Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates

Glenn S. Iwerks [∮‡†] Hanan Samet [‡*] Ken Smith [∮†]

[‡]Computer Science Department,
Center for Automation Research, and
Institute for Advanced Computer Studies
University of Maryland at College Park
{iwerks,hjs}@umiacs.umd.edu

[∮]The MITRE Corporation
7515 Colshire Dr.
McLean, Virginia 22102
{iwerks,kps}@mitre.org

Abstract

In recent years there has been an increasing interest in databases of moving objects where the motion and extent of objects are represented as a function of time. The focus of this paper is on the maintenance of continuous k-nearest neighbor (k-NN) queries on moving points when updates are allowed. Updates change the functions describing the motion of the points, causing pending events to change. Events are processed to keep the query result consistent as points move. It is shown that the cost of maintaining a continuous k-NN query result for moving points represented in this way can be significantly reduced with a modest increase in the number of events processed in the presence of updates. This is achieved by introducing a continuous within query to filter the number of objects that must be taken into account when maintaining a continuous k-NN query. This new approach is presented and compared with other recent work. Experimental results are presented showing the utility of this approach.

*This work was supported in part by the National Science Foundation under grants EIA-99-00268, IIS-00-86162, and EIA-00-91474.

†This work was supported in part by the National Institute for Mental Health and the National Science Foundation under NIH grant R01-MH64417-01.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

1 Introduction

The interaction of GIS and spatial databases has been a topic of research for many years (e.g., [11]). Until recently the primary focus of spatial database research has been on *static* spatial data which are updated infrequently. Examples include buildings, roads, and land use zones. Common types of queries include spatial join, nearest neighbor, and windowing. An example of such a query is one that seeks all the houses that are for sale within one kilometer of a grade school. Most spatial database research in the past focused on data storage, query processing, and display of static spatial data.

In recent years there has been an increasing interest in databases of objects that are in motion. Objects in *moving object* databases change location frequently. Some examples are vehicles, mobile networks, and weather systems. Query operators include both spatial and temporal operators. An example of such a query is one that seeks all the moving school buses currently within one mile of a school. The dynamic nature of the data presents new challenges in data storage, query processing, and display of the results.

This paper addresses the maintenance of continuous k-nearest neighbor (k-NN) queries in the presence of updates on moving points, where the motion of the points is represented as a function of time. The closest analogy to what is done in conventional databases is a materialized view [4]. The difference is that the query result may change as objects move in accordance with the definition of its associated function as time advances, independent of updates.

Our main contribution is a new approach to this problem called the Continuous Windowing k-NN algorithm (CW). This new approach is based on the observation that window queries are easier to maintain on moving objects than k-NN queries. The CW

algorithm filters objects to be considered as nearest neighbor candidates using a within query around the query point. If the within query selects at least k objects, then only those objects in the within query result need to be considered when computing the k-NN query. Using this approach, the cost of maintaining a continuous query result for moving objects can be greatly reduced when the moving object representations stored in the database are updated frequently during query maintenance.

Experimental results are presented showing the utility of our approach. Our primary metric for cost is in the number of disk accesses needed to compute and maintain a query. This is because accessing data on disk is several orders of magnitude slower than accessing data in memory.

Although the examples given in this paper are 1-dimensional (1D) and show static query objects, the techniques and algorithms are general and applicable to higher dimensions and moving query objects.

The rest of this paper is organized as follows. In Section 2 we give definitions and background on event driven moving object query processing. Previous work is reviewed in Section 3. In Section 4, we describe an extension to previous work to support the maintenance of k-NN query results in the presence of updates. In Section 5, we present our new approach to solving the problem. Performance issues of the different approaches are discussed in Section 6. Experimental results of these two approaches are compared in Section 7. Conclusions and future work are discussed in Section 8.

2 Background

In order to represent moving objects, we define a *kinematic object*[1] as a geometric object having a location or extent represented as a function of time. In particular, the *point kinematic object* (PKO) is an object class representing the motion of a moving point by the function $p(t) = \vec{x_0} + (t - t_0)\vec{v}$, where $\vec{x_0}$ is the start location, t_0 is the start time, and $\vec{v}$ its velocity vector. As shorthand, we use $\mathsf{pt}(x_0, v, t_0)$ to denote a PKO instance. We assume an object-relational database environment in which a PKO is an attribute in a relation r, called a *PKO attribute.* For simplicity, and without loss of generality, we consider relations having one PKO attribute. Each instance of a PKO attribute value is a PKO object instance, each with its own unique identifier. The instance of a PKO attribute for some tuple $\tau \in r$ is denoted $\mathsf{Pko}(\tau)$. The tuple in r containing some PKO p is denoted $\mathsf{Tuple}(p)$.

[1] Kinematics is the branch of mechanics that studies the motion of a body or a system of bodies without giving any consideration to its mass or the forces acting on it.

We make a distinction between two basic types of queries. First, the *within query* finds all the objects within a given distance of a query point. Second, the *k-nearest neighbor* query (k-NN) finds the k closest neighbors to a query point.

To support the maintenance of the results of these queries we distinguish between two basic types of events: *within events* (w-event), and *order change events* (oc-events). Events are used to maintain query results as time advances. Events are processed to keep the query result consistent as the points move.

A w-event occurs when a PKO moves to be at a given distance d to a query point. This can happen either while the PKO is moving closer to, or farther from the query point. A w-event instance is denoted as $\mathsf{w}(p, t)$ where p is the PKO object, and t is the time of the event. Figure 1 shows snapshots of a 1-dimensional PKO data set $\{a, b, c\}$ and a query point q at different points in time where $a = \mathsf{pt}(1, 1, 1)$, $b = \mathsf{pt}(3.5, 1, 1)$, and $c = \mathsf{pt}(6.5, -1, 1)$. Query point $q = \mathsf{pt}(5.5, 0, 1)$. A w-event, $\mathsf{w}(b, 2)$, takes place at time $t = 2$ when PKO b comes within distance $d = 1.5$ of query point q. It is crucial to remember that the query point and distance are part of the query, and not explicitly represented in the event notation.

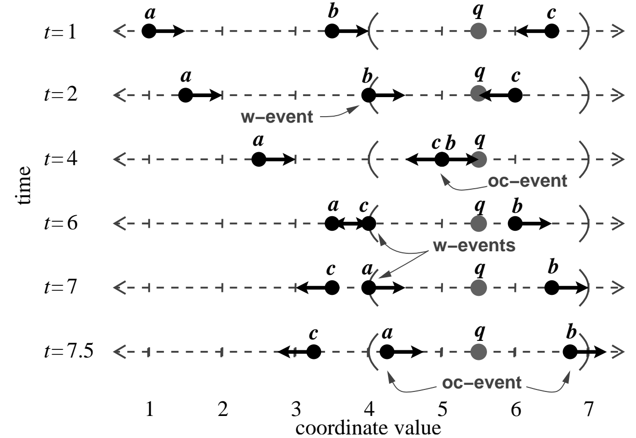


Figure 1: Example snapshots of 1D PKO attributes and events for time interval $1 \leq t \leq 7.5$. Arrow lengths indicate the distance traveled in one time unit.

