

Expressive Location-based Continuous Query Evaluation With Binary Decision Diagrams

Zhengdao Xu
University of Toronto
zhengdao@cs.toronto.edu

Hans-Arno Jacobsen
University of Toronto
jacobsen@eecg.toronto.edu

Abstract

Many location-based services require rich and expressive query language support for filtering large amounts of information. In prominent location-based services thousands of continuous queries execute concurrently. Parts of these queries may overlap or logically depend on each other suggesting to amortize the execution over shared sub-queries and ordering queries according to their dependency relations. In this paper spatio-temporal queries constitute location constraints monitored by applications. We develop Constraint Combination Binary Decision Diagrams (CCBDDs), an efficient location constraint matching algorithm based on Binary Decision Diagrams. With CCBDDs, redundant computations for shared sub-queries are avoided, and query dependencies are identified and pruned. Empirical results show that the CCBDD structure greatly improves matching performance by eliminating otherwise redundant computation and memory use in the evaluation.

1. Introduction

The advances in location positioning technology [30] and the pervasive presence of wireless networks give rise to an increasing number of applications for location-based services. Due to the large amount of information, such as the continuously changing location position of moving objects, the large amount of interest profiles, dynamically changing and static information about the environment, sophisticated filtering and correlation capabilities are crucial to the success of a middleware platform supporting location-awareness. For many applications, it is often extremely important to specify rich and expressive queries about the moving objects in the environment. Sophisticated filtering relies on expressive query language support to shield the application from too much unnecessary data. This paper is concerned with developing algorithms to enable the use of a rich query language supporting spatio-temporal processing among moving objects. We illustrate our approach based on the location constraints, the continuous spatio-temporal queries we developed [26, 25]. The approach presented in this paper is applicable to many other types of queries, such as the Continuous Range query (CR), the Continuous Reverse Range query (CRR), the Continuous K -Nearest Neighbor query (CKNN), and the Continuous Reverse K -Nearest Neighbor

query (CRKNN) etc.

In the following, we briefly review the notion of location constraints. The n -body constraint and the n -body static constraint. The n -body constraint is of the form $|p_1^t, p_2^t, \dots, p_n^t| < d$ (or $> d$). It is satisfied if the smallest circle enclosing the n moving objects, identified by p_1, p_2, \dots, p_n has a diameter smaller (or larger) than d at 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. The n -body static constraint is of the form $|A, p_1^t, p_2^t, \dots, p_n^t| < d$ (or $> d$). A is the coordinate of some static point. The constraint is satisfied if the n moving objects p_1, p_2, \dots, p_n , are within (or outside) the distance d from the static point A at time t . In the following, we mainly focus on n -body constraints to illustrate our idea. All approaches are also applicable to n -body static constraints as well as the above listed queries.

The conjunction, disjunction and negation of location constraints, referred to as constraint combination, are more expressive than a single constraint, referred to as an elementary constraint. For instance, moving object p_1 wants to be notified if both, moving objects p_2 and p_3 , are close by. This could be modeled by the constraint combination $|p_1, p_2| < d_1 \wedge |p_1, p_3| < d_1$. Remind that the distance between two objects is below some alerting distance is a 2-body constraint. In another example, moving object p_1 requires a notification, if it is close to a static point, A , and another moving object p_2 is also close to (or approaching) A . This could represent a situation where p_1 is a pedestrian trying to catch a bus (p_2) at bus station A , as illustrated in the top left of Fig. 1. This is modeled as constraint combination $c_1 \wedge c_2$ ($c_1 = |A, p_1| < d_1$ and $c_2 = |A, p_2| < d_2$.) To express the event that a group of moving objects p_1, p_2, \dots, p_n are assembling beside a location A , we write $|p_1, p_2, \dots, p_n| < d_1 \wedge |p_i, A| < d_2$ (for some $i, 1 \leq i \leq n$). The condition that p_1 is accompanied by either p_2 or p_3 is specified as $|p_1, p_2| < d_1 \vee |p_1, p_3| < d_1$ and the condition that p_1 is accompanied by neither p_2 nor p_3 is specified as $\neg(|p_1, p_2| < d_1 \vee |p_1, p_3| < d_1)$. The notion of constraint combination extends to other types of continuous spatio-temporal queries. For example, the condition that p_i is inside a certain range R_1 (e.g., a floor), but not in range R_2 (e.g., a specific meeting room) can be expressed as the combination of two continuous range queries, $c_3 \wedge \neg c_4$ ($c_3 = p_i \in R_1$ and $c_4 = p_i \in R_2$), as illustrated in the bottom left of Fig. 1. In real applications, some location constraints are more popular than others, so they may be submitted multiple times by different users. For example, the constraint "Send me a notification when the flight AC37 has arrived at

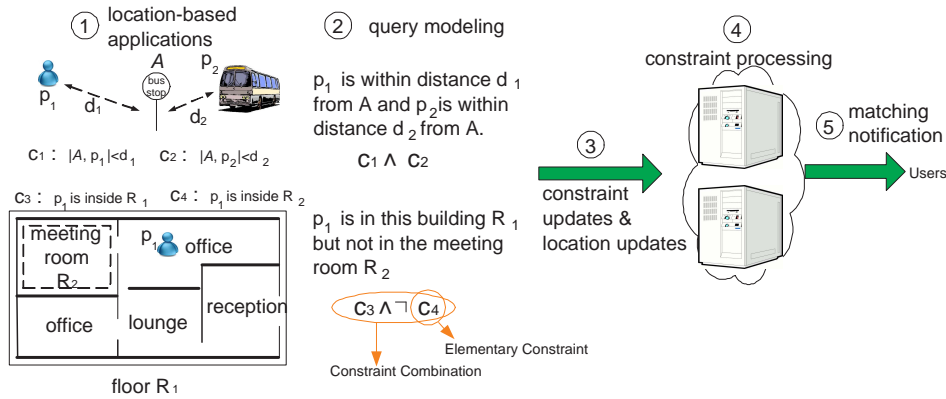


Figure 1. Data Flow for Location Constraint Processing

the Pearson airport.” will be relatively popular among the mobile users. However, this constraint will have to be evaluated multiple times in the traditional location constraint matching engine. Also some constraints may be logically dependent on some other constraints as will be shown shortly, but they are often treated independently. This paper is dedicated to resolve these issues and provides a solution to the efficient processing of constraint combinations.

