Proximity Relation Processing With Evolving Environment

Zhengdao Xu and Hans-Arno Jacobsen
Department of Computer Science and
Department of Electrical and Computer Engineering,
University of Toronto
10 King's College Road, Toronto, Ontario, Canada, M5S 3G4
zhengdao@cs.toronto.edu, jacobsen@eecg.toronto.edu

March 17, 2007

Abstract

An important problem for many location-based applications is the continuous evaluation of proximity relations among moving objects. These relations express whether a given set of objects is in a spatial constellation or in a spatial constellation relative to a given point of demarcation in the environment. The challenge lies in the continuous processing of large numbers of such relations as the location position information of the objects change. In this paper, we propose an adaptive location constraint indexing technique for solving this problem. The proposed indexing technique adapts well to the change of movement patterns of the mobile objects, stabilizes the system performance, and reduces the cost for continuously processing the proximity relations for both in-memory processing and for I/O-incurring environment.

1 Introduction

With the advances in wireless communication and location positioning technology [23], the potential for tracking, correlating, and filtering information about moving objects has greatly increased. For example, it has become possible to track the location of mobile users in a wireless network or in a building [23, 16], the discrete location of vehicles or packages in delivery [1] and even the movement of livestock or fish for environmental purposes [2]

An important problem in the context of applications that leverage this tracking potential is how to efficiently determine whether for any given number of sets of moving objects, the objects per set are *close to* one another or *close to* a given point of demarcation. We refer to this problem as the *location constraint matching problem*. In a wireless network, for example, an operator may want to offer alerting services that notify members of a group (e.g., a group of friends or family), if they are *close to* each

other (e.g., friend finder and buddy tracking application) or *close to* a designated point in the environment (e.g., the CN Tower in Toronto). In a goods and packages delivery chain, the operator may want to know if a set of packages routed to the same or similar destinations are *close to* one another and may benefit from common delivery to amortize delivery cost. In ad hoc networking, for example, given a set of mobile devices, one may want to know if they are *close to* each other to benefit from the proximity for resource sharing (e.g., content, communication, caching and computation).

To model the position correlation among a set of n moving objects and a set of moving objects and a point of demarcation, we introduce two types of constraints, the n-*body constraint* and the n-*body static constraint*. We refer to these constraints as *location constraints*, as they are defined over the location position of the objects. *close-to-relation* is only a special case of the correlation specified by the constraints. The general location constraints are formally defined as follows:

1. The n-*body constraint* is of the form $|{p_1}^t, {p_2}^t, ..., {p_n}^t|$ op d (the operator op is either $<$ or $>$). It is matched if the n moving objects, identified by, $p_1, p_2, ..., p_n$, can($<$) or can not ($>$) be enclosed by a sphere with diameter d, , at some time, t. p_i ($1 \leq i \leq n$) is the identifier of object i. In our notation ${p_i}^t$ is interpreted as the coordinate of object i at time t. d is referred to as the *alerting distance*.

2. The n-*body static constraint* is of the form $|A, {p_1}^t, {p_2}^t, ..., {p_n}^t|$ op d (the operator op is either $<$ or $>$), where A is the coordinate of some static point. It is matched if the n moving objects, identified by, $p_1, p_2, ... , p_n$, are within ($<$) or out of($>$) the given range d, of the static point A at some time, t.

The location constraint matching problem can then be stated as follows: Given a set of location constraints $C = \{c_1, c_2, ..., c_k\}$, which designate the desired location relationship among a set of m, possibly, moving objects $P = \{p_1, p_2, ..., p_m\}$, *continuously* determine all constraints c_i in C that are matched. The location constraints are continuous queries that once submitted to the system remain active until explicitly revoked. If we define the moving objects associated with constraint c_i as $P_{c_i}^t = \{{p_1}^t, {p_2}^t, ..., {p_n}^t\}$, the above constraints could be shorthanded as $|P_{c_i}^t|$ op d and $|A, P_{c_i}^t|$ op d, respectively. The definition above does not include the operator $=$, $\leq$ and $\geq$ because these operations can be expressed with the negation of operator $<$ and $>$. For example, $P_{c_i}^t \leq d$ is simply $\neg P_{c_i}^t > d$ and $P_{c_i}^t = d$ is $\neg(P_{c_i}^t > d) \wedge \neg(P_{c_i}^t < d)$.

close-to-relation corresponds to the constraints with the smaller than operator($<$). Henceforth, we restrict our discussion to the $<$ operator. The constraints with $>$ operator can be resolved symmetrically. Fig. 1 shows the data flow of a typical location-based service employing location constraint evaluation, as illustrated by the location constraint matching problem. Location updates of mobile objects are streamed into the system and trigger the evaluation of constraints by the constraint solver. Constraint matches are communicated back to the interested subscribers via notifications.

Existing data management and indexing techniques for moving objects [8, 3, 13, 14, 10] are well suited to support relatively smaller numbers of range queries or nearest neighbor queries. These techniques are not suited to solve the location constraint matching problem, because they do not address the efficient evaluation of correlations over a large number of sets of these objects. In a wireless network, often thousands of

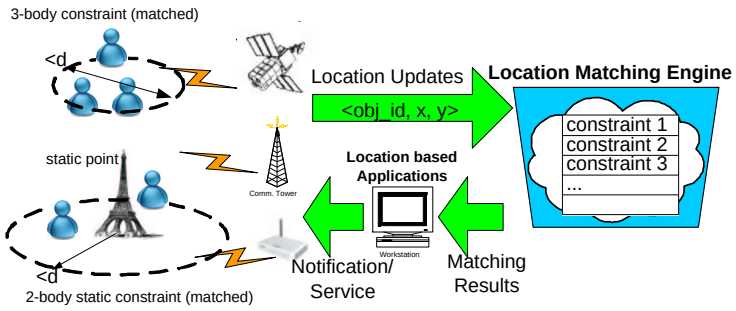


Figure 1: Data Flow for Location Constraint Matching

moving objects could be involved. If these objects continuously move and frequently change location vectors, the underlying database would be busy updating the objects' location without being able to evaluate the location constraints among different sets of objects.

The location constraint matching problem is different from the nearest neighbor problem [7, 22] in that the nearest neighbor problem determines the nearest object(s) among all the objects in the space to a given point, while the location constraint matching problem determines whether a specific set of objects are in a given spatial constellation to each other at one point in time. Our work focuses on how to evaluate a large number of constraints efficiently; it does not primarily focus on solving a single constraint over static data.

Our algorithm can be customized with different metrics for calculating the proximity relation and is not strictly tied to the application of the smallest enclosing circle relation. Moreover, in later sections we show how to support different query types, such as continuous reverse range queries and continuous reverse kNN queries.

In prior work [19], we developed a space partitioning and a Grid-file based indexing scheme for solving the location constraint matching problem. The approach statically decides on a space partitioning. This approach works well when the movement pattern of objects is regular and stable (i.e., objects only move within static clusters). This supports location constraint matching for areas with well know hotspots, such as shopping malls, stadiums, and resorts. However, the performance of this approach deteriorates, if the clusters themselves move, objects form new clusters, and objects move across the plane. Scenarios supporting this are the movement of crowds through a city, commuters (bus, train, planes), ad hoc assembly of crowds. In these cases, the static index can not cope with the evolving and changing movement patterns. In the worst case, the performance can not even compete with the naïve approach, where all the constraints are evaluated sequentially.

These limitations are the driver underlying this work. We propose an adaptive space partitioning scheme supported by two indexing structures to solve the location constraint matching problem. The algorithms adapt to the change in the movement patterns of objects and support a wide-range of query types and proximity relations.

We develop a detailed analytical model to quantify the cost of constraint evaluation for both in-memory and I/O-incurring environments. We experimentally validate the efficiency of our algorithm against various movement patterns, compare to alternative approaches, and demonstrate the scalability of the approach.

In Section 2, we describe the location constraint matching algorithm. In Section 3, we develop the analytical model to asses the constraint evaluation cost, determine system parameters, and estimate secondary storage access cost. The system architecture of the location constraint processing is described in Section 4. Section 5 presents the experimental evaluation of the algorithm. In Section 6, we put our work in perspective to related approaches.

2 Location Constraint Matching Algorithm

Our solution is based on space partitioning. The intuition behind our approach is that, a space partition that approximates the location of the moving objects, rather than the exact position of each object, is sufficient for most constraint evaluations. The details will be illustrated in the following subsections. Below we describe our space partition schemes and show how they benefit our approach for efficient location constraints evaluation.

2.1 Space Partitioning

The intuition of space partitioning is that partitions with sufficient granularity can approximate the position of the objects inside the partition and with this partition information, certain constraints can be resolved with confidence and therefore can be pruned.

To partition the space, we propose the adaptive k-d-tree (AKDT) and the adaptive multi-layer grid (AMLG) indexing that can dynamically adapt to movement patterns and constraint workloads.

AKDT is fundamentally a k-d-tree, except that it partitions the space rather than the objects in the space (as a traditional k-d-tree does). In the AKDT tree (see Fig. 2), each leaf node (n_3,n_7,n_8 ...) represents one basic partition of the space and every internal node represents the partition which is the union of all the leaf node partitions underneath it; the internal node also stores the information of the splitting line. For instance, in Fig. 2 the node n_4 stores not only the information of the partition ($S_2 + S_3$) which is the union of n_7 and n_8 but also the splitting line L_4 that separates S_2 and S_3. The root node of the tree represents the whole space. Initially, each object can be associated with a leaf node partition by tracing from the root node to the leaf node and the branching decision is made depending on which side of the splitting line the object takes. For example, object a is associated with partition S_3 (node n_8) because a is on the left side of L_1 (left branch of root) and under L_2 and L_4 (right branch of node n_1 and n_4).

We assume that the position of each object is retrieved with GPS, ground-based sensors or other location positioning technology [23]. We call each position retrieval a *location update* and we call it a *partition update* when an object moves from one

partition to another. To avoid the overhead of top-down scan for each partition update a backtracking algorithm is used to quickly associate the object with the new partition. Fig. 2 shows this, for object a moving from partition S_3 to S_2. The association process only backtracks from node n_8 to node n_4 and realizes that a is still inside the partition represented by n_4, therefore, the association process is tracing downwards from node n_4 to n_7 (because a moves across L_4).

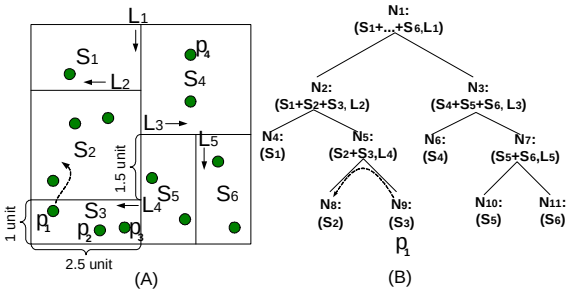


Figure 2: Space Partitioning with AKDT

A simple alternative method we propose as point index is the adaptive multi-layer grid index (AMLG) which is based on the grid file [9], In AMLG (see Fig. 3), the whole space is partitioned with different layers of grids, the grids in the same level are of equal-sized cells. The association from the object to the partition is similar to the AKDT.

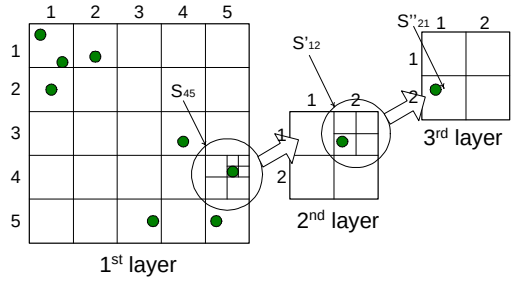


Figure 3: Space Partitioning with AMLG

Partition information (an approximation to the precise position) of the object may already allows us to tell that some location constraints are *satisfied* [1] . In Fig. 2, the 2-body constraint $|b, c| < 3$ must be *satisfied* because b and c are in the same partition and the diagonal of the partition is less than 3. Likewise, the partition information alone also may tell that some constraints are *unsatisfied.* For instance, 2-body constraint

[1] A constraint is *satisfied* if it is matched no matter where the associated objects locate within their partitions. A constraint is *unsatisfied* if it cannot be matched no matter where the associated objects locate within their partitions. And a constraint is *uncertain* if it cannot be evaluated solely based on the partition information of the associated objects.