An oc-event occurs when two PKO objects change order with respect to their distance to a query point. An oc-event instance is denoted as $\mathsf{oc}(p, t)$ where p is the point involved, and t is the time. Figure 1 shows the oc-event $\mathsf{oc}(b, 4)$, which corresponds to time $t = 4$ when object b is moving closer to q than another PKO c. In the example here, PKO c is the nearest neighbor prior to the event. In the case of a k-NN query, the kth neighbor is the other point involved in the event. It is important to note that this other point involved is part of the query result, and not explicitly represented

in the event notation. Likewise, the query point is part of the query and not explicitly represented in the event notation.

For some event e (either a w-event or an oc-event), $\mathsf{Pko}(e)$ denotes the PKO explicitly represented in that event (e.g., $\mathsf{Pko}(\mathsf{w}(p,t)) = p$). The time of an event is denoted as $\mathsf{Time}(e)$ (e.g., $\mathsf{Time}(\mathsf{w}(p,t)) = t$). If e is a null event (denoted $e = \emptyset$), then $\mathsf{Time}(e) = \infty$. Since a PKO instance is an object with a unique id, we can retrieve the tuple to which the PKO instance belongs. To indicate the tuple containing some PKO instance p we write $\mathsf{Tuple}(p)$. The Euclidean distance between PKO instances p and q at time t is $||p,q,t|| = |p(t),q(t)| = \sqrt{(q(t)-p(t))^2}$

```
     procedure SE_Within(r, q, d, t)
1.     Q ← ∅,  W ← ∅
2.     foreach tuple τ ∈ r do
3.       if ||Pko(τ), q, t|| < d then W ← W ∪ τ
4.       e ← next_w_event(Pko(τ), q, d, t)
5.       if Time(e) < ∞ then Q ← Q ∪ e
6.     end foreach
7.     while Q ≠ ∅ do
8.       e ← Pop(Q),    t ← Time(e)
9.       if W ∩ Tuple(Pko(e)) = ∅ then
10.        W ← W ∪ Tuple(Pko(e))
11.        e ← next_w_event(Pko(e), q, d, t)
12.        if Time(e) < ∞ then Q ← Q ∪ e
13.      else W ← W − Tuple(Pko(e))
14.    end while
```

Figure 2: $\mathsf{SE_Within}(r,q,d,t)$

Event-driven within query processing is performed by examining all within events in temporal order while updating the result appropriately[2]. Figure 2 gives a simple event-driven algorithm $\mathsf{SE_Within}()$ for maintaining a within query. For example, consider the 1D scenario in Figure 1, and a query to find all objects within distance $d = 1.5$ of q. Initially, relation r containing tuples with PKO attributes $\{a,b,c\}$[3] is scanned (lines 2–6) to find the initial result at time $t = 1$, $W = \{c\}$, and the next w-event for each point. The w-events are inserted into the priority Q, so that $Q = \{\mathsf{w}(b,2),\mathsf{w}(c,6),\mathsf{w}(a,7)\}$. Function $\mathsf{Pop}(Q)$ removes the next event from priority queue Q and returns it. Function $\mathsf{next_w_event}(p,q,d,t)$ returns the next event after time t when the PKO p will be at distance d from q, or it returns a null event with time stamp ∞ if no such event exists. This is a simple computation based on solving the equation $|p(time),q(time)| = d$ for *time*. For dimensionality greater than 1, this is a quadratic equation with a closed form solution. If the roots exist and are real numbers, then the next one greater than t is returned. Events are processed one-by-one (Figure 2, lines 7–14). If a PKO is moving closer to the query point then the w-event is called an *enter* event. If a PKO is moving farther away from the query point then the w-event is called an *exit* event. Line 9–12 process enter events adding the incoming object to the within result and computing the next exit event. Line 13 processes exit events removing the object from the within result. No new events are generated by exit events. Figure 3 shows a trace of the event processing portion of the algorithm (lines 7–14) up to time $t = 7$ for the 1D example in Figure 1 where $d = 1.5$.

line #	e	Q	W	t
8	$\mathsf{w}(b,2)$	$\{\mathsf{w}(c,6),\mathsf{w}(a,7)\}$	$\{c\}$	2
10	"	"	$\{b,c\}$	"
11	$\mathsf{w}(b,8)$	"	"	"
12	"	$\{\mathsf{w}(c,6),\mathsf{w}(a,7),\mathsf{w}(b,8)\}$	"	"
8	$\mathsf{w}(c,6)$	$\{\mathsf{w}(a,7),\mathsf{w}(b,8)\}$	"	6
13	"	"	$\{b\}$	"
8	$\mathsf{w}(a,7)$	$\{\mathsf{w}(b,8)\}$	"	7
10	"	"	$\{a,b\}$	"
11	$\mathsf{w}(a,13)$	"	"	"
12	"	$\{\mathsf{w}(b,8),\mathsf{w}(a,13)\}$	"	"

Figure 3: Trace of $\mathsf{SE_Within}()$ event processing for the 1D example in Figure 1 up through $t = 7$.

An event-driven k-NN query processing algorithm finds the soonest oc-event to occur in the future out of all possible oc-events. This is called the *nearest neighbor event* (nn-event) because it will cause the k-NN query result to change. A nn-event is an oc-event, but not every oc-event is a nn-event. Figure 4 outlines a simple event-driven algorithm to maintain a 1-NN query.

```
     procedure SE_Nearest_Neighbor(r, q, t)
1.     nn ← ∅,  done ← false
2.     foreach tuple τ ∈ r do
3.       if nn = ∅ ∨ ||Pko(τ), q, t|| < ||Pko(nn), q, t|| then
4.         nn ← τ
5.     end foreach
6.     while ¬done do
7.       e ← ∅
8.       foreach τ ∈ r ∧ τ ≠ nn do
9.         e' ← next_oc_event(Pko(τ), q, Pko(nn), t)
10.        if e = ∅ ∨ Time(e') < Time(e) then e ← e'
11.      end foreach
12.      if e ≠ ∅ then t ← Time(e),  nn ← Tuple(Pko(e))
13.      else done ← true
14.    end while
```

Figure 4: $\mathsf{SE_Nearest_Neighbor}(r,q,t)$

The algorithm first scans r to find the nearest neighbor *nn*. The algorithm then examines every object to find the next oc-event for that object. Function $\mathsf{next_oc_event}(p,q,nn,t)$ returns the next oc-event after time t with respect to query point q, and the nearest neighbor *nn*. If no such event exists, then

[2] There are cases when an object may only "touch" the event threshold and then move back (closer or farther) to its former state. Due to space limitations, these cases are not discussed here.

[3] For brevity only the PKO attributes of tuples are shown.

it returns a null event with time stamp ∞. This is a simple computation based on solving the equation $|p(time), q(time)| = |nn(time), q(time)|$ for *time*. For dimensionality greater than 1 this is a quadratic equation with a closed form solution. If the roots exist and are real numbers, then the next one greater than t is returned. When the next nn-event comes due, the algorithm again examines every object and computes their oc-events to find the next nn-event. Figure 5 shows a trace of the event processing portion of the algorithm (lines 6–14) up to time $t = 7.5$ for the 1D example in Figure 1.

