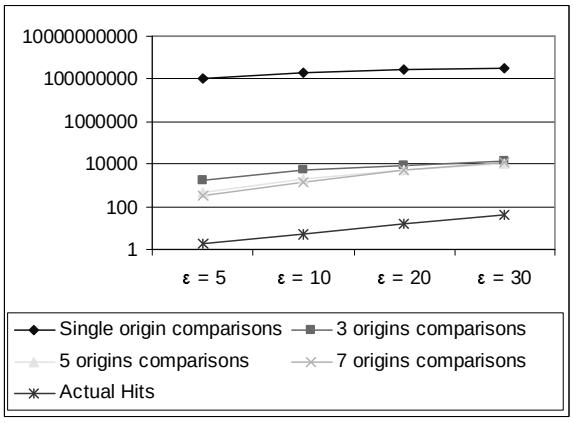

Figure 11: Number of comparisons performed and actual hits for different number of origins.

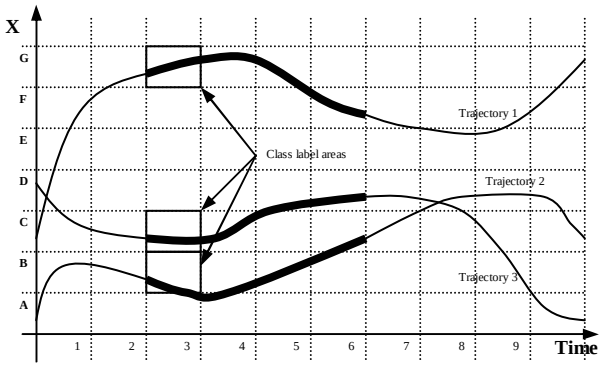

Figure 12: Assigning class labels to the trajectory segments

them is larger than the threshold ϵ). Thus, given a subsequence, all subsequences which can possibly match with it, belong to classes with areas within distance ϵ from its class.

Class partitioning can be easily applied on the symbolic representation space easily since we have already discretizied the spatial domain on non-overlapping areas by using a grid. Each cell has a different symbol (label) which can be used as a class label. For example Figure 12 depicts three 1-dimensional trajectory subsequences (shown in bold) spanning from time instance 3 to time instance 6. The approximated representation of these subsequences are: GGGF, CCDD and BABC. Using time (frame) 3 to decide the class assignments, the subsequence from trajectory 1 is sent to class G, the one from trajectory 2 to class C and the third to class B. If we have a join query with threshold value $\tilde{\epsilon} = 1$, by looking only at the class labels we can deduct that the subsequences from trajectory 1 and trajectory 3 cannot have a match. This is because for time instant 3 the distance between their symbolic representations (which is a lower bound of the actual distance) is $5 > 1 = \tilde{\epsilon}$. The same holds for the subsequences of trajectory 1 and trajectory 2. However, the distance between the class labels of the subsequence from trajectory 2 (C) and the one from trajectory 3 (B) is equal to the threshold $\tilde{\epsilon}$. Hence it is possible to have match between them and this pair cannot be pruned by this heuristic. It should thus be tested further by using the Multiple Origin Sliding window evaluation.

The process of assigning class labels to the trajectory sub-

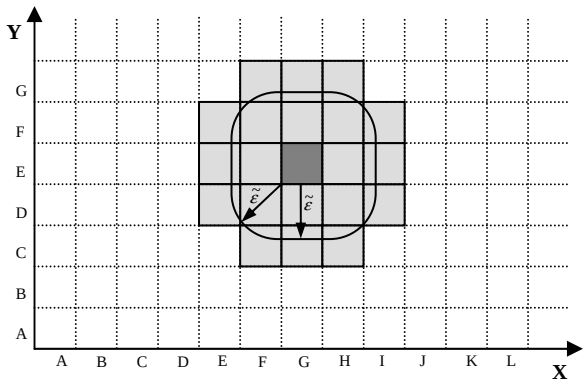

Figure 13: Class labels within distance $\tilde{\epsilon} = 1.3$ from class GE.