$|c, d| < 1$ must be *unsatisfied* because the smallest distance between partitions S_3 and S_4 (where c and d is located) is at least 1.5. And a constraint is *uncertain* if the result can not be determined from the partition information. Only the constraints that are *uncertain* needs to be further evaluated with the precise position information. The constraints that are *satisfied* or *unsatisfied* are immediately pruned.

2.2 Adaptive Adjustment of Space Partitioning

This section describes how to dynamically adjust the space partitioning to account for changes in the movement patten of objects over time. To address this, AKDT dynamically sets and relinquishes the splitting lines as the movement pattern of the objects and constraint load changes. For the adjustment, each partition S_i (i is the partition ID) maintains the following additional information:

1. U_{S_i}: The number of *uncertain* constraints associated with the objects in partition S_i.
2. Obj_{S_i}: the total number of mobile objects inside partition S_i.

The adaptive space partition algorithm consists of two stages, name-ly, the initial partitioning and the adjustment stage. The initial partitioning simply breaks the whole space into small cells with sufficient granularity. The adjustment stage tunes the partition according to the objects' movement on a periodical basis. The adjustment frequency is set to be proportional to the average velocity of the moving objects, which is derived from the recorded location updates over time.

The adjustment process is composed of the *split* (see Fig. 4) and the *merge* process (see Fig. 5). The split process creates two new smaller partitions from a leaf node partition by adding a tentative splitting line (line 3) inside the candidate leaf node. The candidate leaf nodes are selected (line 2) such that the nodes contain the largest number of *uncertain* constraints or they are chosen randomly, if the number of *uncertain* constraints is the same.

The split process may enhance the partition granularity and reduce the number of *uncertain* constraints, but it also induces more partition update overhead. The number of objects Obj_{S_i} serves as an estimation of the partition update rate that is incurred for the leaf node partition when splitting lines are added. The algorithm only commits the splitting line in the partition where it leads to the greatest reduction in the number of *uncertain* constraints (recorded in set Γ_C in line 7) and the smallest increase in partition update rate (recorded in Γ_O in line 8). Only these tentative splitting lines inside partitions $\Gamma_U \cap \Gamma_O$ are confirmed (line 9). The final number of partition splits depends on the size of Γ_C and Γ_O, which are controlled by the parameters N_{Γ_C} and N_{Γ_O}. These two parameters are the main indicator of the adaptation speed, which is directly coupled with, and therefore adjusted according to, the speed of the evolution of the movement pattern. The reduction in the number of *uncertain* constraints is computed in line 5.

Lemma 2 in Section 3 shows that the *uncertain* constraints after the splitting are a subset of the *uncertain* constraints before splitting, therefore, only the original *uncertain*

constraints are considered when computing the number of *uncertain* constraints after splitting (line 4). This reduces the overhead of the split process.

Procedure *Split_Adjust*(INTEGER N_{Γ_U}, N_{Γ_O})
1: begin
2: **for** each candidate leaf node S_i
3: $S_i^{'}$ = S_i + a tentative splitting line l_i
4: compute $U_{S_i'}$ based on $S_i^{'}$
5: $\triangle U_{S_i} = U_{S_i} - U_{S_i'}$
6: **end for**
7: Γ_U = top N_{Γ_U} partitions with highest $\triangle U_{S_i}$
8: Γ_O = top N_{Γ_O} partitions with lowest Obj_{S_i}
9: replace S_i with $S_i^{'}$ if $S_i \in \Gamma_U \cap \Gamma_O$
10: end

Figure 4: Partition Adjustment (Split)

The merge process is an reverse operation of the split process. It removes the current splitting lines and merges the leaf node partitions. The merge process is targeted at reducing the partition update overhead at the cost of a minor increase in the number of *uncertain* constraints. The candidate nodes are selected (line 2) such that the nodes contain the most objects (therefore the highest expected partition update rate) or they are chosen randomly, if all of the nodes contain the same number of objects. Based on Lemma 2 in Section 3, the *uncertain* constraints before merging are a subset of the *uncertain* constraints after merging, therefore, the original *uncertain* constraints must be included (without computation) when computing the number of *uncertain* constraints after merging (line 4). This reduces the overhead of the merging process.

The candidate splitting line generated in the splitting process is random in terms of its orientation (horizontal or vertical) and its position could be strict (e.g. bisects the partition equally) or follows some given distribution (e.g., normal distribution). Split and merge processes run periodically to split the partition that has not been partitioned to the granularity (to reduce the number of *uncertain* constraint) or to merge the partition that cause too many partition updates. For AMLG indexing, the splitting and merging process is similar to that of AKDT.

2.3 Matching Algorithms

With the space partition, the evaluation of location constraints can be performed very efficiently. The matching algorithm consists of two parts, the constraint evaluation based on partition information and constraint evaluation based on exact position information. For the evaluation based on the exact position, Welzl's algorithm [17] is

Procedure *Merge_Adjust*(INTEGER N_{Γ_U}, N_{Γ_O})
1: begin
2: **for** each candidate second level node S_i
3: $S_i^{'}$ = S_i - splitting line in S_i
4: compute $U_{S_i'}$ based on $S_i^{'}$
5: $\triangle U_{S_i} = U_{S_i'} - U_{S_i}$
6: **end for**
7: Γ_U = top N_{Γ_U} partitions with lowest $\triangle U_{S_i}$
8: Γ_O = top N_{Γ_O} partitions with highest Obj_{S_i}
9: replace S_i with $S_i^{'}$ if $S_i \in \Gamma_U \cap \Gamma_O$
10: end

Figure 5: Partition Adjustment (Merge)

adopted, which computes the smallest circle that encloses n points in $O(n)$ time. Below, we introduce partition-based evaluation that happens only when some object updates its partition or when the partition scheme is adjusted. The implementation of the algorithms is orthogonal to the space partitioning methods, it takes advantage of the space partitioning and uses the partition as a rough estimation for the position of the moving objects. For certain constraint evaluations, this approximation is enough to determine a result. In the following, we present our partition-based evaluation algorithms in detail.

Partition-Based Evaluation for N-body Constraint: For n-body location constraint $|p_1{}^t, p_2{}^t, ..., p_n{}^t| < d$, when some object p_i moves into a new partition S_i, all the constraints it is associated with need to be re-evaluated based on the partition information (Fig. 6). Suppose that c_j is the constraint p_i is associated with and $P = \{p_1, p_2, ..., p_n\}$ is the set of bodies involved in c_j ($p_i \in P$).

If the partitions that contain the objects in P cannot all be intersected by the circle with diameter d (this is checked by function *Intersect* in line 4), c_j is *unsatisfied*. On the other hand, if all objects in P are in the partitions whose union can be covered by a circle with diameter d (this is checked by function *Enclose* in line 6), c_j is *satisfied*. If c_j does not fall into the above two categories, the constraint is *uncertain* (line 9).

The result of an evaluation is valid as long as the objects remain in their partitions and the partition scheme does not change. Fig. 7 illustrates partition-based evaluation. The 4-body constraint $|a, b, c, e| < 3$ is *unsatisfied* because a, b, c and e are contained inside the partitions S_{11}, S_{21} and S_{55}, and $Intersect(S_{11}, S_{21}, S_{55}) > 3$ (the distance between S_{11} and S_{55} is $3\sqrt{2}$ and the diameter of the circle intersecting both S_{11} and S_{55} must be larger than $3\sqrt{2} > 3$). Constraint $|a, b, c, d| < 3$ is *satisfied* because $Enclose(S_{11}, S_{12}, S_{21}) = 2\sqrt{2} < 3$. However, constraint $|e, f, g, h| < 3$ is *uncertain* because $Intersect\ (S_{34}, S_{45}, S_{53}, S_{55}) < 3$ and $Enclose\ (S_{34}, S_{45}, S_{53}, S_{55}) > 3$.

procedure *Partition_Based_NB*(MobileObject p)
1: begin
2: **for** each Constraint c that p is associated with
3: **Let** $P = \{p_1, p_2, ..., p_n\}$ be the set of bodies in c;
4: **if** (*Intersect*($\bigcup_{i=1}^{n} p_i.partition$)$> c.d$))
5: $c.result$ = *unsatisfied*;
6: **else if** (*Enclose*($\bigcup_{i=1}^{n} p_i.partition$)$< c.d$)
7: $c.result$ = *satisfied*;
8: **else**
9: $c.result$ = *uncertain*;
10: **end for**
11: end

Figure 6: Partition-Based Evaluation for N-body Constraint

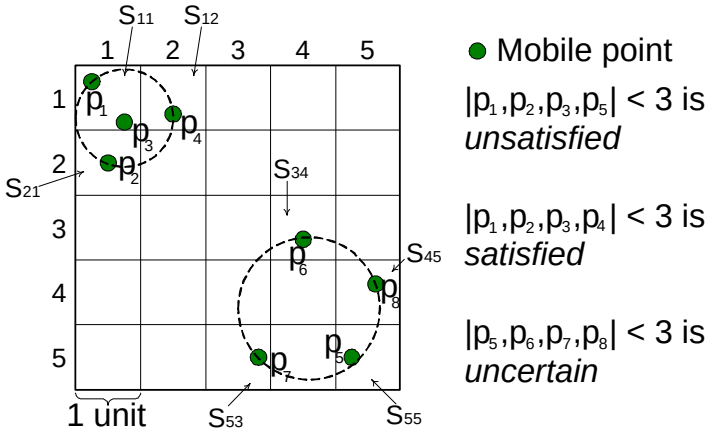


Figure 7: Illustration of Partition Based Evaluation (N-body)

Partition-Based Evaluation for N-body Static Constraint: The n-body static constraint, $|A, {p_1}^t, {p_2}^t, ..., {p_n}^t| < d_A$, is matched if and only if all p_i (i∈1..n) are in the circle O with static point A as center and d_A as radius. Depending on whether the partition is inside, intersecting, or outside the boundary of O, we identify the *internal*, *bounding*, and *external* partitions of O (e.g., the partition completely inside O is identified as *internal* partition and so on.) The distance between the moving object and static point A can be tracked according to the type of partition the object is in.

The partition-based evaluation (see Fig. 8) is as follows: if some object in $P = \{p_1, p_2, ..., p_n\}$ is in the *external* partition, the constraint is *unsatisfied* (line 6); if all objects are in the *internal* partition, the constraint is *satisfied* (line 8); if the constraint does not belong to the above two cases, the constraint is *uncertain* (line 10). Fig. 9 illustrates the partition-based evaluation. The 1-body static constraint $|I, e| < 2$ is *unsatisfied* because S_{55} (where e is located) is an *external* partition w.r.t. O (a circle

with I as the center and 2 as the radius). On the other hand, constraint $|I, f| < 2$ is *satisfied* because S_{34} (where f is located) is an *internal* partition w.r.t. O. However, constraints $|I, a| < 2$, $|I, b| < 2$, $|I, c| < 2$, $|I, d| < 2$, $|I, g| < 2$ and $|I, h| < 2$, are *uncertain* because S_{11}, S_{12}, S_{21}, S_{53}, S_{45} are *bounding* partitions w.r.t. O.

```
procedure Partiton_Based_NBS(MobileObject p)
1: begin
2:       for each Constraint c that p is associated
4:         let P = [p_1, p_2, ..., p_n] be the set of bodies in c;
5:         if (∃p_i ⊂ P, p_i ∈ external partition)
6:           c.result = unsatisfied;
7:         else if (∀p_i ⊂ P, p_i ∈ internal partition)
8:           c.result = satisfied;
9:         else //in bounding partition
10:          c.result = uncertain;
11:      end for
12: end
```

Figure 8: Partition-Based Evaluation for N-body Static Constraint

Constraint Evaluation Algorithm: Change in location information triggers constraint evaluation as shown in Fig. 10. We distinguish the case where the location change leads to partition updates and where no such updates are incurred.

For a partition update, all constraints an object is associated with have to be evaluated using the partition-based evaluation function (line 5, 6) and the *uncertain* constraints of the object have to be explicitly evaluated afterwards (with Match_Uncertain_Constraints in line 7). This evaluation is based on the exact location position information. The number of *uncertain* constraints and the number of objects in the previous and current partitions are updated afterwards (line 8, 9).

On the other hand, if the location update does not incur partition update, only *uncertain* constraints (line 11) need to be evaluated. Partition-based evaluation is invoked every time an object changes partition or when the partition is adjusted, but not for every location update.