Note that SE_Nearest_Neighbor() does not have a queue for events. This is because the nearest neighbor changes on each nn-event thereby rendering previously computed oc-events irrelevant. The asymptotic running time for SE_Nearest_Neighbor() is $\mathcal{O}(E_{nn} * N)$ where E_{nn} is the number of nn-events processed throughout the course of the query maintenance, and N is the cardinality of r. The asymptotic running time for SE_Within() is $\mathcal{O}(N + E_w)$ where E_w is the number of w-events processed throughout the course of the query maintenance.

line #	τ	e	e'	nn	t
7	-	$\emptyset$	-	c	1
8	a	"	"	"	"
9	"	"	oc(a, 6.5)	"	"
10	"	oc(a, 6.5)	"	"	"
8	b	"	"	"	"
9	"	"	oc(b, 4)	"	"
10	"	oc(b, 4)	"	"	"
12	"	"	"	b	4
7	"	$\emptyset$	"	"	"
8	a	"	"	"	"
9	"	"	oc(a, 7.5)	"	"
10	"	oc(a, 7.5)	"	"	"
8	c	"	"	"	"
9	"	"	oc(c, ∞)	"	"
12	"	"	"	a	7.5

Figure 5: Example trace of SE_Nearest_Neighbor() event processing (lines 6–14, Figure 4) for the 1D example in Figure 1 up to time $t = 7.5$.

These simple algorithms serve to illustrate the fundamental differences in processing nn-events vs. processing w-events. The oc-events from which the nn-event is chosen are dependent on the query result which changes when a nn-event occurs. This makes all previous oc-events computed with respect to the old query result irrelevant. This results in a need to compute new oc-events when the query result changes. On the other hand, pending w-events do not become irrelevant when the query result changes because w-events are independent of the query result. Any successful approach to maintaining continuous k-NN queries must have a way to reduce the number of oc-events considered for the next nn-event when nearest neighbor changes. One approach presented in previous work is to use an index to aggregate objects arranged in a hierarchy to reduce the number of objects considered each time the next nn-event is computed [13]. Another is to maintain a list of objects sorted by distance from the query point trading an increase in the number of events processed for a decrease in the number of objects considered per event [7].

3 Previous Work

Event processing of PKO data has been studied in domains such as simulation [3, 8], computational geometry [2], and moving object databases. Most work in moving object databases has focused on indexing and support of ad-hoc spatial queries [1, 10, 12, 15]. More recently, continuous spatial queries of kinematic data have been studied [7, 13]. However, to the best of our knowledge, no previous work reporting experimental results on maintaining continuous k-NN queries in the presence of updates on kinematic data sets has been reported.

3.1 Plane-Sweeping Technique

In [7], Mokhtar et. al. describe a method to maintain the k-NN query result on sets of PKO data over time. The algorithm starts by creating a list of PKO's sorted by their current distance from a query point. Events are then computed corresponding to the instances in time when any point will change its position on the list with its neighbor. These events are inserted in a priority queue sorted by time. If an object is updated, then any events on the queue involving that object are recomputed. The list of PKO's is updated as each event is processed in temporal order. The first k elements on the list form the k-NN result set.

The asymptotic size of the priority queue is $\mathcal{O}(n)$ where n is the number of moving points. The number of points that need to be examined when an event is processed is $\mathcal{O}(1)$ since only immediate neighbors on the list need to be examined. If n is large, it would be reasonable to assume that much of the event queue would reside on disk. The rate of events that need to be processed in this approach depends on the distribution and motion characteristics of the data set, but it would be easy to imagine cases were the number of events processed over a given amount of time would be much greater than for the algorithm presented in Figure 4 because an event is processed every time any two objects change order on the list and not just when an object changes order with the nearest neighbor. The performance behavior in practice of this algorithm remains unclear since the paper is

theoretical, and no implementation details or experimental results are presented.

3.2 TPR-tree

The Time Parameterized R-tree (TPR-tree) described in [10] by Saltenis et. al. is an index for kinematic data sets. It is a disk-based object hierarchy R-tree variant. In the R-tree, each node is stored in one disk page. Each node has an associated minimum bounding box (MBB). Leaf nodes contain the minimum bounding boxes (MBB) for the objects themselves. Each internal node has an MBB for each subtree bounding the objects in the subtree.

The bounding boxes (BB) in a TPR-tree need not be minimal. Each BB is moving hyper-rectangle specified by two PKO objects defining opposite corners of the BB. Each dimensional component of the velocity vector of one of the BB corner points is the minimum for all objects bounded by the box. Similarly, each dimensional component of the opposing point defining the BB is the maximum velocity vector component for all bounded objects. This ensures that all objects bounded by the box stay bounded as time progresses, but the BB rarely stays minimal. The TPR-tree insertion and deletion algorithms try to compensate for this lack of tight containment in a number of ways. One is that on each update, the BB is adjusted to be minimal at the time of the update. Another is that the algorithm tries to put objects moving in a similar manner (e.g., speed, direction), or to a similar destination, in the same leaf node. Even with these adjustments, the TPR-tree tends to degenerate as time progresses and must be rebuilt periodically. Saltenis et. al. use the TPR-tree to support ad-hoc spatio-temporal window queries.

3.3 TP KNN Algorithm

In [13], Tao and Papadias describe a method, called the time parameterized k-NN algorithm (TP KNN), to compute the k-nearest neighbors (k-NN) of a kinematic data set, and the nn-event. This algorithm works for both PKO objects and objects with kinematic bounding boxes. Tao and Papadias use the TPR-tree in their experiments, but assert that any object hierarchy index for kinematic data will work. To compute the initial k-NN result set, they evaluate the position of the bounding boxes in the index and kinematic data objects at the query start time. The initial result set is then found using established k-NN algorithms. They consider both a depth-first (DF) [9] and best-first (BF) [6] k-NN algorithm in their approach.

To find the k nearest neighbors, the DF algorithm examines each node starting at the root of the index tree. The bounding box of each subtree is placed on a local priority queue sorted by its distance from the query object. The first object on the queue is popped, and the process is repeated recursively. When the first leaf node is reached, the closest k objects in the leaf are put in a global array. The recursion then unwinds to the parent node where the next node in the local queue is popped off. If the next node on the queue is closer to the query object than the current k^{th} nearest neighbor then the algorithm is invoked recursively on that node. If it is not closer, then the rest of the nodes on the local queue may be ignored. This is because they can not contain any objects closer to the query object than the current k^{th} nearest neighbor. The recursion continues to unwind until a closer node is found or the recursion exits. When subsequent leaf nodes are processed, each object in the leaf is compared to the current k^{th} nearest neighbor and added to the result set if they are closer. This results in a new even closer k^{th} nearest neighbor. The distance to the kth nearest neighbor defaults to infinity until the k neighbor candidates are initially found. The BF algorithm is similar to the DF algorithm except that it uses a global priority queue to order the index nodes by distance from the query point. Thus the next closest node is always examined first regardless of which node was just processed.