Location constraints are like continuous queries that once submitted to the system remain active until explicitly revoked. Fig. 1 shows the data flow of a typical location-based service employing the processing of location constraints. First, the location based applications formalize the location constraint and constraint combinations and they are submitted into our system. Location updates of moving objects are streamed into the system and trigger the evaluation of constraints. Matching constraints are communicated back to the interested subscribers. Notice that a efficient partition-based processing algorithm for large amounts of elementary constraints has been proposed in our earlier work [25]. In this paper, we look at how the constraint combinations can be evaluated efficiently. The algorithm in [25] of course can be adopted to enhance efficiency, but this is not the focus of this paper.

Although constraint combinations are more expressive than a single constraint, to the best of our knowledge, there is little work on efficiently processing such kind of location constraint combinations. Existing data management and indexing techniques for moving objects [10, 20, 18, 19, 14, 31, 29, 13, 24, 8] are mainly focusing on the efficient processing of a single spatio-temporal query, such as a range query and the k nearest neighbor query (k NN). In [25], a location constraint matching algorithm supporting the concurrent, continuous evaluation of many elementary location constraints (i.e., hundreds of thousands of location constraints) is developed. This work addresses the problem of evaluating large numbers of elementary constraints based on different space partitioning techniques. It, however, does not address the problem of efficiently processing constraints combinations, which is the subject of this paper. Moreover, the prior work lacks to address how to use historical information of constraint processing (e.g., which elementary constraint appears more often or is frequently evaluated and matched) and dependency relationships among constraints to optimize processing, which are techniques

developed in this paper.

In this paper, we develop algorithms for the efficient evaluation of location constraint combinations. Our approach is based on Binary Decision Diagrams (BDD). We extend BDDs to accommodate the location constraints underlying our query language and develop the Constraint Combination Binary Decision Diagrams (CCBDDs). The CCBDDs data structure exploits the shared execution by aggregate location constraints in the system, so that the redundant computation for repetitive elementary constraint is avoided. We also show the rules for identifying the dependent constraints and skip the unnecessary computation with transition links that infers the dependency relationship between the constraints. To efficiently identify the dependency relationship of large amounts of dynamically changing constraints, a lattice structure is constructed and updated as the constraint set evolves over time. Also, we extend queries to CRR and $CRKNN$ queries, and even the queries that returns a non-Boolean result, such as CR and $CKNN$.

Through experimental evaluation, we show that CCBDDs are efficient both in terms of time and space complexity. Dependent constraint pruning further improves the system performance. We also take historical information of the constraint processing (constraint occurrence, evaluation frequency etc.) into account to further amortize the matching cost. The performance gain can be achieved by placing the popular elementary constraints in favorable variable ordering. With slight overuse of memory, the lattice structure greatly reduce the time required for identifying the dependency relationship, so that the system can cope with frequent update of constraints while still processing huge amounts of existing constraints.

We start by providing background information on BDDs in Section 2. In Section 3, we develop the location constraint matching algorithm based on CCBDDs. In Section 4, we introduce dependency rules among location constraints and present a lattice structure to efficiently manage dependency relationships. The description of the system architecture of our research prototype can be found in Section 5. Section 6 presents the experimental evaluation of our approach. We put our work in perspective to related approaches in Section 7.

2. Background of Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are abstract representations

of Boolean functions [2]. BDDs represent Boolean functions as rooted, directed acyclic graphs (DAG). BDDs simplify many Boolean operations in functions. BDDs are mainly used in various numerical computations [7, 12, 17].

Boolean functions are logic expressions based on predicates and operators. There are two prominent normal forms: Conjunctive Normal Form (CNF)¹ and Disjunctive Normal Form (DNF)². When a computational problem can be specified in terms of Boolean functions, it can take advantage of the BDD-based algebraic operations to determine satisfiability³, equivalence⁴ and tautology⁵.

There are a few variants of BDDs. We adopt three of them for our approach and they are Ordered BDDs (OBDD), Reduced Ordered BDDs (ROBDD) and Shared BDD (SBDD) [9, 4].

Each node in a BDD has two outgoing edges. This binary representation of the node is based on the if-then-else (ITE) operation. Each non-terminal node v of a BDD has two children: $low(v)$ if the variable v is assigned 0, and $high(v)$ if v is assigned 1. A terminal node is denoted by a rectangular box, labeled either 0 or 1. Each node represents a Boolean function $f = v' \cdot low(v) + v \cdot high(v)$, denoted as $ITE(v, low(v), high(v))$, that is $ITE(v, low(v), high(v)) = v' \cdot low(v) + v \cdot high(v)$. In the following illustrations, we use a solid arrow to represent the "then" edge (1), and use a dashed arrow to represent the "else" edge (0).

Any Boolean expression can be decomposed to elements that can be expressed with the ITE operation. This is based on Shannon's decomposition theorem: $f = v \cdot f_v + v' \cdot f_{v'}$, where f_v and $f_{v'}$ are f evaluated at $v = 1$ and $v = 0$, respectively. v is the top variable in f . The following rules can be extracted based on Shannon's decomposition and applied to decompose an expression: (1) $ab = ITE(a, b, 0)$, (2) $a + b = ITE(a, 1, b)$, (3) $a = ITE(a, 1, 0)$ and (4) $a' = ITE(a, 0, 1)$. The operation can be applied recursively until the function is fully decomposed. With the above rules, the expression $f(a, b, c, d) = ab + cd$ can be decomposed recursively as follows:

$$\begin{aligned} f(a, b, c, d) &= ab + cd \\ &= ITE(ab, 1, cd) // rule 2 \\ &= ITE(ITE(a, b, 0), 1, cd) // rule 1 \\ &= ITE(ITE(a, b, 0), 1, ITE(c, d, 0)) // rule 1 \\ &= ITE(ITE(a, ITE(b, 1, 0), 0), 1, ITE(c, ITE(d, 1, 0), 0)) // rule 3 \end{aligned}$$

The BDD for the expression can now be easily determined by starting with the inner most ITE expression since these will be at the bottom of the graph, and moving up to the outermost ITE expression which is the root of the graph. In this example, b , c and d are at the bottom and a becomes the root (Fig. 2(A)).