2.4 Conflicting and Congruous Constraints

We notice that there could be constraints that are conflicting with each other. For instance, $|{p_1}^t, {p_2}^t| > d$ and $|{p_1}^t, {p_2}^t| < d$ cannot be *satisfied* at the same time. If one constraint is evaluated to be *satisfied*(true), the other must be *unsatisfied*(false). Likewise, constraint could also be congruous; i.e. if a constraint is *satisfied*, some other constraint must also be satisfied. For instance, if $|{p_1}^t, {p_2}^t, {p_3}^t| < d$ is *satisfied*,

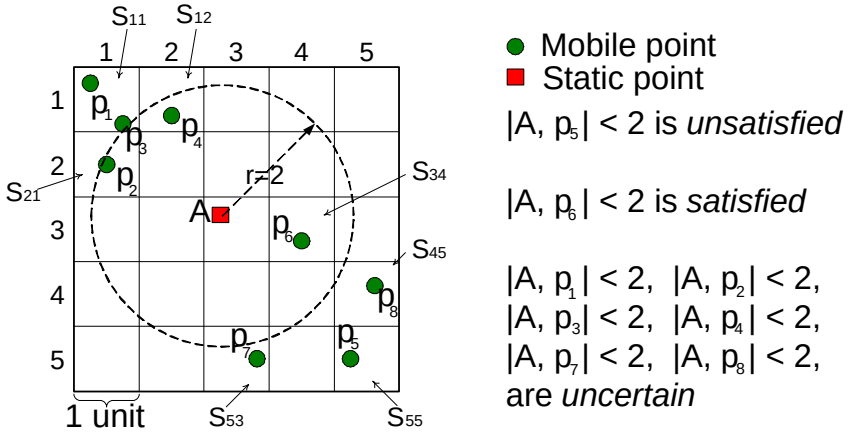


Figure 9: Illustration of Partition-Based Evaluation (N-body Static)

then it must follow that $|{p_1}^t, {p_2}^t| < d$ is also *satisfied*. Notice that the conflicting and congruous relation is asymmetric. In the above example, $|{p_1}^t, {p_2}^t| < d$ is *unsatisfied* might not indicate $|{p_1}^t, {p_2}^t| > d$. Also even if $|{p_1}^t, {p_2}^t| < d$, $|{p_1}^t, {p_2}^t, {p_3}^t| < d$ might not be true. To amortize the evaluation cost, a secondary graph structure is used to index the conflicting and congruous constraints. This graph structure is constructed and incrementally updated when the new constraint is inserted or removed from the system; and this can also be done offline for the constraints are known beforehand. The conflicting or congruous relationship are identified with a set of rules as follows. Suppose that c_1 and c_2 are conflicting or congruous constraints and each is associated with a set of mobile objects (P_1 for c_1 and P_2 for c_2). Depending on the result of the constraint, there could only be four possible cases:

1. c_1 is *satisfied* $\rightarrow$ c_2 *satisfied*(congruous):
   a.1) $c_1.\texttt{op} = c_2.\texttt{op} = " < "$ and $c_1.d \leq c_2.d$ and $P_1 \supseteq P_2$.
   a.2) $c_1.\texttt{op} = c_2.\texttt{op} = " > "$ and $c_1.d \geq c_2.d$ and $P_1 \subseteq P_2$.
2. c_1 is *satisfied* $\rightarrow$ c_2 *unsatisfied*(conflicting):
   b.1) $c_1.\texttt{op} = " < "$ and $c_2.\texttt{op} = " > "$ *or* $" = "$ and $c_1.d \leq c_2.d$ and $P_1 \supseteq P_2$.
   b.2) $c_1.\texttt{op} = " > "$ and $c_2.\texttt{op} = " < "$ *or* $" = "$ and $c_1.d \geq c_2.d$ and $P_1 \subseteq P_2$.
   b.3) $c_1.\texttt{op} = " = "$ and $c_2.\texttt{op} = " < "$ and $c_1.d \geq c_2.d$ and $P_1 \subseteq P_2$.
   b.4) $c_1.\texttt{op} = " = "$ and $c_2.\texttt{op} = " > "$ and $c_1.d \leq c_2.d$ and $P_1 \supseteq P_2$.
3. c_1 is *unsatisfied* $\rightarrow$ c_2 *satisfied*(conflicting):
   c.1) $c_1.\texttt{op} = " > "$ and $c_2.\texttt{op} = " < "$ and $c_1.d < c_2.d$ and $P_1 \supseteq P_2$.
   c.2) $c_1.\texttt{op} = " < "$ and $c_2.\texttt{op} = " > "$ and $c_1.d > c_2.d$ and $P_1 \subseteq P_2$.

procedure *Evaluation*(MobileObject p)
1: begin
2: $p.prev\_partition = p.partition$;
3: $p.partition$ =Find_New_Partition(p);
4: case 1: $p.prev\_partition \neq p.partition$
5: Partition_Based_NB(p);
6: Partition_Based_NBS(p);
7: Match_Uncertain_Constraints(p);
8: update $U_{p.partition}$ and $U_{p.prev\_partition}$;
9: update $Obj_{p.partition}$ and $Obj_{p.prev\_partition}$;
10: case 2: $p.prev\_partition = p.partition$
11: Match_Uncertain_Constraints(p);
12: end

Figure 10: Evaluation Algorithm

4. c_1 is *unsatisfied* $\rightarrow$ c_2 *unsatisfied*(congruous):

   d.1) $c_1.\mathtt{op} = c_2.\mathtt{op} =$ " $<$ " and $c_1.d \geq c_2.d$ and $P_1 \subseteq P_2$.

   d.2) $c_1.\mathtt{op} = c_2.\mathtt{op} =$ " $>$ " and $c_1.d \leq c_2.d$ and $P_1 \supseteq P_2$.

The outcome of the constraints may be partially conflicting or congruous with each other. This happens when the object sets of the constraints are not completely the same ($P_1 \neq P_2$) and their intersection $P_1 \cap P_2$ contains more than two objects. Because of the overlap of the involved object sets, the result of some constraint may be valuable partially for the evaluation of other constraints. For instance, if $|{p_1}^t, {p_2}^t, {p_3}^t| < d$ is *satisfied*, then $|{p_2}^t, {p_3}^t, {p_4}^t| < d$ would probably also be *satisfied*, since both constraints involve p_2 and p_3 and $|{p_2}^t, {p_3}^t| < d$. This is what we call partially congruous. Likewise, if $|{p_1}^t, {p_2}^t, {p_3}^t| > d$ is *satisfied*, then $|{p_2}^t, {p_3}^t, {p_4}^t| < d$ would probably be *unsatisfied*. This is what we call partially conflicting. To fully make use of the correlation information embedded in those constraints with object sets partially overlapped, the notion of the virtual constraint is adopted. A virtual constraint, c_v, involves the mobile objects that is the join of the object sets of both constraints(e.g., $P_1 \cap P_2$), and its operator $c_v.\mathtt{op}$ and alerting distance $c_v.d$ are defined with the following rules:

1. c_v is *satisfied* $\rightarrow$ c_1, c_2 are *satisfied*(congruous):

   a.1) if $c_1.\mathtt{op} = c_2.\mathtt{op} =$ " $>$ ", then $c_v.\mathtt{op} =$ " $>$ " and $c_v.d = \mathtt{max}_{i=1,2}(c_i.d)$

2. c_v is *satisfied* $\rightarrow$ c_1, c_2 are *unsatisfied*(conflicting):

   b.1) if $c_1.\mathtt{op} = c_2.\mathtt{op} =$ " $<$ " or "=", then $c_v.\mathtt{op} =$ " $>$ " and $c_v.d = \mathtt{max}_{i=1,2}(c_i.d)$

3. c_v is *unsatisfied* $\rightarrow$ c_1, c_2 are *satisfied*(conflicting):

   c.1) if $c_1.\mathtt{op} = c_2.\mathtt{op} =$ " $>$ ", then $c_v.\mathtt{op} =$ " $<$ " and $c_v.d = \mathtt{max}_{i=1,2}(c_i.d) + \epsilon$ $(\epsilon > 0)$

4. c_v is *unsatisfied* $\rightarrow$ c_1, c_2 are *unsatisfied*(congruous):

   d.1) if $c_1.\texttt{op} = c_2.\texttt{op} = \text{" } < \text{"}$, then $c_v.\texttt{op} = \text{" } < \text{"}$ and $c_v.d = \texttt{max}_{i=1,2}(c_i.d)$

5. c_v is *satisfied* $\rightarrow$ c_1 is *satisfied*(congruous), c_2 is *unsatisfied*(conflicting):

   e.1) if $c_1.\texttt{op} = \text{" } > \text{"}$ and $c_2.\texttt{op} = \text{" } < \text{"}$, then $c_v.\texttt{op} = \text{" } > \text{"}$ and $c_v.d = \texttt{max}_{i=1,2}(c_i.d)$

6. c_v is *unsatisfied* $\rightarrow$ c_1 is *satisfied*(conflicting), c_2 is *unsatisfied*(congruous):

   f.1) if $c_1.\texttt{op} = \text{" } > \text{"}$ and $c_2.\texttt{op} = \text{" } < \text{",,}$ then $c_v.\texttt{op} = \text{" } < \text{"}$ and $c_v.d = \texttt{max}_{i=1,2}(c_i.d) + \epsilon \ (\epsilon > 0)$

Notice that we deliberately neglect the low-probability cases. For example, when c_v, with $c_v.\texttt{op} = \text{" } = \text{"}$ and $c_v.d = c_i.d$, is *satisfied*, c_i, with $c_i.\texttt{op} = \text{" } < \text{"}$ or $\text{" } > \text{"}$, will be *unsatisfied*. But the probability that c_v with equality operator is satisfied is very low, the pruning base on this virtual constraint is not likely to be efficient. In the evaluation, the constraint of the virtual node has higher priority and they will always be evaluated before other constraints.

Similar to n-body constraints, the rules for conflicting and congruous relationship for n-body static constraint can be likewise derived as below:

1. c_1 is *satisfied* $\rightarrow$ c_2 *satisfied*(congruous):

   a.1) $c_1.\texttt{op} = c_2.\texttt{op} = \text{" } < \text{"}$ and $dist(c_1.A, c_2.A) \leq c_2.d - c_1.d$ and $c_1.d \leq c_2.d$ and $P_1 \supseteq P_2$.

   a.2) $c_1.\texttt{op} = c_2.\texttt{op} = \text{" } > \text{"}$ and $dist(c_1.A, c_2.A) \leq c_1.d - c_2.d$ and $c_1.d \geq c_2.d$ and $P_1 \supseteq P_2$.

2. c_1 is *satisfied* $\rightarrow$ c_2 *unsatisfied*(conflicting):

   b.1) $c_1.\texttt{op} = c_2.\texttt{op} = \text{" } < \text{"}$ and $dist(c_1.A, c_2.A) > c_1.d + c_2.d$ and $P_1 \cap P_2 \neq \phi$.

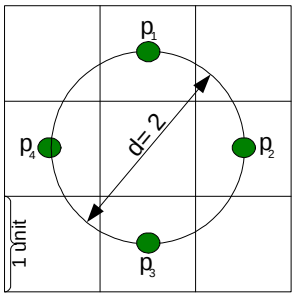
   b.2) $c_1.\texttt{op} = \text{" } < \text{"}$ and $c_2.\texttt{op} = \text{" } > \text{"}$ and $dist(c_1.A, c_2.A) \leq c_2.d - c_1.d$ and $c_1.d \leq c_2.d$ and $P_1 \cap P_2 \neq \phi$.

3. c_1 is *unsatisfied* $\rightarrow$ c_2 *unsatisfied*(congruous):

   c.1) $c_1.\texttt{op} = c_2.\texttt{op} = \text{" } < \text{"}$ and $dist(c_1.A, c_2.A) \leq c_1.d - c_2.d$ and $c_1.d \geq c_2.d$ and $P_1 \subseteq P_2$.

   c.2) $c_1.\texttt{op} = \text{" } > \text{"}$ and $c_2.\texttt{op} = \text{" } < \text{"}$ and $dist(c_1.A, c_2.A) > c_1.d + c_2.d$ and $P_1 \subseteq P_2$.