In the TP KNN algorithm, once the k nearest neighbors are found, the next nn-event is found. This is the so called TP component of the approach described in [13]. Finding the initial k-NN result is called the non-TP component. The TP component is obtained using the same neighbor finding methods as the non-TP component, but now instead of ordering nodes and objects by distance from the query point, they are ordered by their next oc-event time (referred to as their influence time in [13]). The oc-event times of the aggregate kinematic BB's will always be less than or equal to the oc-event times of the objects that they bound. In this way, the algorithm avoids examining every kinematic object when trying to compute the nn-event. Tao and Papadias support continuous k-NN queries by invoking the TP component each time the nn-event is processed. Their approach does not support updates during query processing.

4 Extending TP KNN for Updates

In this section we extend the continuous TP KNN algorithm presented in [13] to support updates during processing of the query, termed the *extended TP* (ETP) algorithm. Later in Section 5 we present a new contribution of our own, the Continuous Windowing KNN algorithm (CW) algorithm, and compare it to the ETP algorithm. The notation and functions de-

fined in Section 2 are used in this section.

One simple approach to extend the continuous TP KNN algorithm is to invoke the non-TP and TP components of the algorithm whenever an update occurs. A more efficient approach, Extended_TP_Knn() (ETP) given in Figure 6, is based on the observation that the TP and non-TP components are only needed if an update affects points in the k-NN result set, affects points involved in the nn-event, or introduces an oc-event earlier than the current nn-event.

```
     procedure Extended_TP_Knn(r, q, k, t)
 1.    tpr ← TP_Build_Index(r)
 2.    TP_Compute_Knn_Result(tpr, q, k, t, K)
 3.    e ← TP_Find_Next_Knn_Evt(tpr, q, t, K)
 4.    while ¬ done do
 5.      u ← Next_Update(r)
 6.      while Time(e) < Time(u) do
 7.        if Tuple(Pko(e)) ∩ K = ∅ then
 8.          K ← (K ∪ Tuple(Pko(e))) − Kth(K)
 9.        e ← TP_Find_Next_Knn_Evt(tpr, q, Time(e), K)
10.      end while
11.      if update u is an insertion then
12.        e ← TP_Process_Insert(u, tpr, q, e, K)
13.      else
14.        e ← TP_Process_Delete(u, tpr, q, e, K)
15.    end while
```

Figure 6: Extended_TP_Knn() (ETP)

In Figure 6, parameter r is a relation with schema R, q is the PKO query point, k is the number of neighbors to find, and t is the current time. Variable K is the query result, *tpr* is the TPR-tree index, u is an update tuple where Time(u) denotes the time of the update, and e is the nn-event. Function Next_Update(r) returns the next scheduled update u for relation r. If no updates are currently pending, then it returns a null update with time stamp ∞. For simplicity, we assume there is always at least one update or event pending; otherwise, the process can simply sleep until one is available. Function Kth(K) returns the kth nearest neighbor of K at the time K was last modified.

The algorithm starts by using TP_Build_Index() to build the TPR-tree (line 1). It then applies the non-TP component of the TP KNN algorithm (described in [13] and Section 3.3) to find the initial k-NN query result in a call to TP_Compute_Knn_Result() (line 2). A call to TP_Find_Next_Knn_Evt() computes the nn-event using the TP component of the TP KNN algorithm. If no such event exists, then TP_Find_Next_Knn_Evt() returns a null event with time stamp ∞. The main loop then processes updates and events in the order of the time of their arrival until some arbitrary termination condition is met (e.g., time limit, event limit, update limit). For each event processed, the result set is adjusted if needed, and then the earliest nn-event is computed using the TP component of the TP KNN algorithm again (lines 7–9).

```
    function TP_Process_Insert(u, tpr, q, e, K)
1.    TPR_Insert(tpr, Pko(u))
2.    if ||Pko(u), q, Time(u)|| < ||Pko(Kth(K)), q, Time(u)||
3.    then
4.      K ← (K ∪ u) − Kth(K)
5.      e ← TP_Find_Next_Knn_Evt(tpr, q, Time(u), K)
6.    else
7.      e' ← next_oc_event(Pko(u), q, Pko(Kth(K)), Time(u))
8.      if Time(e') < Time(e) then e ← e'
9.    return e
```

Figure 7: TP_Process_Insert()

TP_Process_Insert() handles the insertion of tuples (see Figure 7). TPR_Insert() inserts the PKO attribute into the TPR-tree index (line 1). If u is closer to the query point than the current k^{th} nearest neighbor then the current k^{th} nearest neighbor is removed, u is added to K, and then the new nn-event is found (lines 2–5). If u is not closer to q than the current k^{th} nearest neighbor, then the next oc-event for only u is computed using next_oc_event(). If the oc-event for u is earlier in time than the current nn-event then it becomes the new nn-event (lines 6–8).

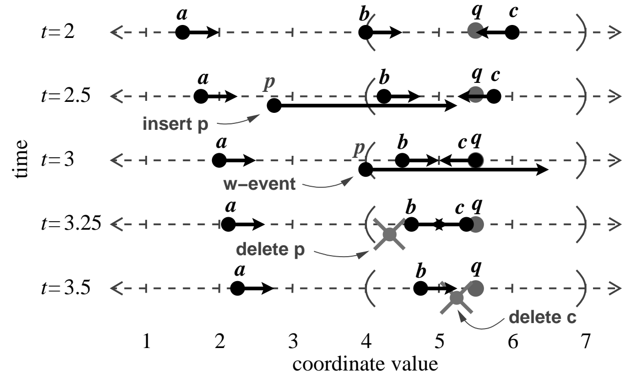


Figure 8: Example snapshots in time of 1D PKO's, events, and updates up to time $t = 3.5$. Arrow length indicates distance traveled in one unit of time.

Figure 8 is a 1D example with updates. Columns 1 through 4 in Figure 9 show how updates given in the example affect pending events for the ETP algorithm.

t	u	nn	e (ETP)	Q (CW)
2	-	c	oc(b, 4)	{oc(b, 4), w(c, 6), w(a, 7), w(b, 8)}
2.5	insert p	c	oc(p, 3.5)	{w(p, 3), oc(b, 4), w(c, 6), w(a, 7), w(b, 8)}
3.0	-	c	oc(p, 3.5)	{oc(p, 3.5), w(p, 4.2), w(c, 6), w(a, 7), w(b, 8)}
3.25	delete p	c	oc(b, 4)	{oc(b, 4), w(c, 6), w(a, 7), w(b, 8)}
3.5	delete c	b	oc(a, 7.5)	{w(a, 7), w(b, 8)}

Figure 9: Event trace for the example in Figure 8.