The BDD representation of the Boolean function allows the variables of the function to appear on any level of the graph. The level of the graph where a certain variable appears can be fixed (with the root at level 0). An OBDD is a BDD with the input variable of the functions ordered and the BDD is built and traversed in ascending variable order (the variables appear at fixed levels of

the graph.) The labeled values of non-terminal nodes indicate the variable ordering with respect to \prec . Fig. 2(B) shows a BDD with variable ordering $a \prec b \prec c$.

A ROBDD is the same as an OBDD with the additional restriction that there is no duplicate nodes in the diagram. This ensures that ROBDDs are a canonical form representations of Boolean functions⁶. To avoid duplication of a node, the nodes representing the same constraint are merged and the internal node with identical children is shortcut (the incoming nodes must be redirected to its child). Fig. 2(C) shows the difference between an OBDD and a ROBDD in representing the expression $f = a + b$. As we can see, b on the right side of OBDD is shortcut in ROBDD.

When manipulating multiple functions, all functions can be represented using a single multi-rooted diagram, which is called Shared BDD (SBDD). SBDDs provide a more compact representation of the functions by removing redundant nodes common to the functions. In addition to the space cost improvement, sharing sub-functions also reduces the time cost for operations performed on the diagram. Fig. 2(D) shows a SBDD representing two Boolean expressions $f = b + c$ and $g = a + b + c$. To evaluate f and g , the shared nodes b and c are computed only once. Notice that the sharing may not always be possible. This will be discussed in Section 3

BDDs allow for efficient and rapid complementation of functions. In order to complement a function, all that is required is to attach a negation attribute to the function node. Also, if a complement is used, only one constant node is required. By using regular and complement in such ways, it is possible to use De Morgan's laws with little memory or performance impact. This allows for the use of both CNF and DNF and the easy switch between them.

3. BDDs-based Modeling of Constraint Combinations

In the BDD-based constraint processing, each elementary constraint is assigned a Boolean variable, which is a symbol representing the result of the elementary constraint. A constraint combination is expressed using a BDD as a Boolean function of those variables based on a predefined order. From the BDD optimizations mentioned in the previous section, we adopt the techniques of node sharing (from SBDD), and duplicate node removal and variable ordering (from OBDD and ROBDD), respectively. To avoid dependent constraint computation, a transition link is used to connect dependent variables (nodes) in different constraint combinations. We call the data structure thus derived the Constraint Combination BDD (CCBDD). It is explained in detail below.

3.1 Constraint Sharing

In our implementation, each constraint combination is represented by one BDD. BDDs are the basic components of the CCBDD. Like SBDD, several BDDs in a CCBDD can share nodes for same common elementary constraint. When the BDD has the same nodes as the newly inserted BDD, those nodes are reused. By sharing the common elementary constraints in constructing BDDs, the constraints are stored more compactly. Also the matching algorithm can reuse the evaluation result of the nodes that are shared by several BDDs. Redundant computations can thus be eliminated. For example, when some object updates its location, all the elementary constraints associated with the object and all the constraint combinations based on these elementary constraints need

⁶This means that a given Boolean function can only be represented in one way using ROBDD

¹the Boolean functions are expressed as a product of sums

²the Boolean functions are expressed as a sum of products

³determine whether a Boolean function evaluates to 1 (true) for at least one truth assignment

⁴determine whether two expressions denote the same function

⁵determine whether an expression evaluates to 1 (true) for all assignments

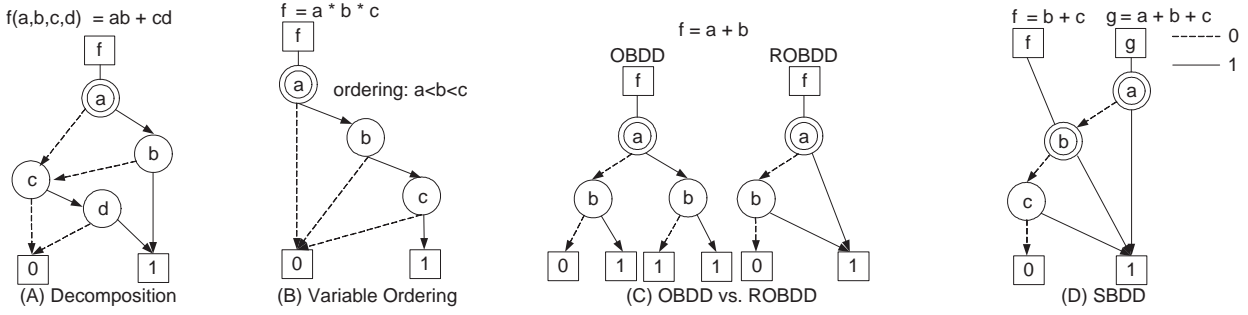


Figure 2. Expression Decomposition (A), Variable Ordering (B), BDD Variations (C) (D)

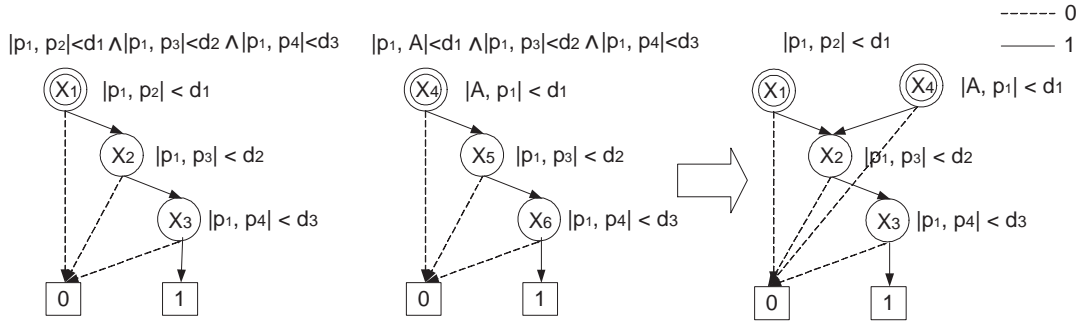


Figure 3. Two BDDs Are Combined

to be reevaluated. However, our CCBDD insures that the each elementary constraint is evaluated only once.