To construct the graph structure, initially, each constraint(including the virtual constraint) is regarded as node in the graph, it is then connected to its conflicting and congruous constraints (other nodes) with unidirectional edges (asymmetric relation), each appropriately labeled with transit condition, e.g. *satisfied* $\rightarrow$ *unsatisfied* (shorthanded as $s \rightarrow u$) etc. Then, the nodes with the same neighbors and are connected to those neighbors with the same transit condition are merged into a compound node. Reversely, when the constraint is removed, the corresponding node may need to be



$|p_1, p_2, p_3, p_4| < 2.5$ is *satisfied*, therefore $|p_2, p_3, p_4| < 2.5$, $|p_1, p_3, p_4| < 2.5$, $|p_1, p_2, p_4| < 2.5$, $|p_1, p_2, p_3| < 2.5$, $|p_1, p_2| < 2.5$, $|p_1, p_3| < 2.5$, $|p_1, p_4| < 2.5$, $|p_2, p_3| < 2.5$, $|p_2, p_4| < 2.5$, $|p_3, p_4| < 2.5$ are all *satisfied*.

Figure 11: Pruning that is not Possible with Space Partition

deleted. Notice that this graph maybe disconnected and is partition into discrete components. one constraint in a node is computed, a width-first transverse of the graph starting from that node is followed, and the results of the constraints represented with those nodes are immediately evident without computation. Precaution for the loop is taken appropriately with boolean bit indicator for the visited node. Virtual constraints are evaluated based on adaptive space partition just as an ordinary constraint. This secondary graph data structure is most effective on conflicting and congruous constraint pruning when the constraints represented by the nodes are *uncertain*, because the corresponding redundant computation cannot be pruned with space partition. Notice that, the constraint pruning with conflicting and congruous property is based on the syntax and it is orthogonal to the space partition pruning. In the following Fig. 11, the constraints $|B, C, D| < 2.5$, $|A, C, D| < 2.5$, $|A, B, D| < 2.5$, $|A, B, C| < 2.5$, $|A, B| < 2.5$, $|A, C| < 2.5$, $|A, D| < 2.5$, $|B, C| < 2.5$, $|B, D| < 2.5$, $|C, D| < 2.5$ are not possible to be pruned based on partitions where A, B, C, D are located, because the minimum size of the circle enclosing any two or three of them is larger than 2.5 and the minimum size of the circle intersecting any two or three of them is smaller than 1. But all of them can be pruned (as *satisfied*) if we know that $|A, B, C, D| < 2.5$ is *satisfied*.

In the following Fig. 12, we give an example of conflicting or congruous constraints indexed with unidirectional graph structure. The five constraints being indexed are shown in node N_1, N_3, N_4 N_5 and N_6. The constraints in node N_3 and N_4 share the same objects p_2 and p_3, therefore a virtual node N_2 representing constraint $|p_2, p_3| > d$ is constructed. If the virtual node N_2 is *satisfied*, the both node N_3 and N_4 will be *unsatisfied* (unidirectional edge from N_2 to both N_3 and N_4). Also, if N_3 or N_4 is *satisfied*, then N_2 must be *unsatisfied* (unidirectional edge from both N_3 and N_4 to N_2). Since the two unidirectional edges between N_2 and N_3 (and N_4) have the same label ($s \rightarrow u$), they are combined as a single bi-directional edge. Observe that N_3 and N_4 are connected to the edges with the same label to other nodes, therefore they can be combined to form a compound node. All the other transit edges are based on the rules listed above. An example of constraint processing is that when the object p_2 updates its location, then the constraint $|p_2, p_3| > d$ in N_2 will be evaluated first (given it is *uncertain*) because virtual node has higher priority. If it is *unsatisfied*, then the constraint in N_1 is *satisfied* because the transit condition from N_2 to N_1 is $u \rightarrow s$. Since

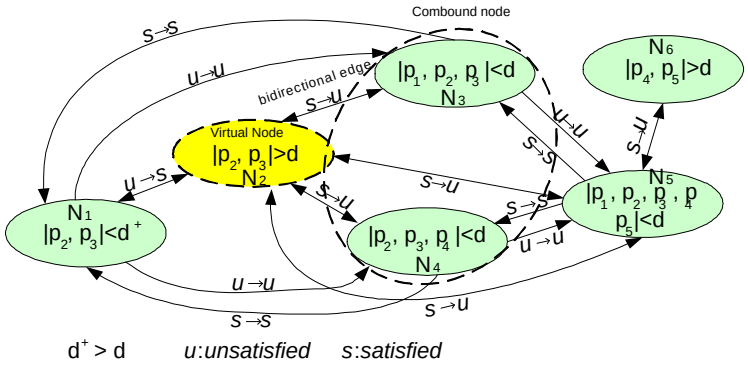


Figure 12: Graph Structure for Conflicting and Congruous Constraints

there is not edge coming out from N_1 labeled with $s \rightarrow *$, the transverse terminates. On the other hand, if constraint in N_2 is *satisfied*, then constraints in compound node $N_3 \& N_4$ and N_5 must be *unsatisfied*($s \rightarrow u$). Other scenarios can be likewise verified.

2.5 Extended Constraints

In this section we illustrate how the partition-based evaluation algorithm can be applied to other constraints. The illustration is based on a continuous reverse range query (CRR) and a continuous reverse k-nearest neighbor query (CRKNN).

1. Given a rectangular region R, a continuous reverse range query, $CRR(R, o)$ continuously checks whether a moving objects o is inside R.

2. Given a static point P, a continuous reverse k-nearest neighbor query, $CRKNN$ (P, o), continuously checks whether P is the kNN of o.

In the discussion below we refer to these queries as constraints to emphasize the inverse character of the problem, i.e., given a potentially large set of constraints and changing object location information, we are looking for the matched constraints. As in the previous section, the objective of the partition-based evaluation is to reduce the number of constraints that have to be evaluated against precise position information.

Suppose object o is located inside partition S_o. The partition-based evaluation for $CRR(R, o)$ works as follows. If S_o is disjoint with R, the constraint is *unsatisfied.* If S_o is completely enclosed by R, the constraint is *satisfied*, otherwise (S_o intersects R), it is *uncertain.* Fig. 13(A) shows three rectangular areas, R_1, R_2 and R_3 and three object o_1, o_2, and o_3. Since partition S_1, which accommodates o_1, is disjoint with R_1, the constraint $CRR(R_1, o_1)$ can not be satisfied, therefore it is *unsatisfied.* $CRR(R_2, o_1)$ is a *satisfied* constraint since S_1 is completely inside R_2 and $CRR(R_3, o_1)$ is *uncertain* since R_3 partially intersects S_1. Likewise, we may conclude that $CRR(R_1, o_2)$ is *uncertain* and $CRR(R_2, o_2)$, $CRR(R_3, o_2)$, $CRR(R_1, o_3)$, $CRR(R_2, o_3)$ and $CRR(R_3, o_3)$ are *unsatisfied.*

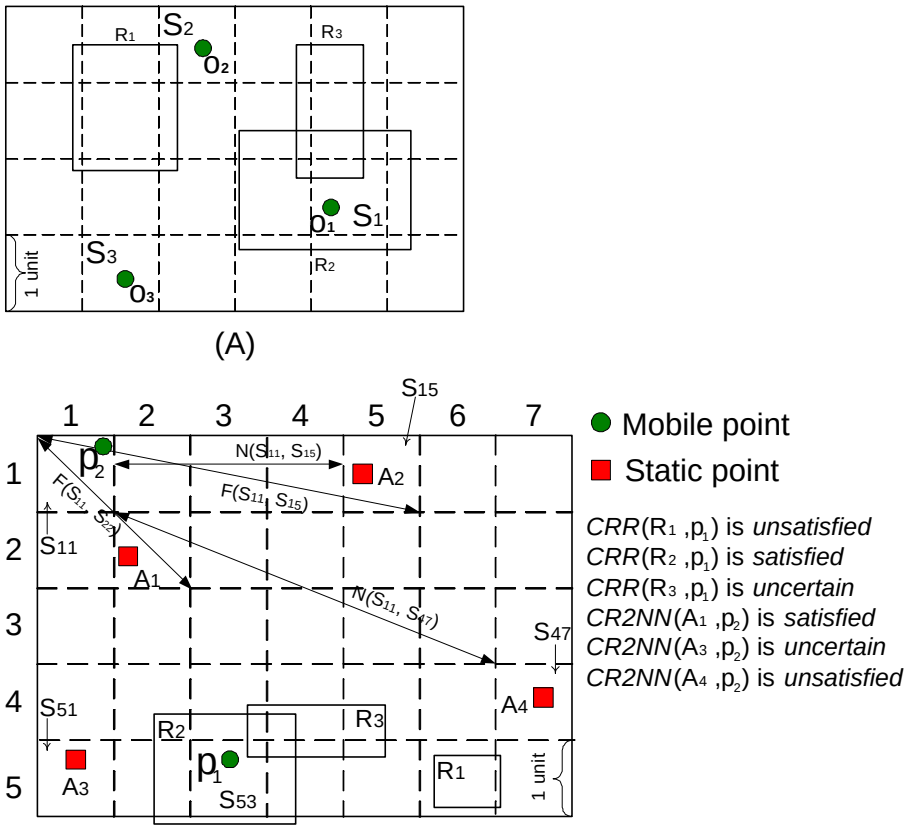


Figure 13: (A) CRR query and (B) CRKNN query

The partition-based evaluation for $CRKNN(P, o)$ works as follows. Suppose P is inside the partition S_P. For object o, the system maintains the partition where o is located (S_o) and the partitions where its kNNs are located (without loss of generality, suppose they are S_i ($1 \leq i \leq k$)). If the nearest distance between S_o and S_P is greater than the furthest distance between S_o and S_i, or $N(S_o, S_P) > max_{1 \leq i \leq k} F(S_o, S_i)$ (function N and F return the nearest distance and furthest distance between two partitions. If one of the partition is degraded to a point then they return the nearest distance and furthest distance between a partition and a point), then P can not be one of the kNNs of o, and $CRKNN(P, o)$ is *unsatisfied*. However, if the furthest distance between S_o and S_P is smaller than the smallest distance between S_o and S_i (for some i), or $F(S_o, S_P) < \exists i N(S_o, S_i)$, then P must be one of the kNNs of o, and $CRKNN(P, o)$ is *satisfied* and the current kNNs have to be updated. If a constraint is not *unsatisfied* or *satisfied*, it is *uncertain*.

Fig. 13(B) shows a continuous reverse 3 nearest neighbor constraint $CR3NN(P, o)$. Suppose the current 3 nearest neighbors of o are A, B, and C, and object o triggers a

partition-based evaluation (e.g., due to a partition update). Since $N(S_o, S_P) = \sqrt{20} > max_{i \in \{A,B,C\}} F(S_o, S_i)$ ($F(S_o, S_A) = F(S_o, S_B) = \sqrt{13}$, $F(S_o, S_C) = \sqrt{17}$), P cannot be one of o's 3 nearest neighbors, $CR3NN(P, o)$ is *unsatisfied*. However, if o is inside partition $S_{o'}$, the constraint will be *satisfied*, because $F(S_{o'}, S_P) = \sqrt{13} < N(S_{o'}, S_A) = 4$. Similarly, if o is inside partition $S_{o''}$, the constraint is *uncertain*.

Continuous query CRR or $CRKNN$ all produces a result as a boolean value (yes or no answer). Next we show that the partition-based method can be applied to process the traditional continuous queries that has non-boolean result. Here, we focus our study on two most popular spatial queries, the continuous range query ($CR(R)$, R is the range) and continuous k nearest neighbor query ($CKNN(q)$, q is the query point).

First, for each continuous range query $CR(R)$, two lists, (the *internal* list and *bounding* list) are maintained. The *internal* list stores the references to the objects that are in the *internal* partition (those are inside the query range) of R and *bounding* list stores the reference to the objects that are in the *bounding* partition (those are intersecting the query range) R. Moreover, to keep track of the objects in the *bounding* list, a boolean vector bv, with each bit corresponding to an object, indicates whether it is inside (true) or outside (false) the range. The query result is simply all the objects in the *internal* list plus the objects in *bounding* list with the bv bits marked as true. When the object does not change partitions, containment condition is checked only for the objects in the *bounding* list and the boolean vector are also updated accordingly. Other location updates are pruned and the result of the query is partially updated each time. If the partition update occurs, the list should be appropriately modified incrementally to reflect the change. The partition update only involves the incremental update of two list. As an example, in Fig. 13(A) o_1 is in the *internal* list of $CR(R_2)$ and *bounding* list of $CR(R_3)$, o_2 is in *bounding* list of $CR(R_1)$, therefore $CR(R_2)$ always includes o_1 as result, and $CR(R_1)$ and $CR(R_3)$ may or may not includes o_2 and o_1 as result, respectively. No query includes o_3 as result, so any location update from it is safely pruned.