In Figure 8 a tuple with PKO attribute p = pt(2.75, 2.5, 2.5) is inserted into relation r. This is handled in lines 6–8 of Figure 7 resulting in a change

of the nn-event (Figure 9, row 2, column 4).

```
   function TP_Process_Delete(u, tpr, q, e, K)
1.  TPR_Remove(tpr,Pko(u))
2.  if K ∩ u ≠ ∅ then
3.     TP_Compute_Knn_Result(tpr, q, |K|, Time(u), K)
4.     e ← TP_Find_Next_Knn_Evt(tpr, q, Time(u), K)
5.  else if Tuple(Pko(e)) = u then
6.     e ← TP_Find_Next_Knn_Evt(tpr, q, Time(u), K)
7.  return e
```

Figure 10: TP_Process_Delete()

TP_Process_Delete() handles deletion of tuples (see Figure 10). TPR_Remove() deletes the PKO attribute from the TPR-tree index (line 1). If u is part of the current query result, then the query result and nn-event must be recomputed from scratch using both the non-TP and TP components of the TP KNN algorithm (lines 2–4). If u is not part of the result but is involved in the current nn-event then a new nn-event is found using the TP component of the TP KNN algorithm (lines 5–6). For example, at time $t = 3.25$ in Figure 8, p is deleted. The TP component of the TP KNN algorithm is used (lines 5–6 in Figure 10) in finding the next nn-event (Figure 9, row 4, column 4). The deletion of the nearest neighbor c at time $t = 3.5$ in the example of Figure 8 requires that both the TP and non-TP components of the TP KNN algorithm be used (lines 2–4 Figure 10).

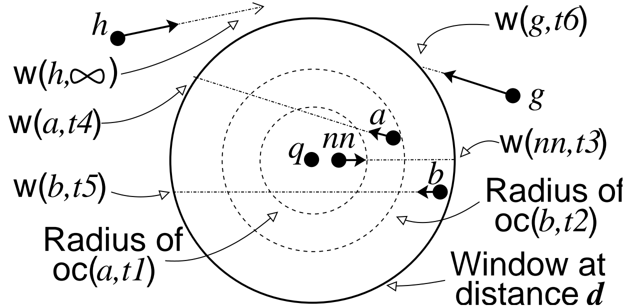


Figure 11: Example illustrating the CW approach. Only oc-events involving PKO's a and b in the window indicated by the outer circle are considered for the next nn-event with respect to nearest neighbor nn and query PKO q. PKO's g and h are not considered for the next nn-event until they enter the window.

5 Continuous Windowing KNN (CW)

The Continuous Windowing KNN algorithm (CW) is our main contribution. It is motivated by the observation that w-events are fundamentally less expensive to process than oc-events (see Section 2). The basic approach of the CW algorithm is to filter the number of moving points that need be considered when processing oc-events by maintaining a within query around the same query point containing at least k points in the within query result. Figure 11 is an example where the outer circle is the within window. Only PKO's inside the window are considered in searching for the next nn-event. Conceptually, the data set is divided into two subsets of points: those that are close to the query point, and those that are far away. Although overall this introduces more events than just those that result in a change to the k-NN query result, it makes processing the oc-events cheaper. For the algorithm presented below we assume that the query point is moving, but not updated. Section 6 discusses updates to the query point.

```
    function Continuous_Windowing_Knn(r, q, k, c, d, t)
1.   CW_Load(r, q, d, t, W, Q)
2.   CW_Compute_Knn_Result(q, k, t, W, K, Q)
3.   u ← Next_Update(r),  e ← Pop(Q)
4.   while ¬ done do
5.     if (|W| < k) ∨ (c * k < |W|) then return false
6.     if Time(u) < Time(e) then
7.       CW_Process_Update(u, q, k, d, W, K, Q)
8.       u ← Next_Update(r)
9.     else
10.      if e is a w-event then
11.        CW_Process_Within_Evt(e, q, k, d, W, K, Q)
12.      else CW_Process_Nn_Evt(e, q, k, d, W, K, Q)
13.      e ← Pop(Q)
14.  end while
15.  return true
```

Figure 12: Continuous_Windowing_Knn() (CW)

The CW algorithm is shown in Figure 12. All notation, variables and functions are as described in the previous sections unless otherwise specified. Parameter $c > 1$ is a small number. Variable e can be either a w-event or an oc-event.

```
   procedure CW_Compute_Knn_Result(q, k, t, W, K, Q)
1.  l ← list of all τ ∈ W sorted by ||Pko(τ), q, t||
2.  K ← list of first k elements of l
3.  min_e ← ∅      /* Time(min_e = ∅) = ∞ */
4.  foreach tuple τ ∈ W ∧ τ ≠ Kth(K) do
5.    e ← next_oc_event(Pko(τ), q,Pko(Kth(K)), t)
6.    if Time(e) < Time(min_e) then min_e ← e
7.  end foreach
8.  if Time(min_e) < ∞ then Q ← Q ∪ min_e
```

Figure 13: CW_Compute_Knn_Result()

The algorithm starts by computing a within query result W for distance d from PKO q. The body of CW_Load() is identical to lines 1–6 of Figure 2 which scans relation r to find all PKO's within distance d of query point q at time t and stores them in W. It also computes all future w-events and places them on priority queue Q. The k-NN result K is computed from the contents of W, not r, by CW_Compute_Knn_Result() (described below). The main loop then processes updates and events in temporal order until the constraint $(|W| \geq k) \wedge (c*k \geq |W|)$ fails, or some arbitrary termination condition is met (e.g., time limit, event limit,

update limit).

```
    procedure CW_Process_Update(u, q, k, d, W, K, Q)
1.  if ||Pko(u), q, Time(u)|| < d then
2.    if u is insert then W ← W ∪ u
3.    else W ← W − u
4.    CW_Update_Knn_Result(q, k, Time(u), W, K, Q)
5.  e ← next_w_event(Pko(u), q, d, Time(u))
6.  if Time(e) < ∞ then
7.    if u is insert then Q ← Q ∪ e
8.    else Q ← Q − e
```

Figure 14: CW_Process_Update()

Parameter d is initially chosen so that $k \leq |W| \leq c * k$ where $c > 1$ is a small number such that $c * k$ is the upper bound on the number of objects that may comfortably fit in main memory. If this constraint fails as the objects move, then the algorithm will not have enough information (i.e., $|W| < k$), or enough main memory (i.e., $c * k < |W|$) to maintain K. In this case, the algorithm must exit and restart at the time it left off with an adjusted value for d. The query result maintained up to the time of constraint failure remains valid.

CW_Compute_Knn_Result(), shown in Figure 13, computes K from W. By our assumption that W fits in main memory, this can be achieved without any disk accesses. All tuples in W are sorted by the distance of their PKO attribute from q at time t. The set K consists of the first k elements of the list[4]. Procedure CW_Compute_Knn_Result() also inserts the next nn-event in Q. This is done by computing all oc-events for the tuples in W and inserting the next oc-event to occur (the nn-event) into Q if it exists.