Fig. 3 shows, on the left side, the BDDs for two constraint combinations, $|p_1, p_2| < d_1 \wedge |p_1, p_3| < d_2 \wedge |p_1, p_4| < d_3$ and $|A, p_1| < d_1 \wedge |p_1, p_3| < d_2 \wedge |p_1, p_4| < d_3$. On the right side is the combined BDD with two output nodes. In the combined BDD, the internal node X_2 is reused for X_5 and X_3 is reused for X_6 since the same elementary constraints are represented. During the evaluation (e.g., when p_3 updates its location), the node X_2 for constraint $|p_1, p_3| < d_2$ is only evaluated once.

Note that variable order in the BDD is important, because the sharing of one node implies the sharing of all the sub-branches (down to the terminal nodes) rooted at that node. If the elementary constraints in Fig. 3 were not ordered the same way, e.g., order $X_1 < X_2 < X_3$ for the first constraint combination and $X_4 < X_6 < X_5$ for the second constraint combination, then the sharing would not be possible, because the elementary constraints $|p_1, p_3| < d_2$ and $|p_1, p_4| < d_3$ are reversed in the second BDD. We will discuss how the ordering is determine in the following subsection.

Since the number of distinct constraints in the constraint processing engine is extremely large, a hash index for managing the elementary constraints is used. Multi-level hashing is based on the object set involved in the constraint (sorted by object id), the operators ($>$, $<$...) and the alerting distances. This makes the search more efficient. When a new constraint is submitted into the system, it is compared with the constraints that are already in the system. If the constraint is not already there, it is inserted into the hashtable. The mapping from the hashtable to the BDDs (representing constraint combinations) is also constructed. Fig. 4 shows

the hashtable along with the mapping to the BDDs.

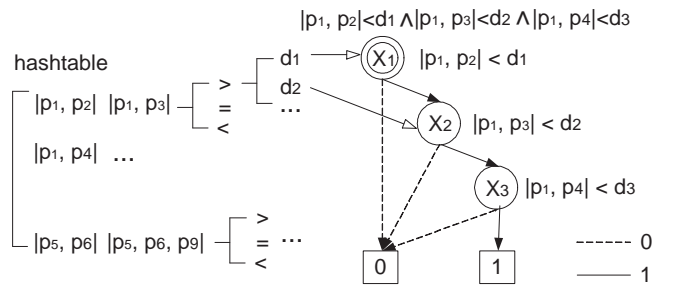


Figure 4. Boolean Variable Index

3.2 Variable Ordering

In order to share the common nodes, the variables in BDDs have to be ordered, but determining the ordering of the variable is difficult. Improper ordering makes node sharing impossible. For example, in the variable ordering $X_1 < X_2 < X_3 < X_4$, the sharing between combination $X_1 \wedge X_2 \wedge X_3$ and combination $X_2 \wedge X_3 \wedge X_4$ is not possible. Although the common nodes X_2 and X_3 are in same oder, but variable X_4 succeeding X_3 encumbers the sharing, because the sharing of X_2 and X_3 implies the sharing of node X_4 below them, but X_4 is not in the first constraint combination, see Fig. 5.

Although the variable ordering problem is NP-complete [3],

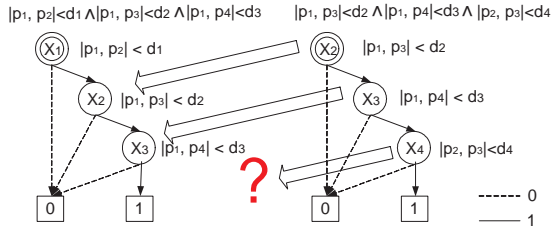


Figure 5. Impossible Node Sharing

various heuristic approaches can be used to order elementary constraints. A reasonable variable ordering may result in more constraint sharing and can avoid the exponential growth of the number of nodes in a CCBDDs. First, the ordering can be based on the number of the occurrences of elementary constraints in constraint combinations. The higher the occurrences, the more beneficial it would be if the node would be shared. Therefore, it should be assigned a higher rank. The second ordering is based on the frequency of the evaluation of elementary constraints. For a particular application, the frequency of the location updates of the objects can be analyzed based on historical data. The elementary constraint involving these objects are also evaluated with the same frequency, triggered by the location updates. In our implementation, we use a combined approach, giving priority to the constraint with higher occurrences. When two constraints have the same occurrence, the one with higher evaluation frequency gets a higher rank. The elementary constraint with higher occurrences or with higher evaluation frequency is placed at the bottom of the BDDs so that it is more likely this elementary constraint is shared with other constraint combinations, because no other node are appended beneath.

3.3 Garbage Collection for Constraint Update

Since the constraints are continuously updated (with insertion and deletion), an efficient garbage collection algorithm should be used to dynamically manage the size of the CCBDDs. Each node added to the diagram has a reference count associated with it. Whenever the node is referenced (e.g., if a new constraint combination is added), the count is increased by 1. When a constraint combination is removed from the graph, the reference count of all nodes associated with the constraint combination is decreased by 1 accordingly. Nodes with reference counts of 0 are dead nodes. Garbage collection removes dead nodes to free up memory. The problem with garbage collection is oscillation, where a node keeps being removed and added. It arises if a node occurs frequently between constraint combinations, and constraint combinations are added and removed often. To cope with this problem, the garbage collection can be set at a certain threshold (when the number of dead nodes reaches a certain number), in order to reduce unnecessary creation and freeing up of similar nodes. If the nodes have not yet been garbage collected, then they can be revived at minimal cost since they already exist but are not being used. This saves the overhead of having to reallocate memory and setting up all the information necessary to manage the nodes.

Another problem with frequent constraint updating is how to efficiently identify the dependency relationship between newly inserted constraints and the constraints that are already in the system. This is discussed in Section 4.