There are two types of continuous k nearest neighbor query ($CKNN(q)$), the order sensitive $CKNN$ and order insensitive $CKNN$. For the order sensitive $CKNN$, the order (based on the distance to the query point from near to far) of the objects distinguishes the query results, e.g., $< o_1, o_2 >$ is different from $< o_2, o_1 >$ since o_1 is closer to the query point in the first result but o_2 is closer to the query point in the second result. However, for the order insensitive $CKNN$, the order of the objects does not differentiate the result, e.g., $< o_1, o_2 >$ is considered to be the same as $< o_2, o_1 >$.

To incrementally process the order sensitive $CKNN$, each $CKNN$ keeps a record of current k nearest neighbors and the partitions that they are located (overhaul algorithm [21] can be used to get the initial result). To incrementally update the rank and query result, array $sd[i](fd[i])$ is used to record the smallest(largest) distances between the query point and the partition that contains ith nearest neighbor ($1 \leq i \leq k$).

We call the partition whose smallest distance to the query point is smaller than the $fd(k)$ the *k-neighborhood* partition and we call the partition whose largest distance to the query point is no larger than $sd(k)$ the *k-internal* partition. Then the k nearest neighbors are a subset of all the objects inside *k-neighborhood* partitions and are a superset of all the objects inside the *k-internal* partitions. Particularly, the

k-neighborhood partition that is not *k-internal* partition may contain objects that are not k nearest neighbors, but *k-internal* partition definitely contains only the objects in k nearest neighbors. It is clear that location updates outside *k-neighborhood* partition are immediately pruned because they would not affect the result anyway. Only the location updates involving the *k-neighborhood* partitions need to be further processed and this dramatically reduces the computation because only the objects within the vicinity of the query point are involved in the evaluation. The details are elaborated with the following few cases. If one of the k nearest neighbor is moving out of the *k-neighborhood* partition, then it is removed as one of the k nearest neighbors and the 1 nearest neighbor query is issued [21] and the result becomes the new kth nearest neighbor. Now suppose that the object moves into the *k-neighborhood* partition (this includes the case where the object moves within the *k-neighborhood* partition with or without partition update) or moves from a non-*k-neighborhood* partition into a *k-neighborhood* partition and suppose the destination partition is P. Then if the interval $[N(P,q),\ F(P,q)]$ does not overlap with any interval $[sd[i],\ fd[i]](1 \leq i \leq tr)$ (or $fd[i] \leq N(P,q), F(P,q) \leq sd[i+1]$ for some i), this object becomes one of the $i+1$ nearest neighbor , and the original nearest neighbors further away than this object need to increase their order by 1 and the original kth nearest neighbor is dropped from the answer list. Now, what if the interval $[N(P,q),\ F(P,q)]$ overlaps with some interval $[sd[i],\ fd[i]](1 \leq i \leq k)$. Then only the objects inside these partitions are compared with the new position of object and the order of the nearest neighbor are appropriately adjusted. Order insensitive $CKNN$ is much simpler than order sensitive $CKNN$, because any movement within the *k-internal* partitions is neglected since they would not affect the result anyway.

In Fig. 14, the current four nearest neighbors of the query q are $< A,\ B,\ C,\ D >$ in increasing order and they are located in partitions S_A, S_B, S_C and S_D. The sd and fd arrays record the distance from q to each partition, $sd[1] = N(S_A,q)$, $fd[1] = F(S_A,q)$, $sd[2] = N(S_B,q)$, $fd[2] = F(S_B,q)$, $sd[3] = N(S_C,q)$, $fd[3] = F(S_C,q)$, $sd[4] = N(S_D,q)$ and $fd[4] = F(S_D,q)$. Suppose that A is moved from partition S_A to S_B (dashed arrow), then its new position is compared only with the original ith nearest neighbor if the interval $[N(S_B,q),\ F(S_B,q)]$ intersects with the interval $[sd[i],\ fd[i]]$. In this case, $[N(S_B,q),\ F(S_B,q)]$ intersects with $[sd[1],\ fd[1]]$, $[sd[2],\ fd[2]]$ therefore, B is compared with A. In this case, A is further away than B, so A and B change their order which makes current result $< B,\ A,\ C,\ D >$, and sd and fd are updated accordingly. On the other hand, suppose E moved into partition $S_{E'}$, since S_E and $S_{E'}$ are both non-*k-neighborhood* partition, nothing needs to be done, the new position is simply discarded. But if it is moved to $S_{E''}$, its position has to be compared with the position of the third nearest neighbor C (because the interval $[N(S_{E''},q),\ F(S_{E''},q)]$ intersects only with $[sd[3],\ fd[3]]$) and their orders are adjusted and the fourth nearest neighbor D is dropped from the original result, making the current result $< A,\ B,\ E,\ C >$.

3 Analytical Model

In this section, we develop an analytical model for the cost of evaluating constraints.

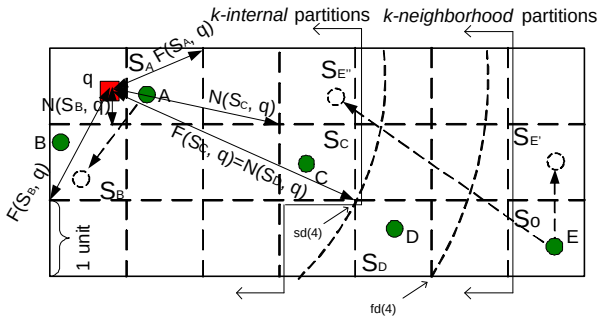


Figure 14: CKNN query

In our model n_c denotes the number of constraints. There are O objects in the 2D plane and each constraint is associated with $\overline{o}$ objects on average. Among the n_c constraints, on average C of them are *uncertain.* The model estimates the matching cost for in-memory and I/O-incurring environments and formalizes algorithmic properties. For a random movement pattern, we derive an estimation for the optimal partition size that results in the least matching cost.

3.1 Cost for In-memory Processing

We assume all the objects update their location with some fixed frequency, f. Since the frequency of the partition adjustment processes is much lower than the location update frequency, the cost for maintaining the index data structure is ignored.

Constraint evaluation cost is comprised of two factors: The cost for evaluating a constraint based on precise location position information (denoted as $Cost_{lu}$) and the partition-based evaluation cost when an object updates its partition (denoted as $Cost_{pu}$). Suppose that for a duration, t, a total of l location updates are received, p of which incur partition updates. For each location update, the system needs to evaluate on average $C\overline{o}/O$ *uncertain* constraints. If each of them takes approximately α processing time, then the cost for a location update during one unit time equals $\frac{(C\overline{o}/O)\alpha l}{t}$. Since $f = \frac{l}{Ot}$, the above location update cost can be expressed as:

$$Cost_{lu} = C\overline{o}\alpha f \tag{1}$$

However, when the object is moving into a new partition then all the constraints it is associated with (on average $n_c\overline{o}/O$) have to be subjected to partition-based evaluations. If partition-based evaluation incurs β processing overhead, the cost of a partition update per unit time is $\frac{(n_c\overline{o}/O)\beta p}{t}$. p/l is a good estimation of the probability of an object updating its partition (P_{pu}), or $P_{pu} = p/l$. The above cost for partition update can be rewritten to:

$$Cost_{pu} = n_c\overline{o}\beta f P_{pu} \tag{2}$$

The value of α and β depend on the processing power and can be estimated experimentally. Putting together Eq. 1 and Eq. 2, the average cost per unit of time for the adaptive algorithm is given as:

$$\begin{aligned} Cost_{adapt} &= Cost_{lu} + Cost_{pu} \\ &= C\overline{o}\alpha f + n_c\overline{o}\beta f P_{pu} \end{aligned} \tag{3}$$

The per unit time cost for the naïve approach is simply the cost for evaluating all constraints:

$$Cost_{naïve} = n_c\overline{o}\alpha f \tag{4}$$

In Eq. 3, if P_{pu} is small then the second term could also be neglected and the cost can be approximated to:

$$Cost_{adapt} \approx C\overline{o}\alpha f = \frac{C}{n_c} Cost_{naïve} \tag{5}$$

Discussion: Our algorithm outperforms the naïve approach when $Cost_{adapt} < Cost_{naïve}$. However, with static space partitioning, this performance point can not be guaranteed. For instance, if the majority of the constraints are *uncertain* or P_{pu} is too high, then the performance of the algorithm is much worse. On the other hand, with adaptive space partitioning, the partition adjustment algorithm tries to reduce P_{pu} and the number of *uncertain* constraints so that an overall performance increase can be expected.

Given a fixed space partitioning, P_{pu} depends only on the movement pattern (e.g., the velocity and the position) of the objects while the constraint evaluation depends on the correlation of the position of objects. There is no direct relationship between P_{pu} and the number of *uncertain* constraints. Also, in general, given the frequency of location sampling, there is no direct dependency between P_{pu} and the partition scheme. For example, even when the space is partitioned sparsely, the object could still generate a large number of partition updates by moving back-and-forth across a splitting line, and conversely, an object could move with fast speed without generating any partition update at all in a densely partitioned region (e.g., it is moving inside a partition).

This makes an analysis to derive the optimal partition scheme difficult. For example, it is not possible to answer the question *"what is the optimal size of the partitions to reduce the evaluation cost to a minimum?"*. More generally, we can not even claim that there is a trade off between the number of *uncertain* constraints and the partition update rate. Reducing *uncertain* constraints by making the partition size smaller may not result in a higher P_{pu}. Therefore, for a system with stringent quality-of-service requirements (e.g., regarding response time and accuracy, for example), a practical way would be to sample a number of partition scheme choices and choose the best one (i.e, the one that leads to smaller $Cost_{adapt}$). This is exactly what our evaluation algorithms does. It samples the partition space by tentatively inserting or deleting partitions through the splitting and merging process. Eventually, it makes the real partition change that will ultimately reduce the overall cost.

Based on this discussion, below, we develop a simpler model to estimate the optimal partition size. This model is based on the following additional assumptions:

1. All objects follow the random movement pattern, with an average speed of v. As a result, the distribution of the objects on the plane is uniform.

2. The partitions are equal-sized squares with the length of each side equal to a, which is much smaller than the extension [2] of the whole space s ($a \ll s$) and the size of the partition is much smaller than the size of the whole space S($a^2 \ll S$).

3. All the constraints have a unique alerting distances, denoted as d ($a \ll d \ll s$).

First, for the n-body constraint, we claim that the size of the diameter of the smallest circle enclosing a set of objects is uniformly distributed. With the uniform distribution, the probability of a constraint being *uncertain* (*satisfied* or *unsatisfied*) can be expressed with the partition size, space size and the alerting distance and therefore, the estimation of the optimal partition size becomes possible. This is stated in the Lemma 1:

Lemma 1. Under random movement of objects in Euclidean 2D space, the distribution of the size of the diameter of the smallest circle enclosing a set of objects is uniform.

Proof (sketch): we only need to prove that the probability that the diameter of the enclosing circle equals to a particular value d_1 is the same as it equals to another particular value d_2. The random movement pattern results in the even distribution of the objects in space. Let's denote the center of the circle (enclosing object i, $1 \leq i \leq n$) with the diameter d_1 as o_1, then we can map the position p_i of each object i to another position p_i' such that $\frac{\overline{p_i' o_1}}{\overline{p_i o_1}} = \frac{d_2}{d_1}$. This is a one to one mapping from one placement of the objects into another placement (shown in Fig. 15). That means if there is a setting of a placement of the objects whose smallest enclosing circle is d_1, we can find another setting of a placement such that the smallest enclosing circle is d_2. Since the objects are uniformly distributed, each placement is equally probable. Therefore the size of the diameter is also equally probable, which implies the uniform distribution of the diameter of the circle. □

Based on Lemma 1 and the aforementioned assumptions, we can estimate the optimal size of the partition that minimizes the evaluation cost. This is stated in the following theorem.