CW_Process_Update() is invoked when r is updated (see Figure 14). If u is within distance d of q at update time, then two things happen. First, W is updated appropriately (lines 2–3). Second, CW_Update_Knn_Result() in line 4 updates the k-NN results set and any pending nn-event on Q if needed. Finally, any w-event associated with tuple u is calculated and added to Q, or removed from Q as needed (lines 5–8).

```
    procedure CW_Update_Knn_Result(q, k, t, W, K, Q)
1.  K ← ∅
2.  Remove any pending nn-event from Q
3.  CW_Compute_Knn_Result(q, k, t, W, K, Q)
```

Figure 15: CW_Update_Knn_Result()

A very simple solution for CW_Update_Knn_Result() is shown in Figure 15. If CPU time is a factor, then a more efficient case-by-case approach may be needed. Since we assume that W and K will fit in main memory, the only impact on the number of disk accesses occurs when modifying Q to change the nn-event.

[4]For simplicity, and brevity, we do not consider the case when two PKO attributes are exactly at the same distance from q at time t

For example, consider Figure 8. Columns 1 through 3, and 5 of Figure 9 show how Q changes for each time step of the example. At $t = 2.5$ a tuple with PKO attribute $p = \mathsf{pt}(2.75, 2.5, 2.5)$ is inserted into relation r. Location $p(2.5)$ is farther from q than $d = 1.5$ so a new w-event $\mathsf{w}(p, 3)$ is added to the priority queue Q (handled by lines 5–8 of Figure 14). At time $t = 3$ the w-event is processed and the new nn-event $e = \mathsf{oc}(p, 3.5)$ is inserted into the queue (lines 10–11 of Figure 12). At time $t = 3.25$ p is deleted, and the new nn-event, $\mathsf{oc}(b, 4)$, is computed by examining the elements of $W = \{b, c\}$ (all lines of Figure 14). At time $t = 3.5$ the nearest neighbor c is deleted (all lines of Figure 14).

```
    procedure CW_Process_Within_Evt(e, q, k, d, W, K, Q)
1.  if W ∩ Tuple(Pko(e)) = ∅ then
2.    W ← W ∪ Tuple(Pko(e))
3.    Q ← Q ∪ next_w_event(Pko(e), q, d, Time(e))
4.  else
5.    W ← W − Tuple(Pko(e))
6.  CW_Update_Knn_Result(q, k, Time(e), W, K, Q)
```

Figure 16: CW_Process_Within_Evt()

CW_Process_Within_Evt() in Figure 16 is invoked on w-events. An *enter* event is a w-event where Pko(Tuple(e)) is moving closer to the query point. An *exit* event is when it is moving farther away. For an enter event, Tuple(e) is added to W and the subsequent exit event is calculated and inserted in Q (lines 2–3). Time $t = 3$ of Figure 8 shows an example enter event. In the example, the event generates a new nn-event $e = \mathsf{oc}(p, 3.5)$. On an exit event, Tuple(e) is simply removed from W (line 5). Any change in W could also change the k-NN result, or the nn-event. The result K and the associated nn-event are reexamined and changed if needed in the call to CW_Update_Knn_Result() (line 6).

CW_Process_Nn_Evt(), invoked in line 12 of Figure 12, handles nn-events. Again, since K and W both fit in memory, and K is calculated using W, not r, processing nn-events does not induce any new disk accesses except for a possible update to Q. CW_Process_Nn_Evt() could simply be a call to CW_Update_Knn_Result() (Figure 15). For brevity, these algorithms do not show how to handle cases when events or updates happen at exactly the same time.

6 Performance Issues

Analysis of algorithms for kinematic data is difficult without making many simplifying assumptions. Performance is dependent on many factors such as data set size, location distribution, velocity distribution, distribution of updates among tuples, and update fre-

quency distribution. Rather than attempting a rigorous analysis on an overly constrained subset of these factors, this section discusses some key performance issues of the ETP, CW, and Plane-Sweeping technique (PS) (described in Section 3.1), and how these factors play a part in the performance of each algorithm.

We assume that for large data sets, the majority of the data is stored on disk. Accesses to disk are orders of magnitude slower than memory, so cost is measured in number of disk accesses. For the sake of discussion, assume that all PKO data and query objects share the same location, velocity, and update rate distributions. Ignoring esoteric cases, assume that all points are moving relative to the query point, and that they are not all moving in the same direction and at the same speed. Note that there are no implementation details for the PS method presented in [7] so we need to make assumptions for this approach in order to analyze it. PS creates a sorted list L_{PS} of all PKO objects by distance to the query point, and has an event queue Q_{PS}. Let us assume an implementation using B-tree variants for both L_{PS}, and Q_{PS} to support efficient insertions and deletions. The CW event priority queue Q_{CW} is implemented using a B+-tree variant (see Section 7).

Initial Build: All three methods require an initial scan of some relation r. ETP scans r to build the TPR-tree index. CW scans the relation to find the query result and pending w-events. PS creates a list sorted by distance.

Data Structure Size: Let n be the size of the PKO data set. The asymptotic upper bound for the TPR-tree, L_{PS} (ignoring Q_{PS} for now), and Q_{CW} data structures is $\mathcal{O}(n)$. The lower bounds for each data structure are not the same. The entire data set must be inserted into the TPR-tree and L_{PS} giving a lower bound of $\Omega(n)$. The best case for CW is when no objects outside the within query result will enter the within query window in the future. In this case, the only objects involved in events in Q_{CW} are those in the within query result giving a lower bound of $\Omega(|W|)$. Given our assumption that $|W| \ll n$, it is likely that the size of Q_{CW} will be much smaller than the data structures for the other approaches.

Rebuilds: Rebuilding these structures from scratch may be required on occasion. Let UI be the average time period between two updates for a single object. By experimentation, Saltenis et. al. determined the TPR-tree performance degrades after time UI because almost all the entries have been updated by that time causing the index to degenerate due to increasing overlap of the index nodes. They conclude the TPR-tree should be rebuilt when time UI is reached. The PS priority queue Q_{PS} needs to be rebuilt whenever the query point is updated because all the events on the queue will no longer be valid. The expected time between updates to the query point is also UI if we assume the same update rate distribution for the query point as the rest of the data set. A rebuild is also needed for CW when the query point is updated for the same reason. A rebuild is also needed in the CW approach if the constraints on $|W|$ fail. The failure rate for these constraints depends heavily on the characteristics of the data and the method used to determine d. At the least, we would expect the CW method to rebuild more often than the other methods.

Number of Events: Only the nn-events are processed in the ETP approach. This makes ETP optimal in the number of events processed throughout the course of a query. There is only one event pending at any one time. CW processes additional w-events. The number of w-events over the course of a query, or on the event queue at any one time, depends on the selectivity of the within window and the motion characteristics of the data. PS processes an oc-event every time a neighbor changes position in L_{PS}. This includes the nn-event. The number of events on Q_{PS} at any one time is $\mathcal{O}(n)$. Since these events are when PKO's change order in L_{PS}, it is easy to imagine cases when the distance between neighbors on the list are small and thus many of these events on the queue will be imminent (e.g., points moving with different speeds and directions). In such cases, many more events would be processed over the course of a query than what the CW method would require.