3.4 Extension to Other Spatio-temporal Queries

It is not difficult to observe that all the above mentioned techniques can be extended to and readily applied to other types of continuous spatio-temporal queries. Examples of such queries include the Continuous Reverse Range query (*CRR*) and the Continuous Reverse *K*-Nearest Neighbor query (*CRKNN*), which are defined as follows:

1. Given a rectangular region R , a Continuous Reverse Range query, $CRR(R, p_i)$ (or $p_i \in R$) continuously checks whether a moving object p_i is inside R .
2. Given a static point A , a Continuous Reverse *K*-Nearest Neighbor query, $CRKNN(A, k, p_i)$, continuously checks whether A is one of the k nearest neighbors of moving object p_i .

The detailed algorithm solving *CRR* and *CRKNN* can be found in [25]. Just as the location constraint mentioned before, the *CRR* and *CRKNN* queries are modeled as nodes in the CCBDDs structure. Different query combinations share the same elementary query that is repeated in different query combinations.

BDDs are also used for manipulation of non-Boolean functions (i.e., polynomial functions). For these functions, a node in the BDD represents the partial (non-Boolean) result of the function for the subgraph rooted at that node. The CCBDDs we propose can also be similarly extended to evaluate the queries that returns a non-Boolean value. We only study one of the simplest of standard non-Boolean queries, the Continuous Range query (*CR*). But it can be easily applied to other queries such as the Continuous *K* Nearest Neighbor query (*CKNN*). The definitions of these queries are as follows.

1. Given a rectangular region R , a Continuous Range query, $CR(R)$ continuously returns the moving objects that are inside R .
2. Given a point A , a Continuous *K*-Nearest Neighbor query, $CKNN(A, k)$, continuously returns the k nearest moving objects to A .

Note that *CR* and *CKNN* differ from the traditional range query and the k nearest neighbor query in that *CR* and *CKNN* incrementally report the update of the result set. For instance, at some time point t , $CR(R)$ may report that point p_1 , which was not previously in the range R at time point $t - 1$, is now in the range R (called positive update, denoted as $CR(R)^t = p_1^+$), or point p_2 , which was originally in the result set, is now moving out of range R (called negative update, denoted as $CR(R)^t = p_1^-$).

The CCBDDs can improve the evaluation of combination of such queries by reusing the result of the common elementary queries. To exemplify, the query combination here could be "return the 3 nearest ambulances to the location of the accident A inside region R " ($CKNN(A, 3) \wedge CR(R)$). Another query combination could be "return the 2 nearest ambulances to another location B inside region R " ($CKNN(B, 2) \wedge CR(R)$). Then the result of $CR(R)$ in both combinations can be reused. That is the result of elementary query (BDD node) from different query combinations is reused for different query combinations.

4. Dependency of Elementary Constraints

Location constraints may logically depend on each other. The matching of one constraint may imply that some other constraints are matched as well. Realizing implication relationships among constraints can therefore speed up evaluation considerably.

4.1 Dependency Relations

In this section we identify several dependency relations among constraints. For instance, if $|p_1, p_2, p_3| < d$ is *satisfied*, then it follows that $|p_1, p_2| < d$ is also *satisfied*. If we assume constraint c_1 is associated with object set P_1 and constraint c_2 is associated with object set P_2 . The dependent relation for n -body constraints can be identified with a set of rules summarized in Table 1. The first and second column in the table are the description of the constraints, the third column is the conditions that must be met before the dependent relation can be identified. The last column specifies the transit condition between the two constraints. For instance, the relation among $|p_1, p_2, p_3| < d$ and $|p_1, p_2| < d$ is identified with the rule in the first row, because they have the same alerting distance and the object set $\{p_1, p_2, p_3\}$ is a superset of $\{p_1, p_2\}$.

Table 1. Dependency Rules

c_1	c_2	conditions	transit condition
$P_1 < d_1$	$P_2 < d_2$	$d_1 \leq d_2, P_1 \supseteq P_2$	c_1 is <i>satisfied</i> \rightarrow c_2 is <i>satisfied</i>
$P_1 > d_1$	$P_2 > d_2$	$d_1 \geq d_2, P_1 \subseteq P_2$	c_2 is <i>satisfied</i> \rightarrow c_1 is <i>satisfied</i>
$P_1 < d_1$	$P_2 > d_2$	$d_1 \leq d_2, P_1 \supseteq P_2$	c_1 is <i>satisfied</i> \rightarrow c_2 is <i>unsatisfied</i>
$P_1 > d_1$	$P_2 < d_2$	$d_1 \geq d_2, P_1 \subseteq P_2$	c_2 is <i>unsatisfied</i> \rightarrow c_1 is <i>unsatisfied</i>
$P_1 > d_1$	$P_2 < d_2$	$d_1 < d_2, P_1 \supseteq P_2$	c_1 is <i>unsatisfied</i> \rightarrow c_2 is <i>satisfied</i>
$P_1 < d_1$	$P_2 > d_2$	$d_1 > d_2, P_1 \subseteq P_2$	c_2 is <i>satisfied</i> \rightarrow c_1 is <i>unsatisfied</i>
$P_1 < d_1$	$P_2 < d_2$	$d_1 \geq d_2, P_1 \subseteq P_2$	c_1 is <i>unsatisfied</i> \rightarrow c_2 is <i>unsatisfied</i>
$P_1 > d_1$	$P_2 > d_2$	$d_1 \leq d_2, P_1 \supseteq P_2$	c_2 is <i>unsatisfied</i> \rightarrow c_1 is <i>unsatisfied</i>

In CCBDDs, dependent elementary constraints are modeled with node coverage. When the dependent constraints are identified, a transition link is added between dependent constraints. The transition is triggered when the object updating the location is involved in the constraints of both nodes. Fig. 6 shows two BDDs (constraint combinations) that do not share any node. However, the node X_6 is depending on X_3 , therefore, a transition link is created pointing from X_3 to X_6 . The transition condition is that if X_3 is *satisfied*, X_6 is also *satisfied*. Also in order to transit, the object updating the location should be in both object sets of X_3 and X_6 (e.g., either p_1 or p_4). That is if the current location update is coming from p_1 , the result of X_6 is updated (given X_3 is *satisfied*), but if the location update is coming from p_5 , the result of X_6 does not need to be updated, because it does not affect the result of X_6 .