Theorem 1. Under random movement of objects in Euclidean 2D space, the optimal partition size for n-body constraint is $\frac{gv\beta s}{\alpha}$ for some constant g. (recall that α and β are the average cost for evaluation based on precise position and partition information, respectively.)

Proof: Since the length of the edge of the partition is a and the size of the diameter of the smallest enclosing circle is uniformly distributed (according to Lemma 1) over the range [0, s], a constraint is *unsatisfied* with probability $1 - \frac{a\lceil d/a \rceil}{s}$, is *satisfied* with probability $\frac{a\lfloor d/a \rfloor}{s}$ and *uncertain* with probability $\frac{a}{s}$.

With random movement pattern, P_{pu} is a proportional to the speed of the object v and inversely proportional to the length of the edge of the partition a, so P_{pu} can be expressed as $\frac{gv}{a}$ for some constant g. According to Lemma 1, in the random movement pattern, the constraint is *uncertain* with probability $\frac{a}{s}$, therefore, the number of

[2] The extension of the space is defined as the maximum possible distance between two points inside the space. Given the extension of the space s, the size of the space $S = c * s^{dim}$, for some constant c in the dim dimensional space. For instance, for 2D square space, s is the diagonal of the square, therefore $S = s^2/2$; for 2D circular space, s is the diameter of the circle, therefore $S = \pi s^2/4$. Likewise, for 3D cubic space, $S = s^3/3\sqrt{3}$ and for 3D spheral space, $S = 4\pi s^3/3$.

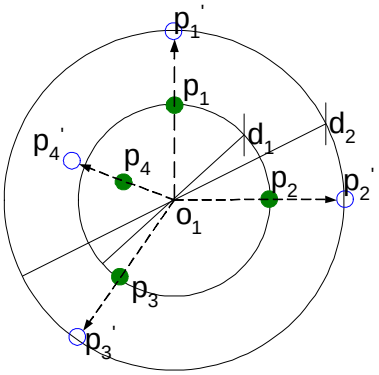


Figure 15: Mapping From one Placement to Another

uncertain constraints $C = n_c \frac{a}{s}$. The cost of the matching algorithm in Eq. 3 can be rewritten as:

$$\begin{aligned} Cost_{adapt} &= C\overline{o}\alpha f + n_c\overline{o}\beta f P_{pu} \\ &= n_c \frac{a}{s}\overline{o}\alpha f + n_c\overline{o}\beta f \frac{gv}{a} \end{aligned} \tag{6}$$

Remember that α and β are the average cost for evalation based on precise position and partition information, respectively. Solving for the optimal length of the edge with $\frac{\partial Cost_{adapt}}{\partial a} = 0$ and get $a^* = \sqrt{\frac{gv\beta s}{\alpha}}$ Notice that C has only one local minimum, therefore it is also the global minimum. The optimal size of the partition is given as

$$a^{*2} = \frac{gv\beta s}{\alpha} \propto v \tag{7}$$

□

First, with uniform distribution, the probability that a constraint is *uncertain* can be estimated. A n-body static constraint is *uncertain* iff there is no object in the external partitions and not all the objects are in the internal partitions. Therefore, the the probability of being *uncertain* is approximated as $(\frac{\pi(d+a)^2}{S})^{\overline{o}} - (\frac{\pi(d-a)^2}{S})^{\overline{o}} = \frac{4da\pi}{S}[(\frac{\pi(d+a)^2}{S})^{\overline{o}-1} + (\frac{\pi(d+a)^2}{S})^{\overline{o}-2}(\frac{\pi(d-a)^2}{S}) + ... + (\frac{\pi(d-a)^2}{S})^{\overline{o}-1}] \approx \frac{4da\pi\overline{o}}{S}(\frac{\pi d^2}{S})^{\overline{o}-1}$. A CRR constraint is *uncertain* iff the object is in the partition that is on the boundary of the query range. Therefore, the probability is $\frac{\overline{p}_q a}{S}$, where $\overline{p}_q$ is the average length of the perimeter of the query range. Finally, a CRKNN constraint is *uncertain* iff the object is either the kth or the $k+1$th nearest neighbor of the static point and kth and the $k+1$th nearest neighbors can not be distinguished from the partitions where they are located. Therefore, this object is about $r = \sqrt{\frac{kS}{O\pi}}$ ($O\frac{\pi r^2}{S} = k$ because the range r includes exactly k of the objects) away from the static point, assuming uniform distribution of the object in random movement. And an object is located in such partitions (with distance r away from the static point) with probability $\frac{\pi(r+a)^2 - \pi(r-a)^2}{S} = \frac{4ra\pi}{S}$. Replacing C in Eq. 3 with this probability times n_c, the optimal size for each of the constraints is

unique and can be computed by solving $\frac{\partial Cost_{adapt}}{\partial a} = 0$, (as in Theorem 1) and they are stated in the following theorems.

Theorem 2. Under random movement of objects in Euclidean 2D space, the optimal partition size for n-body static constraint is $\frac{gv\beta S^{\overline{o}}}{4\alpha d\pi^{\overline{o}} d^{\overline{o}-1}}$ for some constant g.

Theorem 3. Under random movement of objects in Euclidean 2D space, the optimal partition size for CRR constraint is $\frac{gv\beta S}{\alpha \overline{p}_q}$ for some constant g.

Theorem 4. Under random movement of objects in Euclidean 2D space, the optimal partition size for CRKNN constraint is $\frac{gv\beta S}{4\alpha r\pi}$ for some constant g, and $r = \sqrt{\frac{kS}{O\pi}}$.

Apparently, different constraint has different optimal partition size. This make it even more difficult to derive a universal partition scheme because this depends not only on the constraints but also on the percentile distribution of the constraint types, which is different from region to region. This justifies that an optimal global partition scheme is not feasible and dynamic partition adjustment is the only way to solve the problem.

Discussion: If the movement pattern is not random, we can not compute a globally optimal partition size. However, if we interpret a local distribution as random, then there still exists a local optimum, which gives us a locally optimal partitioning. We approximate this local optimum by rescheduling the partitioning in the area that is not yet optimized based on merging and splitting.

We say a partition scheme, ps, is the ancestor of partition scheme, $ps^{'}$, (conversely, $ps^{'}$ is the descendant of ps), if $ps^{'}$ can be reached from ps by adding splitting lines. This descendant relationship is denoted as $ps^{'} \succ ps$. Consider two partition schemes, ps and its descendant $ps^{'}$, then the following lemma states that with the splitting process alone, the number of *unsatisfied* and *satisfied* constraints is non-decreasing and the number of *uncertain* constraint is non-increasing.

Lemma 2: If $ps^{'} \succ ps$, an *unsatisfied* constraint in ps is also an *unsatisfied* constraint in $ps^{'}$; a *satisfied* constraint in ps is also a *satisfied* constraint in $ps^{'}$; an *uncertain* constraint in $ps^{'}$ is also an *uncertain* constraint in ps.

Proof (sketch for n-body constraint case): Let's consider the partition-based evaluation of a constraint $|{p_1}^t, {p_2}^t, ..., {p_n}^t| \leq d$. We denote the smallest circle intersecting the partitions in ps ($ps^{'}$) which contains $p_i (1 \leq i \leq n)$ as $\mathcal{O}$ ($\mathcal{O}^{'}$).

(1)an *unsatisfied* constraint in ps is also an *unsatisfied* constraint in $ps^{'}$: By contradiction, if not, then the diameter of $\mathcal{O}$ is strictly larger than d, and the diameter of $\mathcal{O}^{'}$ is smaller than or equal to d. But since $ps^{'} \succ ps$, the partition containing $p_i (1 \leq i \leq n)$ in $ps^{'}$ is completely covered by the partition containing p_i in ps. Therefore any circle intersecting the former one also intersecting the later one. It can be concluded that $\mathcal{O}^{'}$ intersects all partitions containing p_i in ps and since the diameter of $\mathcal{O}^{'}$ is not larger than the diameter of $\mathcal{O}$, $\mathcal{O}$ is not the smallest intersecting circle, a contradiction.

(2) a *satisfied* constraint in ps is also a *satisfied* constraint in $ps^{'}$: Since $ps^{'} \succ ps$, the partition containing $p_i (1 \leq i \leq n)$ in $ps^{'}$ is completely covered by the partition containing p_i in ps. Therefore, the circle enclosing the partition containing p_i in ps also encloses the partition containing p_i in $ps^{'}$, therefore a "*satisfied*" constraint in ps is also a "*satisfied*" constraint in $ps^{'}$.

(3) Since number of constraints remains the same in two partition schemes, an *uncertain* constraint in $ps^{'}$ is also an *uncertain* constraint in ps. □

Lemma 2 helps to reduce the computational cost in the splitting and merging process as stated in Section 2.2. The splitting process aims at a reduction of the number of *uncertain* constraints at the cost of a small increase of the partition update rate, and conversely, the merging process aims at a reduction of the partition update rate at the cost of a small increase of the number of *uncertain* constraints.

The actual number of partitions affected is tuned according to the movement pattern (e.g. with higher speed, more partitions are expected to adjust their layout and vice versa). Our adaptive algorithms, AKDT and AMLG, adapt to a non-uniform environment and evolve with the change of the movement patterns to reach a locally optimal partitioning. This balances the updates required against the evaluation overhead. In the next section, we will experimentally validate these insights.

3.2 Cost under Secondary Storage Access

For applications that need to manage a large number of moving objects with a large number of constraints, secondary storage access cost of the algorithm may become a performance degrading factor. For instance, objects maybe associated with large profiles, as is common in telecommunications applications, or the application may be collocated with other applications, thus not have exclusive access to main memory. The cost (I/O) for accessing data in secondary storage is typically orders of magnitude larger than accessing data residing in main memory and thus it becomes the major overhead that outweighs anything else. So reducing the number of secondary storage accesses becomes a primary issue in this environment.

Our algorithm is well suited for reducing I/O access to the secondary storage. To show this, we consider a simplified model where the main memory space is restrained only to allow the AKDT and AMLG structure, user's position information and only m of all n_c constraints ($m < n_c$). Other location $n_c - m$ constraints reside in secondary storage and are loaded to memory on demand. In our matching algorithm, for efficiency, we allow as many *uncertain* constraints as possible to be loaded into the main memory, so among those m constraints in memory, $min(C, m)$ are *uncertain* constraints and the rest of $max(C - m, 0)$ *uncertain* constraints are still on the secondary storage. (C is the number of *uncertain* constraints, $C < n_c$) Note, when secondary storage is accessed, data is fetched in pages rather than in records and each access to secondary storage will transfer a constant unit of data into memory. Suppose each page contains at most n_{page} constraints, then each disk access may fetch θ ($1 \leq \theta \leq n_{page}$) constraints. θ depends on how constraints are organized in the disk. In the best case where the constraints are redundantly grouped by the each associated objects, each access may fetch exactly $\theta = n_{page}$ constraints. In the worst case when the constraints are randomly stored, however, it may only fetch one constraint for one page access. The I/O cost is measured as the number of accesses to the data on the secondary storage device. Next, we claim and prove the following theorem about the cost of the space partition algorithm:

Theorem 5. If all the *uncertain* constraints can be hold in the main memory, the cost of the matching algorithm is only P_{pu} times that of the naïve approach.

Proof: When the objects change partitions, the algorithm has to retrieve all the constraints for classification. Assuming that each time exactly one page is loaded into the memory due to the space restrain, then the number of secondary storage access for partition updates per unit time is given as:

$$Cost_{IO}^{pu} = \frac{(n_c - m)\overline{o}p}{O\theta t} = \frac{|n_c - m|\overline{o}fP_{pu}}{\theta} \tag{8}$$

and for the location update, only the disk access for the on-disk *uncertain* constraints is necessary, thus:

$$Cost_{IO}^{lu} = \frac{max(C - m, 0)\overline{o}l}{O\theta t} = \frac{max(C - m, 0)\overline{o}f}{\theta} \tag{9}$$

Therefore the total cost per time unit is:

$$\begin{array}{rcl} Cost_{IO}^{adapt} & = & Cost_{IO}^{pu} + Cost_{IO}^{lu} \\ & = & ((n_c - m)P_{pu} + max(C - m, 0))\frac{\overline{o}f}{\theta} \end{array} \tag{10}$$

With the naïve approach, all the constraints in the disk ($\frac{(n_c - m)\overline{o}}{O}$) need to be fetched out. The cost is:

$$Cost_{IO}^{naïve} = \frac{(n_c - m)\overline{o}l}{O\theta t} = \frac{(n_c - m)\overline{o}f}{\theta} \tag{11}$$

Clearly, when the memory is not sufficient for all the constraints ($n_c > m$) $n_c - m > max(C - m, 0)$ (since $n_c > C$). The space partition approach outperforms the naïve approach when P_{pu} is relatively low (which render $Cost_{IO}^{pu}$ to diminish) or when m is larger than C (which render $Cost_{IO}^{lu}$ to diminish). In the later case where $max(C - m, 0) = 0$,

$$Cost_{IO}^{adapt} = \frac{|n_c - m|\overline{o}fP_{pu}}{\theta} = P_{upd}Cost_{naïve} \tag{12}$$

, which proves the Theorem 2. □

This result is somewhat surprising, it means that even with limited memory, reducing the partition update rate by some proportion causes the evaluation cost to be reduced with the same proportion. So in situations were memory is the limiting factor the system performance can be improved by adapting the space partitioning so as to reduce the partition update rate (given that all *uncertain* constraints can be loaded in memory).