sequences is performed on the fly, as we read their symbolic representation from the index structure. Every time we form a symbolic representation and score for a subsequence in some list L_i we also assign a class label for this subsequence. Subsequences are stored as tuples: ¡trajectory id, start time, end time, score¿ and are hashed into groups based on their class label. The next step generates the candidate pairs from each class. The difference from the basic algorithm is that here candidate pairs are created by first merging the sorted subsequences in this class with the (sorted) subsequences from all classes which have label within distance $\tilde{\epsilon}$ from the given class. In the previous example the subsequence from trajectory 3 and the one from trajectory 2 belong to two different classes (B and C respectively) but since their class labels are within distance $\tilde{\epsilon}$ they have to be merged during the candidate pair computations. Figure 13 shows all class labels in a 2-dimensional spatial domain, which are within distance $\tilde{\epsilon} = 1.3$ from the class GE. The sorted lists are created per class, in a lexicographic order. This allows avoiding some duplication of effort (by merging pairs of classes the first time they are encountered).

5.1 Experimental results

The next set of experiments evaluates the effect of the class separation join heuristic. In particular, we examined the advantages of Class Separation over the already improved MOSWE algorithm for the TRSTJ queries. In the following figures MOSWE refers to the multiple origin algorithm while CSJ refers to the Class Separation Join on top of MOSWE. The query interval length was set to 30 minutes (7.5% of the total time) while the number of origins was set to 3.

We first examined the behavior of the algorithms for different sizes of available main memory buffer. This buffer is used by the join algorithms (MOSWE and the Class separation join) as a temporary storage during the sorting of the trajectory subsequence scores stored on the disc. From 100K trajectories lasting for 400 minutes we generate 3.5M trajectory subsequences each with length $\delta t = 50$ min. The number of buffer pages varied from 10 to 100000.

The advantage of the Class Separation algorithm is that trajectory subsequences are grouped in multiple groups (classes) and their sorting (according to their score from the origins) is performed only within the elements in a given classes and its neighboring classes (classes within distance $\tilde{\epsilon}$). In our experiments the average number of pages needed to store the

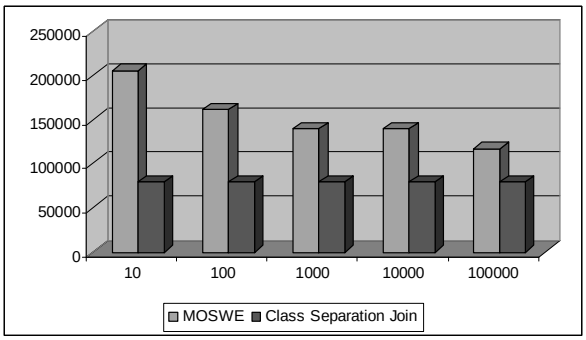

Figure 14: Number of I/O operations while varying the buffer size.

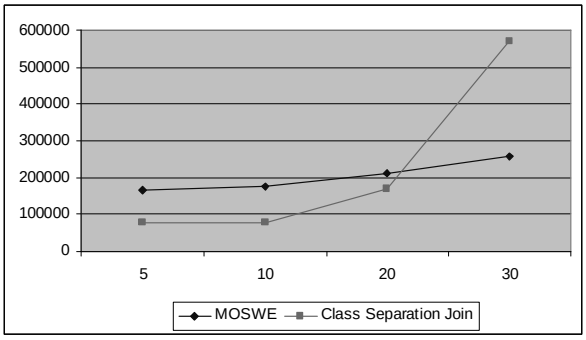

Figure 15: Number of I/O operations while varying threshold ϵ.

scores for a given class was only about 5 pages. In all experiments, the number of buffer pages available was enough to store the whole score relation. As a result, these scores can be sorted in main memory, without the need for an external sorting algorithm. For the MOSWE algorithm the situation was different. The total size of the list with scores was around 25K pages. The limited available buffer implies that an external sorting algorithm is needed with several passes to sort the score list. The results are shown of Figure 14. As expected, the number of I/O operations for the CSJ is always smaller than the I/O operations for MOSWE. However as the buffer size increases the difference becomes less important since the external sorting algorithm will require less steps.