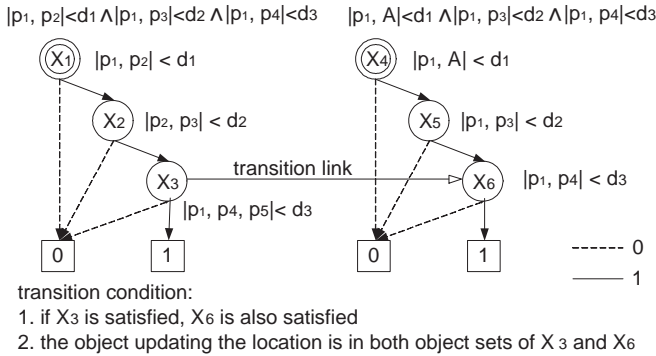


Figure 6. Transition Link

Similar to the query that returns a Boolean value, the result update of one non-Boolean query may be dependent on the result up-

date of some other query. Taking the continuous range query for example, suppose range R_1 completely encloses R_2 and p_1 was originally inside R_2 , now if we know that p_1 is moving out of R_1 , it is of course out of R_2 . Therefore if $CR(R_1)^t = p_1^-$, then $CR(R_2)^t = p_1^-$. Conversely, if p_2 was originally outside R_1 , now if we know that it is moving into R_2 , then it is also inside R_1 . In the case where R_1 and R_2 are overlapping and the intersection region is $R_{1 \cap 2}$, then the event of some object entering $R_{1 \cap 2}$ implies that the object satisfies both range queries.

4.2 Efficient Discovery of the Dependency

In real-world applications, constraints are continuously inserted into and relinquished from the system. An important question is how to quickly identify the dependent relation between the newly inserted constraint and the constraints that are already in the system. From the dependency rules in Table 1, two constraints may have the dependent relationship only if the object set of one constraint is a superset or subset of the object set of another constraint. Once we find those constraints, the dependent relationship will be identified in the refinement step that follows. The problem is how to efficiently determine which constraint has the object set that is the superset or subset of the object set of the new constraint. This problem is non-trivial, because a simple linear scan through all the constraints is very expensive for large constraint loads. Clearly the object sets of constraints are subsets of all the objects registered in the system. Different sets have the partially ordered relationship (or poset). We model the partially ordered relationship with lattice structure. A lattice is a partially ordered set in which every pair of elements has a unique supremum (the elements' least upper bound) and an infimum (greatest lower bound). Lattices can also be characterized as algebraic structures satisfying certain axiomatic identities. An example of the lattice is the positive integers under the operations of taking the greatest common divisor and least common multiple, with divisibility as the order relation ($a < b$ if a divides b .) The lattice structure keeps the containment relation of object sets. When a new constraint is inserted into the system, the object set o of that constraint has to be inserted into the lattice. The breadth first search starts from the top and ends when no node downwards encloses o completely. The nodes beneath the currently visited node (where the insertion stops) represent the object set that is completely disjoint with o , or partially overlapping o , or is the subset of o (assuming object set o has not been inserted before.) The currently visited node actually represents the tightest superset (immediate supremum/superset) of o along that branch. Then, a new node representing o is created; and it is connected with links upwards to the node of the immediate superset and downwards to the node that represents the subset (immediate infimum/subset) of o (see the algorithm `Lattice_Insert` for detail.) And all the supersets or subsets can be achieved by tracing upward from the immediate supremum, or downward from the immediate infimum. As we show experimentally, identifying the superset and subset with the lattice structure is much faster than a linear scan of all the constraints.

Figure 7 shows a lattice structure before and after the object sets $\{p_2, p_3, p_4\}$ and $\{p_2, p_4\}$ are inserted. When set $\{p_2, p_3, p_4\}$ is inserted, the original link from $\{p_2, p_3\}$ to $\{p_1, p_2, p_3, p_4\}$ is removed because the new set broke the immediate containment relation between them. Then the new links are added from $\{p_2, p_3\}$ to $\{p_2, p_3, p_4\}$ and from $\{p_2, p_3, p_4\}$ to $\{p_1, p_2, p_3, p_4\}$. Notice that the actual link in our implementation is bi-directional, and the di-

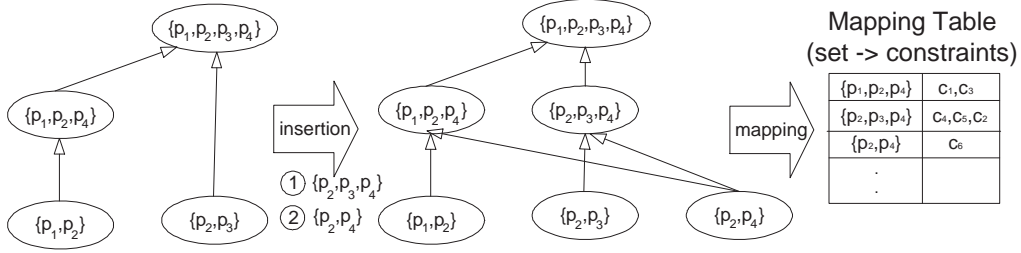


Figure 7. The Lattice Structure of Object Sets

rection of the link in the figure only designates the containment relationship. Each node in the lattice maintains a counter that counts how many constraints are now involved with this object set. After the new node is inserted, a mapping record that associates the object set with the constraints that involve this set is created in the table. However, if the node in the lattice already exists, then only the counter needs to be increased by 1 and the mapping record needs to be updated. During the constraint removal, the counter for the corresponding node is decreased by 1. And a node is removed if the counter equals 0, indicating no constraint is associated with the node.