Cost of Events: The cost of processing each event for each method is not the same. For the PS method it is only necessary to examine the immediate neighbors of objects that switch order on L_{PS} to find the next time they will switch order with their new neighbors. Assuming a B-tree structure for L_{PS} gives a cost of at most $\mathcal{O}(\log n)$ to find the neighbors. The cost of event updates in Q_{PS} is also $\mathcal{O}(\log n)$. The CW approach is even cheaper requiring no disk accesses to examine other objects when either a w-event or oc-event is processed because all the objects that need to be examined are already in main memory. The only cost is in updating the event queue which is $\mathcal{O}(\log n)$. In the ETP approach, the cost of processing an event is $\mathcal{O}(\log n)$ for the depth first (DF) TP component, and $\mathcal{O}(n)$ for the best first (BF) TP component of the algorithm. The worst case for BF happens when all objects are at the same distance from the query point. In practice this is unlikely in low dimensional data sets, and experiments show BF and DF perform similarly on uniformly distributed data. The ETP method has no event queue. The entire cost of the ETP method lies in the TPR-tree operations.

7 Experiments

This section presents experimental results comparing the ETP (Section 4) and CW (Section 5) algorithms. Due to time and space constraints, we did not implement the PS algorithm since the approach is theoretical and no implementation details were given in [7]. In any case, by our analysis, it appears that the PS data structures will be large, $\mathcal{O}(n)$. It also appears that the frequency of events would likely be higher than the CW or ETP approaches, since the differences in distances between objects on the list will be small, especially in the case of uniform data.

In our experiments, we first replicate the results obtained by Tao and Papadias in [13] for their continuous version of the TP KNN algorithm without updates and compare this to the CW algorithm (Figure 17(a)). We then compare the the ETP (both TP BF and TP DF variants) and CW algorithms while updating the data set during query maintenance for different values of various parameters such as update rate, distribution, and values of k.

We use code provided by Saltenis et. al. from their original implementation of the TPR-tree [10][5]. This was built on the GiST [5] code version 0.9beta1. We extended this with DF and BF algorithms for the TP and non-TP components of the TP KNN algorithm.

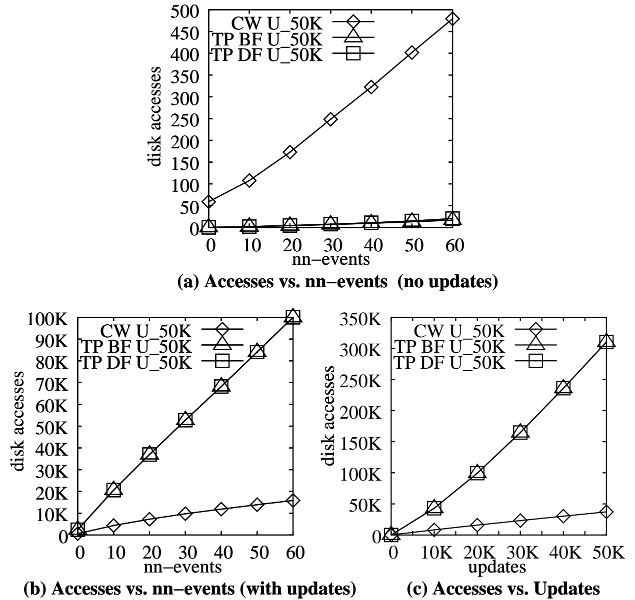
To implement the CW priority queue, we use a B+-tree variant of a priority search tree called the *Event B-tree* (EB-tree). In our implementation, every PKO has a unique id, or key. The priority queue is a B+-tree ordered by key to support efficient insertions and deletions of events. In addition to propagating the min-max key up the B+-tree, the earliest event time of all events in each subtree is also propagated up to the root. The earliest event in the tree is found by following the minimum event time down the branches of the tree to the leaf in which it is stored. This is implemented by extending the same GiST code used by Saltenis et. al.

To generate data, we use the data generator described in [10] modified slightly to generate normal distributions in addition to the uniform distribution. This code was also provided by Saltenis et. al. and used in their original experiments for the TPR-tree.

For comparability we attempt to reproduce the data and parameters as closely as possible to those experiments reported by Saltenis et. al. in [10] (TPR-tree with updates), and Tao and Papadias in [13] (TP KNN algorithm). When we borrow experimental parameters from other work, we indicate where it came from with the paper reference.

In our experiments, we use 2D data sets [10, 13]

[5] A special thanks to Saltenis et. al. and Tao et. al. for making their code available for use and study.



(a) Accesses vs. nn–events (no updates)

(b) Accesses vs. nn–events (with updates)

(c) Accesses vs. Updates

Figure 17: Experiment results

with uniform [10, 13] and normal [13] distributions in a 1000x1000 world [10] with an average velocity uniformly distributed in (0,3] [10]. Our simulations run for 60 time units [10]. Let UI be the average interval between two updates for a single object. We use a value of $UI = 60$ [10] unless otherwise specified. The time interval between successive updates is uniformly distributed between 0 and $2 * UI$ [10]. Uniform data sets of size n are denoted U_n. The normal distributions have a mean location of (500,500). The standard deviation and data set size for normal distributions are specified in the figures. The query point was selected at random from the initial data set, but is not updated during a simulation run. Parameter $k = 10$ [13] unless otherwise specified. The within distance $d = 50$ for the within portion of the CW algorithm in all cases.

Disk pages are 1024 bytes [13]. The disk cache is 50 pages [10, 13] using a least recently used (LRU) [10, 13] replacement policy for both methods. A TPR-tree leaf node holds 50 2D PKO objects (PKO function coefficients, object id), and an internal node holds up to 28 entries (kinematic rectangle, node id) [10]. A leaf node of the EB-tree holds 24 events (event, PKO function coefficients, object id, int {not used}) for 2D PKO objects, and an internal node holds up to 66 entries (event, object id, node id).

Updates are performed by deleting the old value and inserting the new value [10]. No new objects are introduced or removed after the initial load. Disk accesses are computed as an average over 200 data sets [10, 13] and k-NN queries (one query per data set). The ETP and CW methods were run on the same data sets and queries.