Another important parameter that can affect the performance of the Class separation join is the threshold ϵ. With the increase of the threshold ϵ there will be more classes which have class label within distance $\tilde{\epsilon}$ from the current class. As a result, more lists will need to be merged by the Class Separation Join which will affect its performance.

In the following experiments we used four testing sets with the threshold ϵ varying from 5 to 30. The number of buffer pages is set to 100. The results are shown on Figure 15.

The increase in threshold ϵ results also in a moderate increase in the I/O accesses by the MOSWE approach, caused by the larger number of reported candidate pairs. For small values of the threshold ϵ the CSJ still shows better performance than MOSWE modification (around 52% less I/O operations for $\epsilon = 5$). However the increase of the threshold has a more substantial effect on CSJ, as the number of I/O operations increase in a quadratic fashion. For $\epsilon = 30$ the number of I/O operations for the CSJ is 220% larger

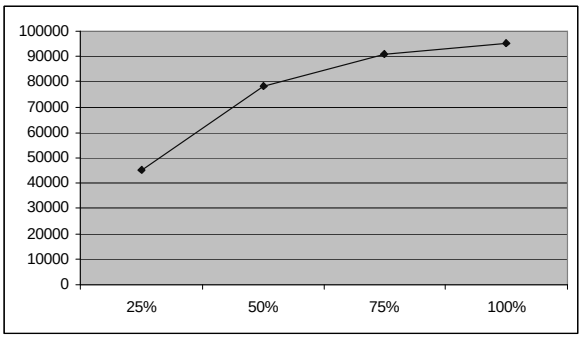

Figure 16: Number of I/Os operations for different time distance δt_{max}

than the corresponding number in MOSWE modification. The quadratic dependency between the threshold ϵ and the number of classes that are within distance $\tilde{\epsilon}$ from the current one is also shown schematically in Figure 13.

We next evaluated the Class Separation Join performance for the variant of the TRSTJ problem that has the extra restriction that the matching trajectory segments should be within some time distance δt_{max}. We implemented this additional restriction as an extra condition in the sliding window evaluation phase. Not only the corresponding difference in *each* score w_j should be less than the threshold distance $k\tilde{\epsilon}$ but also the starting times of the trajectory segments should be within time distance δt_{max}. In the experiments we varied time distance δt_{max} as a percent of the trajectory length which was set to 400 minutes. The results are shown on Figure 16. Here 50% means that the matching parts in the trajectories cannot be more than 200 minutes apart, while 100% means that the matching parts can be anywhere in the trajectory lifetime (i.e., the previous TRSTJ query). Again, the trajectory data access cost dominates the index access cost. The number of index I/O does not depend on the parameter δt_{max}. The trajectory data access cost however is proportional to the number of candidate pairs, and in this way proportional to δt_{max} (since this is another criterion that has to be satisfied by the candidate pairs).

In conclusion, the Class Separation Join approach is very efficient for small values of the threshold ϵ or when the main memory is limited. With the increase of the ϵ Class Separation Join modification start loosing its advantage and the plain MOSWE algorithm is a better choice.

6. CONCLUSIONS

In this paper we defined the "Time Relaxed Spatiotemporal Trajectory Join" query and first manifested that the basic symbolic join algorithm is not an efficient solution for this problem. We then introduced two new heuristics that can drastically reduce query time for the TRSTJ query. The first approach is based on the notion of multiple origins that reduces effectively the number of false positives. An extensive experimental evaluation showed improvement in orders of magnitude. The second heuristic was based on the principle of "divide and conquer" and proved to be very efficient for situations where the memory resources are limited. As future work we plan to extend our techniques for the "best-match" trajectory join problem, that looks for the best match between two trajectories over their whole lifetimes. Another future extension of these techniques is the

use of different distance metrics. In the current paper we use L2 as the distance metric because of its simplicity. Other metrics may also be appropriate but need to be further examined.

7. REFERENCES

[1] Spatio-temporal Generators, http://www.cs.ucr.edu/~marioh/generators/index.html

[2] W. Aref, D. Barbará, Supporting electronic ink databases. Information Systems, 24(4):303–326, 1999.

[3] P. Bakalov, M. Hadjieleftheriou, E. Keogh, V.J. Tsotras. Efficient Trajectory Joins using Symbolic Representations. In Proc. of the International Conference on Mobile Data Management (MDM), 2005.