Algorithm *Lattice.Insert(ObjectSet o, Current r)*

1. create a node n representing o ;
2. $children =$ all children of r ;
3. **if** object set of $r \supset o$
4. **for** the child $r_{chd} \in children$
5. Lattice.Insert(o, r_{chd});
6. **else if** object set of $r \subset o$
7. father of r becomes n 's father;
8. r becomes n 's child;
9. the link between r and its original father is removed;
10. **else**
11. father of r becomes n 's father;

After the insertion, the immediate supremums and infimums of the object set are available to construct the dependent relation between the constraints. To exemplify, in Figure 7, suppose that the constraint $c_4(|p_2, p_3, p_4| < d)$ is already in the mapping table and the constraint $c_6(|p_2, p_4| < d)$ is now inserted into the matching engine. After the update of the lattice structure, one of the supremum of $\{p_2, p_4\}$ is identified as $\{p_2, p_3, p_4\}$, which further maps to the constraint c_4 and a dependent relation is immediately constructed according to the rule in the first row of Table 1 in the refinement step that follows.

5. System Implementation

We have developed the constraint combination processing algorithm embedded in our research prototype called Location-based Toronto Publish/Subscribe System (L-ToPSS). More information about an earlier version of the system prototype is summarized in [27].

The system was deployed as a proof of concept on a mobile cellular network. The overall system architecture is shown in Fig. 8. In our implementation, the system uses the cellular network to obtain location position information of the subscribers. The network exports location tracking capabilities via a Web service. The users can submit or update their queries (constraint combinations in this context) along with the desired notifications (text messages) from

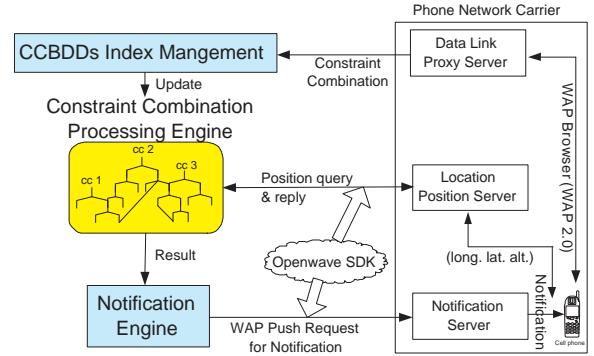


Figure 8. End-to-end System Architecture (fully implemented)

our web page. For experimental purposes, the constraint combinations can also be submitted in batches. These queries are then indexed with the CCBDDs structure by the CCBDDs index management component.

Our system retrieves the location position data through the location position server over the Internet. Both the server and the subscriber can initiate position requests. When the location updates stream into the system, they are processed by the constraint combination processing algorithm, which traverses the CCBDDs structure for the query evaluation, and the matches of the constraint combinations are communicated back to the subscribers via the notification server on the carrier's side. The wireless operator combines GPS, network triangulation, and cell site location technologies to position subscribers. A brief evaluation of the accuracy of the system under different experimental conditions is summarized in [26].

6. Experiments

In this section we evaluate the performance of the constraint processing with CCBDDs. Our implementation of the matching engine is based on the JavaBDD package [1]. In the experiment, the constraint combinations are generated and associated with 5,000 objects moving with an average velocity of 10m/s. On average, a constraint combination consists of 5 elementary constraints, each one associated with n objects, where $n = 4$, unless otherwise specified. To construct the constraint combinations, we first generate an elementary constraint pool with adjustable size. The elementary constraints are generated such that they contain a pre-

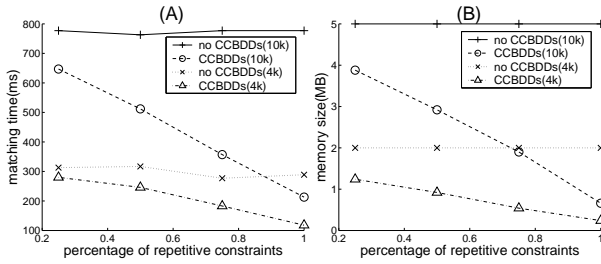


Figure 9. Repetitive Constraint Processing

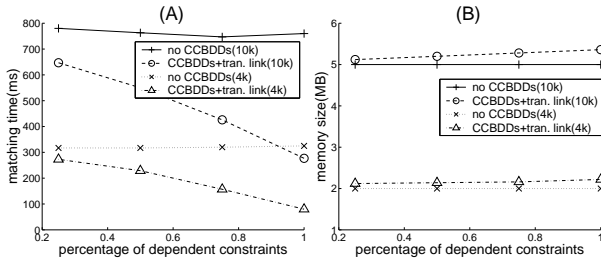


Figure 10. Dependent Constraint Processing

determined percentage, α , of dependent constraints. The elementary constraints among the constraint combinations are randomly drawn from an elementary constraint pool, since the size of the pool can be adjusted, the percentage, β , of the repetitive elementary constraints (the elementary constraints that are exactly the same) in the constraint combinations can be precisely controlled. In reality, constraints involving fewer objects consume less memory. But for the sake of analysis, we store each constraint in the structure with a fixed size of 0.1kB. In the experiment, we measure the processing time and the memory consumption with or without CCBDDs structure. Notice that each location update from a moving object triggers the evaluation of constraint combinations associated with that object. The processing time is defined as the CPU time for processing the constraint combinations when all the objects update their location once (in sequel). All experiments were conducted on a Pentium 4 with 2.13GHz CPU and 2G RAM running under Linux OS.

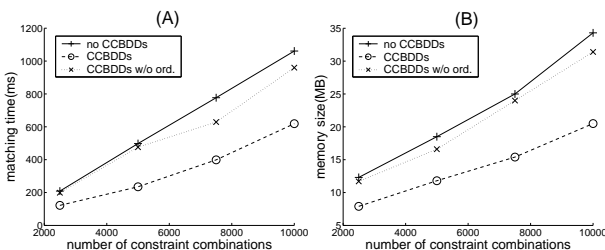


Figure 11. Variable Ordering in BDDs

In the first experiment, we increase the percentage of repetitive elementary constraints and measure the matching time and memory consumption for two constraint loads, 4k and 10k constraint

combinations, respectively. As can be observed from Fig. 9, the matching time without CCBDDs structure is fix at around 300ms (for 4k) and 800ms (for 10k), respectively. However, with the CCBDDs structure, the repetitive constraints are represented as the same node, and therefore are evaluated only once, the matching time decreases linearly as the percentage of repetitive constraints, β , increases (A). Likewise, due to the reuse of the CCBDDs nodes for the repetitive constraints, the memory size to store all the constraint combinations also decreases linearly as repetition increases, while it has no effect on the memory size when CCBDDs is not used (B).