Figures 17 and 18 show the cost to maintain queries

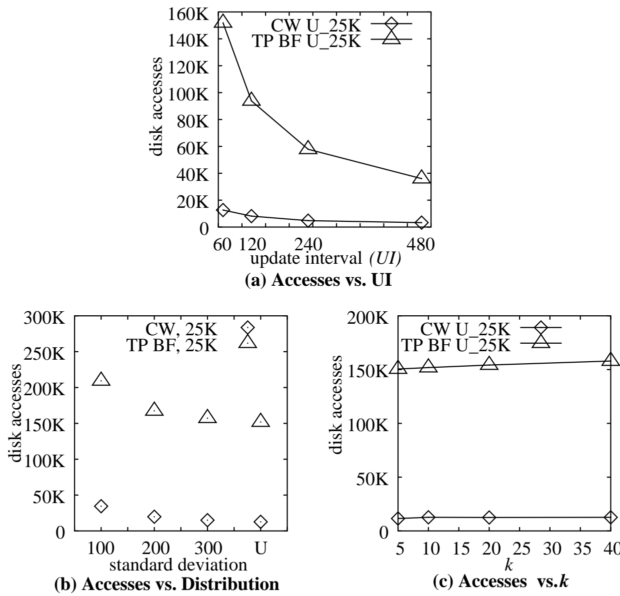


Figure 18: Experiment results

after the initial build and query result computation. The disk cache is flushed after the initial build before the initial query result is computed. Figure 17(a) shows disk accesses vs. nn-events without updates. ETP outperforms CW by over two orders of magnitude when there are no updates. This is because the ETP algorithm does not have to process w-events, and the DF and BF k-NN algorithms exhibit better locality in disk accesses by making good use of the cache. Accesses to the EB-tree are more random. As reported in [13], the BF method performs slightly better than the DF method. Figure 17(b) shows disk accesses vs. nn-events with updates. In this case, updates to the TPR-tree increase the number of disk accesses performed by the ETP algorithm. Figures 17(a) and 17(b) only show up to 60 nn-events because not all experiments had the same number of nn-events. The stopping condition for our experiments was based on elapsed time, not the number of events. All experiments did have at least 60 nn-events, so averaging disk accesses up to 60 nn-events is meaningful. Figure 17(c) shows disk accesses vs. number of updates. Performance in this figure is similar to Figure 17(b). This is because both the expected update rate and event rate are uniformly distributed. CW outperforms ETP by an nearly an order of magnitude when there are updates in our experiments. All three figures seem to exhibit near linear growth in any case.

We show only the TP BF variant of the ETP algorithm in the remainder of the experiments, since the TP BF and TP DF performance was similar for updates. Figure 18(a) shows disk accesses vs. UI. Even for a long average update interval, $UI = 480$, the CW approach outperforms ETP by an order of magnitude. The graph suggests that even a very small number of updates significantly degrades the ETP algorithm's performance. Figure 18(b) shows disk accesses vs. distribution for a data set size of 25000 objects. From left to right, the first three data sets are normally distributed with standard deviations of 100, 200, and 300, respectively, while the last distribution is uniform. In the CW algorithm, data with a smaller standard deviation was clustered leading to larger within results and thus a larger event queue using the constant within distance of $d = 50$. Similar performance is exhibited by the ETP algorithm for a different reason. Clustering of the data causes the BF and DF k-NN algorithms to search more subtrees of the TPR-tree index before the search can terminate. This is because more nodes are at similar distances (or have similar oc-event times) from the query point. This means fewer nodes are pruned in the search. Better means for determining d may improve the performance of the CW algorithm in the case of non-uniform data On the other hand, there is no obvious means of improving the ETP performance in this case. Figure 18(c) shows that the performance of both methods to be insensitive to different k values.

Figure 19(a) demonstrates that the total number of events processed using the ETP method is less than for the CW method. In our experiments, the CW method outperforms the ETP method in the total number of disk accesses in spite of an increased number of events processed when data is updated during query maintenance. Figure 19(b) shows the linear growth in the number of disk accesses for the initial build and query computation vs. data set size demonstrating that the CW method is cheaper to build. The size of the CW event queue is smaller than the TPR-tree used by the ETP method as shown in Figure 19(c).

The experimental results are promising. They help show that fewer disk accesses can be obtained when updates are allowed during query maintenance using the CW method vs. the ETP approach. The degradation of the ETP algorithm is primarily due to the increased overhead needed to maintain the TPR-tree when updates occur. Our experiments also show the cost of the initial build of the data structure is cheaper for CW in the case of uniform data distributions.

8 Conclusion

The CW method outperformed the ETP method when updates occur during query maintenance. In our experiments, we chose a fixed d suitable for the data sets; however, a good choice for d depends on the selectivity characteristics of the data. The next step in our research is to examine different approaches for determining an appropriate d for a given data set. One possible

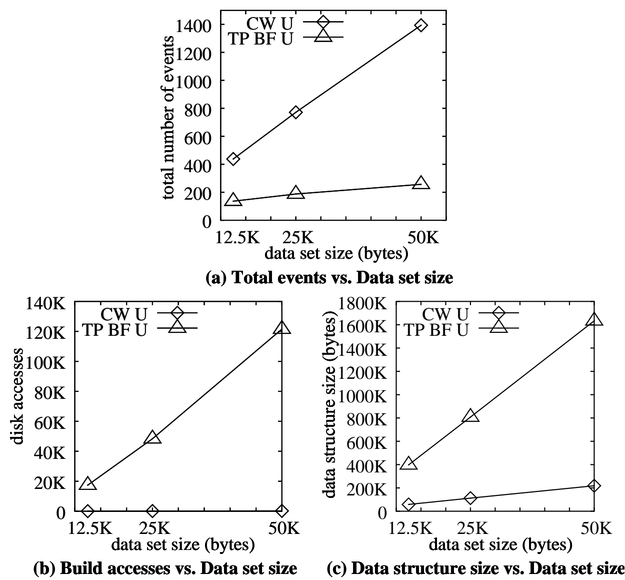


Figure 19: Experiment results

approach is to use selectivity estimation techniques for moving objects such as those described in [14]. This introduces an extra overhead cost of maintaining additional supporting data structures such as histograms. Another promising approach is to use a simple heuristic to handle the case when the selectivity constraint is violated. For example, if less than k objects are selected, then increase d by some factor and recompute. If the number of objects selected becomes too large, then decrease d accordingly and recompute. This also introduces extra overhead to rebuild the query from scratch when a constraint exception occurs. There is also no reason why d can not vary as a function of time to handle non-uniform data sets. Comparison of these and other techniques for determining d is the next step for future work.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems*, pages 175–186, Dallas, TX, May 2000.

[2] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 747–756, New Orleans, LA, January 1997.

[3] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[4] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD Conference*, pages 157–166, Washington, D.C., May 1993.

[5] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 562–573, Zurich, Switzerland, September 1995.

[6] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. (Also University of Maryland Computer Science TR–3919).

[7] H. Mokhtar, J. Su, and O. Ibarra. On moving object queries. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems*, pages 188–198, Madison, WI, June 2002.

[8] Standards Committee on Interactive Simulation (SCIS). *IEEE Std 1278.1-1995*. IEEE Computer Society, USA, March 1996.

[9] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference*, pages 71–79, San Jose, CA, May 1995.

[10] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the ACM SIGMOD Conference*, pages 331–342, Dallas, TX, May 2000.

[11] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[12] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings of the 13th IEEE Conference on Data Engineering (ICDE)*, pages 422–432, Birmingham, U.K., April 1997.

[13] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *Proceedings of the ACM SIGMOD Conference*, pages 334–345, Madison, WI, June 2002.

[14] Y. Tao, J. Sun, and D. Papadias. Selectivity estimation for predictive spatio-temporal queries. In *Proceedings of 19th IEEE International Conference on Data Engineering (ICDE)*, pages 417–428, Bangalore, India, March 2003.

[15] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.