[4] T. Brinkhoff, H. P. Kriegel, B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In SIGMOD, pages 237–246, 1993.

[5] V. P. Chakka, A. Everspaugh, J. M. Patel. Indexing Large Trajectory Data Sets With SETI. In Proc. of Biennial Conference on Innovative Data Systems Research (CIDR), 2003.

[6] L. Chen and M. T. Özsu and V. Oria. Symbolic representation and retrieval of moving object trajectories. In Proc. of the ACM SIGMM international workshop on multimedia information retrieval, pages 227–234, 2004.

[7] H. D. Chon, D. Agrawal, A. El Abbadi. Storage and Retrieval of Moving Objects. In Proc. of the International Conference on Mobile Data Management (MDM), pages 173–184, 2001.

[8] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, M. Vazirgiannis. A foundation for representing and querying moving objects. ACM Transactions on Database Systems (TODS), 25(1):1–42, 2000.

[9] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, D. Gunopulos. Efficient Indexing of Spatiotemporal Objects. In Proc. of Extending Database Technology (EDBT), pages 251–268, 2002.

[10] Y. Huang, N. Jing, E. Rundensteiner. Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations. In VLDB, pages 396–405, 1997.

[11] E. J. Keogh, M. J. Pazzani. A Simple Dimensionality Reduction Technique for Fast Similarity Search in Large Time Series Databases. In Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, Current Issues and New Applications, pages 122–133, 2000.

[12] G. Kollios, V.J. Tsotras, D. Gunopulos, A. Delis, M. Hadjieleftheriou. Indexing Animated Objects Using Spatiotemporal Access Methods. IEEE Transactions on Knowledge and Data Engineering (TKDE), 13(5):758–777, 2001.

[13] N. Koudas, K. C. Sevcik. Size separation spatial join. In SIGMOD, pages 324–335, 1997.

[14] J. Lin, E. Keogh, S. Lonardi, B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In Proc. of ACM SIGMOD workshop on Research issues in data mining and knowledge discovery, pages 2–11, 2003.

[15] M.-L. Lo, C. V. Ravishankar. Spatial joins using seeded trees. In SIGMOD, pages 209–220, 1994.

[16] M. Nascimento, J. Silva. In Proc. of ACM Symp. on Applied Computing (SAC), 1998.

[17] D. Papadias, Y. Tao, J. Zhang, N. Mamoulis, Q. Shen, J. Sun. Indexing and Retrieval of Historical Aggregate Information about Moving Objects. IEEE Data Engineering Bulletin, 25(2), June 2002.

[18] D. Pfoser, C. S. Jensen, Y. Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In VLDB, pages 395–406, 2000.

[19] P. Rigaux, M. Scholl, A. Voisard. Spatial Databases With Application to GIS. Morgan Kaufman, 2001.

[20] J. Schiller, A. Voisard (eds.). Location-Based Services. Morgan Kaufmann, 2004.

[21] J. Shan, D. Zhang, B. Salzberg. On Spatial-Range Closest-Pair Query. In Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD), pages 252–269, 2003.

[22] S. Shekhar, S. Chawla. Spatial Databases: A Tour. Prentice Hall, 2003.

[23] A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. Modeling and Querying Moving Objects. In Proc. of International Conference on Data Engineering (ICDE), pages 422–432, 1997.

[24] Y. Tao, D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In VLDB, pages 431–440, 2001.

[25] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, E. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In Proc. of ACM Knowledge Discovery and Data Mining (SIGKDD), pages 216–225, 2003.

[26] M. Vlachos, G. Kollios, D. Gunopulos. Discovering similar multidimensional trajectories. In Proc. of International Conference on Data Engineering (ICDE), pages 673–684, 2002.

[27] B.-K. Yi, C. Faloutsos. Fast Time Sequence Indexing for Arbitrary Lp Norms. In VLDB, pages 385–394, 2000.

[28] H. Zhu, J. Su, O. H. Ibarra. Trajectory queries and octagons in moving object databases. In Proc. of Conference on Information and Knowledge Management (CIKM), pages 413–421, 2002.