We show experimentally that with adaptive space partitioning, the *uncertain* constraints can be reduced by 30% and the partition update rate can be reduced by 70%. Theoretically, for the random movement pattern, the *uncertain* constraint is only a small portion of the total number of constraints ($\frac{a}{s}$ as stated in Lemma 1) and can be easily fit into the main memory.

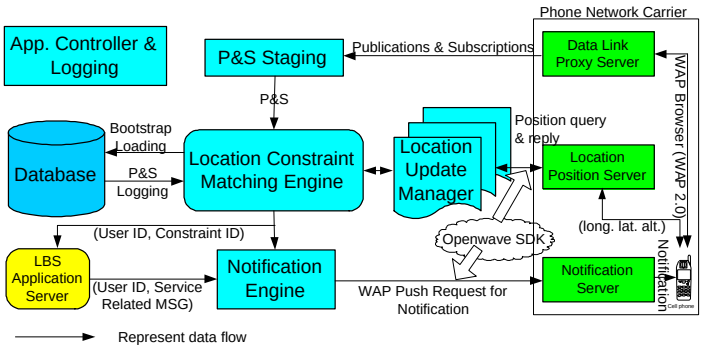


Figure 16: End-to-end System Architecture (fully implemented)

4 System Implementation

We have developed a complete end-to-end location-based service incorporating the location constraint matcher developed in this paper. This was a collaborative project with DigitalWizards, Inc., a Canadian Media and Design company; it tracks the wildlife in Canada and sends the sighting notifications to the nearby users (Fig. 17). The system was deployed as a proof of concept on the Bell Mobility wireless network. We summarize the system's architecture and some of the more practical lessons learned.

The overall system architecture is shown in Fig. 16. In our implementation, the system uses the mobile network to obtain location position information of subscribers. The network exports location tracking capabilities via a Web service. Location position data is retrieved through a location position server over the Internet. Different solutions for obtaining location position data are available in practice. In our case, the operator combined GPS, network triangulation, and cell site location technologies to position subscribers, aiming to provide the most accurate position for a required precision, specified as part of the location request input. Different from the envisioned concept of streaming the location updates to the constraint solver, in our end-to-end system implementation, a location update manager must schedule the location request on behalf of subscribers and pulls the position data from the mobile network carrier's location position server. The pulling frequency is constrained by the performance of executing the request (several seconds for our trials). Moreover, each location request is charged by the network operator. Through the experience of real system, we also find that the accuracy of the location information is very coarse in the bad weather conditions or for indoor environment. The precision is ranging from few meters (sunny, outdoor) to over a hundred meters (rainy) or even above one kilometer (indoor).

5 Experiments

In this section, we present the experimental results that demonstrate the performance of the algorithms. In the experiments we simulate mobile objects moving in a test

Figure 17: GeoTracker Applcation: GUI(left) and Sighting Notification (right)

field of size $40km \times 40km$. We model two movement patterns, random movement and clustered movement. In the random movement pattern, the random direction model is used [12]. This model maintains a uniform distribution of the objects in the test field. In the clustered pattern, the moving objects form groups and leave the clusters with a small probability. Clusters are static or mobile. Fig. 18 illustrates the movement pattern a possible space partitioning.

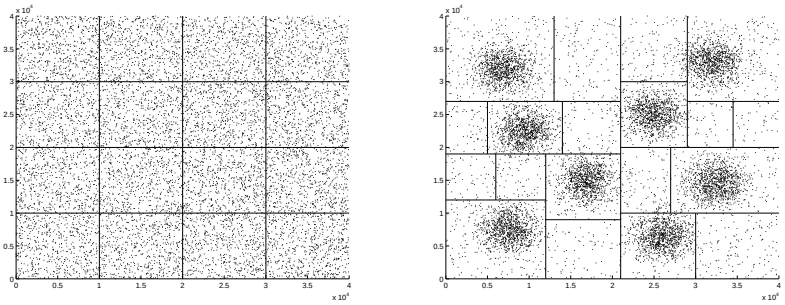


Figure 18: Movement Patterns and the AKDT Space Partitioning

We set the mean of the velocity of the objects to 5m/s, which simulates walking pedestrians, unless otherwise stated. Prior to the experiment, a number of constraints are generated; each constraint is randomly associated with n bodies among a total of 10,000 mobile objects in the field. The alerting distances are uniformly distributed with a certain mean (e.g., 500m).

By changing the mean, the matching load can be adjusted. The *matching load* is defined as the ratio between the average number of satisfied constraints and the total number of constraints.

For simplicity, all moving objects update their location with the same frequency, f (0.5/sec)[3]. We call the time period of $1/f$ seconds one round. In each round, all the moving objects update their location. All constraints are evaluated accordingly and the total matching time is averaged over many rounds. Splitting and merging is carried out once every minute to optimize the partition scheme as needed. The choice of the location update frequency only has an effect on the precision of the evaluation. It has

[3] f is location updates per unit of time.

no influence on the efficiency of the algorithm evaluation.

5.1 Performance Under Random Movement Pattern

This experiment is conducted under random movement pattern where the objects are uniformly distributed. This experiment consists of three groups, with 100,000, 300,000 and 500,000 2-body constraints each. With the adaptive algorithm (both AKDT and AMLG indexing), we measure the average matching time, the number of partition updates and the average number of *uncertain* constraints per round and compare the results to the static index based on the SPKDT tree over different fixed partition sizes in square kilometers (see Fig. 19). To make the figure clear, the result of the naïve approach, which evaluates all the constraints sequentially, is not shown, because its matching time is an order of magnitude higher than that of our matching algorithm.

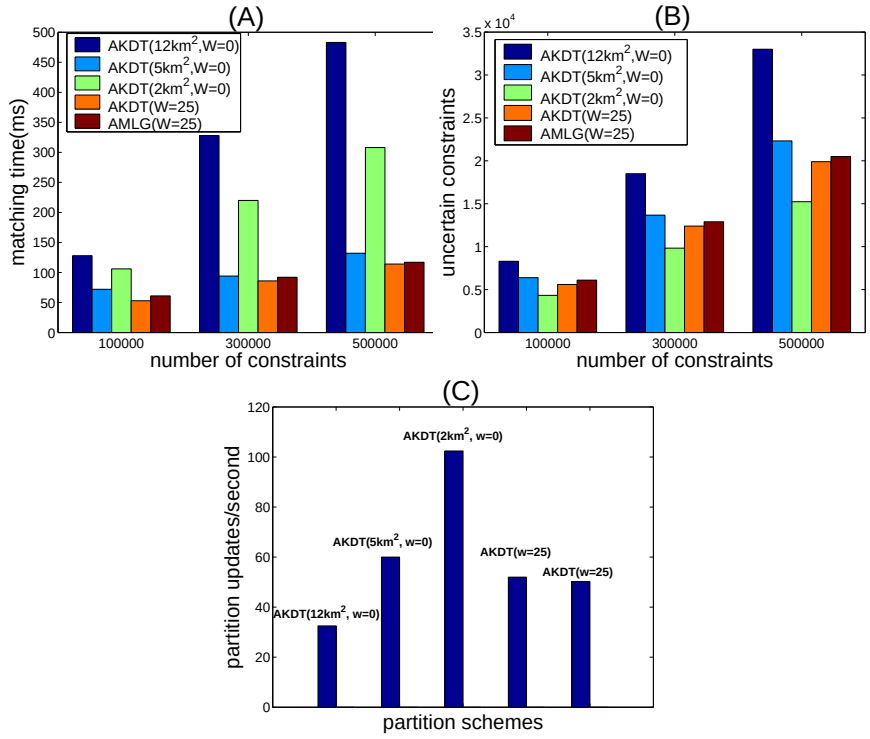


Figure 19: Comparison on Different Indexes

Across those three groups, we observe that, for all adaptive index schemes, the matching time is roughly proportional to the number of *uncertain* constraints (as shown in Eq. 5). The number of *uncertain* constraints evaluated with AKDT is very close to that of AMLG (AKDT's is slightly lower). This means that both space partition scheme work equally well in terms of pruning away the *satisfied* and *unsatisfied* constraints.

Even though the partition computation for AMLG is more efficient than that of AKDT, which has to traverse the tree structure to find the new leaf node, the average matching time for AKDT is comparable to that of AMLG. Clearly, both partition schemes have reduced the partition update rate to the minimum and they no longer make that much a difference to the total cost even though partition computation for AMLG is more efficient. The total matching cost is in fact proportional to the average number of *uncertain* constraints (see Fig. 19(A, B)).

Within each group, we also observe the impact of the partition size on the performance of the matching algorithm with static indexing: The smaller partition size brings about more partition updates (see Fig. 19(C)), thus more time is required for partition-based evaluation. However, it also leads to less *uncertain* constraints that need to be evaluated (see Fig. 19(B)), as predicted by Lemma 2. With partition size $5km^2$, our algorithm performs best and consumes the least amount of matching time. This means that $5km^2$ is close to the optimal partition size given the experimental conditions (as suggested by Theorem 1). Other partition schemes fail because, either the cost of a partition update is too high (for $2km^2$), or the cost for evaluating *uncertain* constraints is too high (for $10km^2$). With the adaptive algorithm, AKDT and AMLG are self-adjusting to the best partition size and achieve the best possible matching efficiency. Although the partitions scheme is evolving with time (due to the on-going adjustment processes), the average partition size stays close to $5km^2$.

Assuming adaptive indexing eventually results in the optimal partition size, we validate the relationship between the average velocity of the objects and the optimal partition size. Fig. 20 shows that the optimal partition size is almost proportional to the average velocity of the objects (as confirmed by Theorem 1).

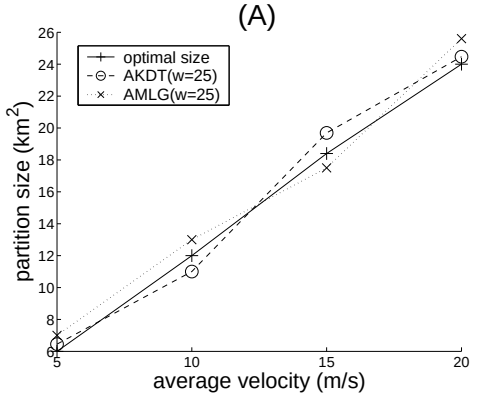


Figure 20: The Effect of the Velocity of the Objects on The Optimal Partition Size

Further experiments with more than two bodies and even with n-body static constraints yield similar results. They are omitted due to space limitations.

5.2 Adaptation to the Change of Movement Patterns

This experiment evaluates how AKDT and AMLG, adapt to movement patterns. The results are compared against non-adaptive solutions to the location constraint matching problem. With 100,000 constraints and 10,000 uniformly distributed objects under the random movement pattern, we evaluate the adaptation of the algorithms over time. We initially set the average partition size to $25km^2$ and measure the matching time and the average partition size. Fig. 21(A,B) show the matching time and partition size during the first 12 minutes of the simulation.

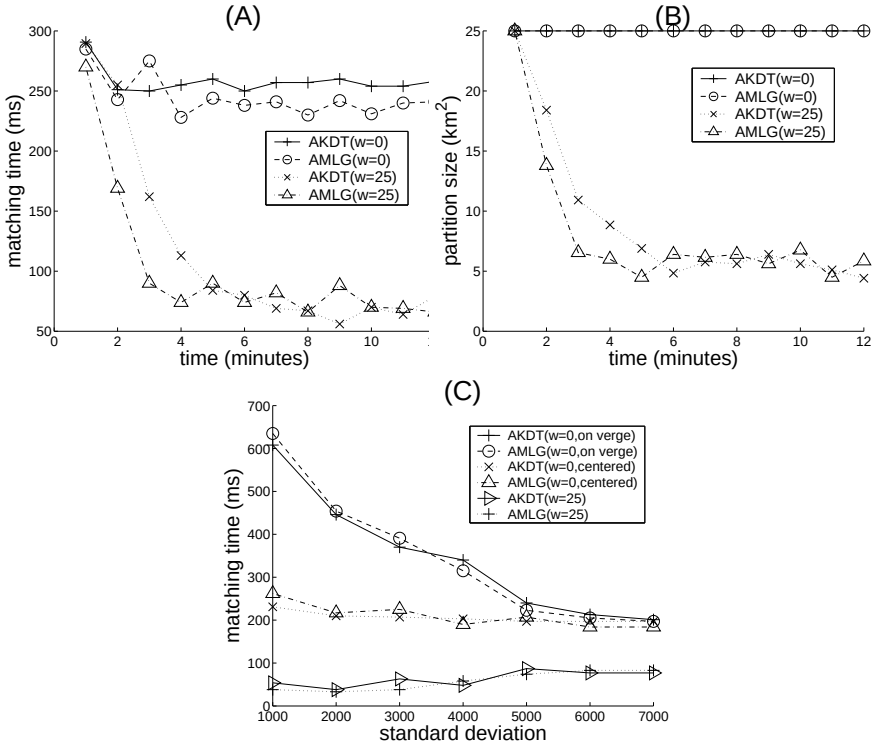


Figure 21: Adaptation to Movement Patterns

Since the SPKDT and Grid indexing are static, they do not evolve at all over time to better accommodate the movement pattern. The matching time for both SPKDT and Grid is around 250ms and the partition size is $25km^2$ (see Fig. 21(A,B)).

AKDT and AMLG actively evolve to reduce, either the evaluation load (by reducing the number of *uncertain* constraints through splits), or the partition update overhead (by reducing the number of partition updates through merges). We see that within seven minutes, the matching time of both AKDT and AMLG is reduced and stabilizes below 100ms and the partition size is adjusted to around $5km^2$. This performance gain comes about through the pruning of large numbers of *uncertain* constraints at the cost

of moderate increases due to partition updates. The adaptive algorithm finds the optimal partition size through self-adjustment and reaches the balance between the number of *uncertain* constraints and the number of partition updates.

In a further experiment, we evaluate the adaptation of the algorithm with the evolution of the movement pattern. We start with a random movement pattern, and the objects gradually form clusters within the next 50 minutes and then the clusters are moving (randomly). We see from Fig. 21(C) that the moving clusters create huge problems for the static indexes. Their matching time is not only high but also very unstable with many spikes resulting from the large partition update overhead when the clusters move crossing the splitting lines. However, the AKDT and AMLG handle this quite well. The matching time is relatively low and much more stable.

5.3 Secondary Storage Access Evaluation

In this experiment, we measure the number of disk access required for 10,000 objects and 100,000 2-body constraints. We assume that the AKDT and the AMLG data structure, the position information of the objects and a certain number of location constraints are residing in memory. Other location constraints have to be swapped in and out of memory on demand. As page replacement strategy, we use the least recently used replacement (LRU) scheme. The size of each page is 4k. We store as many *uncertain* constraints in memory as possible, so that the disk access is already small.

Fig. 22 plots the number of secondary storage accesses against the number of pages that are allowed in memory. For this test, we run the simulation on AKDT and AMLG against objects with various movement patterns and compare them with the non-adaptive SPKDT and Grid-based indexes. Fig. 22 shows that the disk accesses for both adaptive and non-adaptive matching algorithms are much less than that of the naïve approach because of the limited number of constraints accessed, a benefit resulting from the pruning capability of the partitioning scheme. All space partition approaches have a critical point of the page number after which the number of disk accesses is almost proportional to the disk accesses for the naïve approach (around 80 pages for adaptive and 120 pages for non-adaptive). This is the point where all the *uncertain* constraints fit into main memory. The adaptive algorithms (AKDT and AMLG) greatly reduce the *uncertain* constraints (by 30%), therefore less memory pages are required to maintain them.

When the page number exceed this point, the number of disk accesses equals $P_{pu}Cost_{IO}^{naïve}$, as predicted by Theorem 2. The line with the adaptive algorithm is much flatter than the non-adaptive one after the critical point. Based on Theorem 2, we can deduce that the adaptive algorithms have partition update rates that are much lower (about 70% lower in this example) than the non-adaptive algorithms. Therefore, the adaptive algorithms outperform the non-adaptive algorithms in the I/O incurring environments.

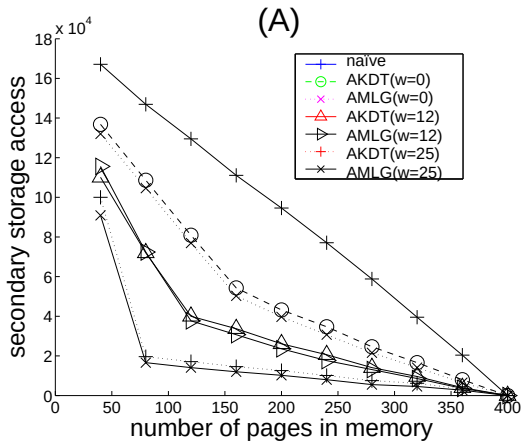


Figure 22: Secondary Storage Access

6 Related Work

Determining the *close-to* relationship among, either a set of mobile, or static entities has received some attention in the literature. Corral *et al.* [5] study the problem of determining the K closest pairs between two spatial sets of static points. Their approach is similar to a join query and discovers the K smallest distances between two sets of points. Different from the location constraint processing, this work does not consider the *close-to* relationship involving more than two objects. Moreover, the approach partitions all objects into two groups and does not consider the distance of objects from the same group.

The nearest neighbor problem determines the nearest object(s) to a given point among all the objects in the space [11, 21]. This is a different query type from the location constraint matching problem that determines whether a specific set of objects are in a given spatial constellation to each other at a given point in time.

The buddy tracking system [15] is the only work known to us that is looking at a problem statement similar to the location constraint matching problem we are stating. The approach is exclusively looking at a 2-body problem in the 2D space. The approach is based on a distributed algorithm for solving the problem. It assumes that the mobile objects communicate with each other directly to resolve the 2-body constraints. The objective is to reduce the communication cost (i.e., the messages exchanged between mobile entities.) A non-distributed quadtree-based algorithm is also sketched, but not evaluated. However, it only works for 2-body constraints. The approach does not support adaptive space partitioning. We therefore expect it to perform similarly to a static space partitioning scheme, which performs poorly under a skewed or clustered movement pattern. Moreover, the solution in the paper is restricted to one global alerting distance for all registered constraints, which is a severe limitation.

In prior work we have build a location-based service [18] processing proximity re-

lations based on the location constraints defined in this paper. We have also evaluated the precision of available location positioning technologies [20] in the same application context. These systems were based on a non-adaptive, static solution to the constraint matching problem [19]. This approach does not adapt with changing movement patterns and requires knowledge of object clusters in advance to setup the data structures. The AKDT and AMLG in this paper address these limitations through a dynamic space partitioning scheme.

The location constraint matching problem also reminds one of the publish/subscribe matching problem [6]. Indeed, location constraints could be interpreted as subscriptions and location updates as publications. However, location constraints are not of the form commonly assumed by publish/subscribe systems, which makes the application of these matching algorithms difficult. An extension of publish/subscribe for processing location-aware data (i.e., notify a subscriber with a matching subscription close to a publishing entity) has been proposed by Burcea and Jacobsen [4].

7 Conclusions and Future Work

Location constraint processing is essential for location based applications that aim at tracking, correlating, and filtering information about moving entities. In this paper, we define two types of location constraints, the n-body constraint and the n-body static constraint, which capture a proximity relation among sets of moving objects. We propose adaptive space partition with AKDT and AMLG index and the constraint matching algorithm to evaluate large sets of location constraints. The dynamic index with AKDT and AMLG partitions the whole space into smaller parts and evolves with the change of the movement pattern of the objects. The distance between the partitions serves as bounds to the distance between objects lying inside these partitions. Using the partition information of the moving objects, only the constraints that are likely to be satisfied (yet uncertain) are chosen for further consideration.

Our experimental results show that the performance of AKDT and AMLG index schemes is very close and for all movement patterns, the matching time is roughly proportional to the number of *uncertain* constraints, which is much less than the naïve approach and non-adaptive approach. We show and experimentally validate that with random movement pattern, there exists optimal space partition, the size of which is proportional to the average velocity of the moving objects. The experiment shows that the AKDT and AMLG adapt themselves to the movement pattern and the partition scheme they produce is very close to the optimal partition one can get manually with non-adaptive index. We further show that our approach is well suited for large constraint loads that go beyond capabilities of main memory processing. When the memory space enough for *uncertain* constraints is available, the number of disk accesses is proportional to the partition update probability. Experimental results also show that comparing with the non-adaptive indexing, the adaptive indexing reduces the number of *uncertain* constraints by 30% and the partition update rate by 70%.

In future work, we intend to model a correlation between the certainty of a constraint match and the projected object positions at time of match and time of notification. That is, besides reporting the constraints that are satisfied, we will also report on

the likely accuracy of the matched constraint at a given point in time. This will provide insights on how to adjust the location update frequency in order to achieve a desired level of accuracy. We also intend to study how to predict possible future matches and matching in the environment with obstacles.

References

[1] New and enhanced features of fedex insight. *http://www.fedex.com/us/*.

[2] Radio frequency identification systems (RFID). *http://www.ti.com/tiris/docs/solutions/animal/livestock.shtml*.

[3] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. In *In Proc. 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2000.

[4] I. Burcea and H. A. Jacobsen. L-ToPSS - Push-oriented Location-based Services. In *4th VLDB Workshop on Technologies for E-Services (TES'03)*, 2003.

[5] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proceedings 1994 ACM SIGMOD Conference, Dallas, TX*, pages 189–200, 2000.

[6] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD*, 2001.

[7] D. R. Karger. Finding Nearest Neighbors in Growth-restricted Metrics. In *ACM Symposium on Theory of Computing (STOC)*, 2002.

[8] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of the 18th ACM PODS Conference*, 1999.

[9] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. In *ACM Trans. Database systems*, 1984.

[10] S. Prabhakar, Y. Xia, D. Kalashnikov, W. A., and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 2002.

[11] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.

[12] E. Royer, P. Melliar-Smith, and L. Moser. An analysis of the optimum node density for ad hoc mobile networks. In *Proceedings of the IEEE International Conference on Communications*, 2001.

[13] 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*, 2000.

[14] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R-Tree: A Dynamic Index for Multi-Dimensional Objects. In *The VLDB Journal*, 1987.

[15] A. Amir. A. Efrat. J. Myllymaki. L. Palaniappan. K. Wampler. Buddy tracking - efficient proximity detection among mobile friends. In *INFOCOM*, 2004.

[16] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. In *ACM Transactions on Information Systems (TOIS) archive*, 1992.

[17] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, LNCS. Springer, 1991.

[18] Z. Xu and H. A. Jacobsen. Efficient constraint processing for highly personalized location based services. In *VLDB04*, 2004.

[19] Z. Xu and H. A. Jacobsen. Efficient constraint processing for location-aware computing. In *6th International Conference on Mobile Data Management (MDM'05), Ayia Napa, Cyprus*, 2005.

[20] Z. Xu and H. A. Jacobsen. L-ToPSS: Constraint processing system supporting efficient location based services. In *The Third International Conference on Mobile Systems, Applications, and Services (MobiSys 2005) Seattle, WA, U.S.A.*, 2005.

[21] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *Proceedings of ICDE*, 2005.

[22] Jun Zhang, Manli Zhu, Dimitris Papadias, Yufei Tao, and Dik Lun Lee. Location-based spatial queries. *In SIGMOD Conference*, 2003.

[23] Y. Zhao. Standardization of mobile phone positioning for 3G systems. *IEEE Communication Magazine*, 2002.