In Fig. 10, we show that using transition links to prune the computation of dependent elementary constraints has a similar effect on the processing time as the pruning of the repetitive constraints. The matching time decreases linearly as the percentage of dependent constraints, α , increases (A). But since transition links incur additional storage, the CCBDDs structure plus transition links requires more memory (B).

For the above experiments, we have applied the variable ordering technique mentioned in Section 3. The elementary constraint rank is based on popularity, popular constraint (higher occurrences and evaluation frequency) with higher rank, and less popular constraint (lower occurrences and low evaluation frequency) with lower rank. Higher ranked constraints are translated into nodes at the relatively lower level of BDDs, so they get a higher chance to be reused by other BDDs. Now, we compare against the case where the variable ordering heuristic is not applied. Fig. 11 (with $\beta=40\%$ $\alpha=0$) shows that CCBDDs without variable ordering heuristic achieves only slight performance gain over the baseline where no CCBDDs structure is used, in terms of processing speed (A) and memory use (B), because many nodes are left unshared. This means CCBDDs has almost no effect when variable ordering is not applied. Therefore, in our experiment, ordering is always used together with CCBDDs without specifying.

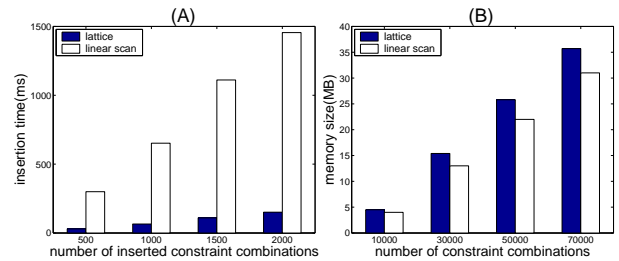


Figure 12. Effect of Lattice

In Fig. 7, we show the processing cost for establishing the dependent relation when new constraints are inserted. Here, 25% of the elementary constraints are dependent ($\alpha=25\%$). The processing time is proportional to the number of constraint combinations inserted (A) and the size of the lattice depends on the number of constraints already maintained in the system (B). Therefore, as the input constraint combinations or the existing constraints increases, the time for insertion and memory consumption also increase, respectively. From Fig. 12, We observe that even though the lattice structure incurs 10% more memory consumption (B), the processing time to identify the dependent constraints is reduced to 10% of the cost of processing without lattice structure (A).

In the following experiment, we test the processing time and

memory space consumption for the *CRR* and *CRKNN* queries. The experimental setting is exactly the same as before, except that the elementary constraints are replaced with elementary *CRR* or *CRKNN* ($k=3$) queries, and in total we generate 10k query combinations, each with 5 elementary constraints. Fig. 13 shows the saving in matching time (A) and memory use (B) when the repetition of constraints increases. Also when CCBDDs structure is applied to non-Boolean queries, such as *CR* and *CKNN* ($k=3$), a similar trend is observed in Fig. 14. Due to the space limitation, we do not further elaborate this.

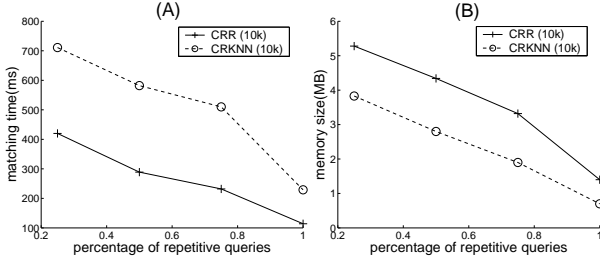


Figure 13. Repetitive Query Processing (CRR,CRKNN)

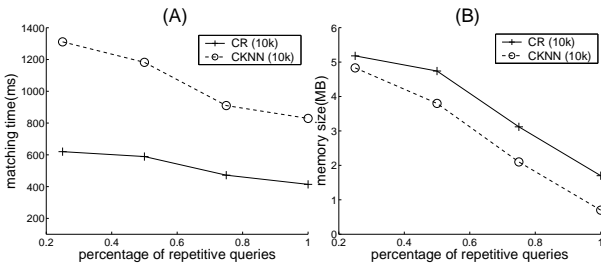


Figure 14. Repetitive Query Processing (CR,CKNN)

7. Related Work

Determining the geometric relationship among a set of moving objects in the Euclidean space has received wide attention in the literature. Spatio-temporal queries in Euclidean space include k closest pairs [6], the (k) nearest neighbor [31, 16], continuous range query [23], and buddy tracking [21]. All these above mentioned approaches study the efficient processing of a single query. The Scalable INcremental hash-based Algorithm (SINA) [13] studies how to efficiently process a large amount of spatial queries. It achieves scalability with shared execution, which performs the spatial join between a set of moving objects and a set of queries. The efficiency is achieved with incremental evaluation, which computing only the updates of the previously reported answer. Their work in [24] detailed and compared three query join policies, and a hot join policy which is enhanced from the incremental join policy is shown to be the best policy because it incorporate the use of No-Action region. Their work, although targeting

at single queries of a large amount, does not consider the evaluation of query combinations. The work [8] studies the evaluation of continuous constraint queries (CCQ) over data streams. The CCQ is modeled with Constraint Satisfaction Problems (CSPs) and the solution they propose minimizes the memory consumption and is suitable for data stream at high rate. Their work, however, does not exploit the common components in the combinations to reduce processing overhead.

Our prior work [27, 28, 25] studies the processing of proximity relations of n bodies in the 2D Euclidean space, and in [26] we evaluate the precision of available location positioning technologies to offer a solution for constraint evaluation under position uncertainty. None of these work discuss the processing of constraint combinations. Although the combination of continuous queries exhibits enhanced expressiveness, as to the best of our knowledge, there is no existing work studying how to efficiently process query combinations that group elementary queries with conjunction, disjunction and negation.

BDDs and variants of BDDs have been used widely in many fields such as digital system design, finite-state system analysis, and artificial intelligence. BDDs have also been used for formal verification [5] and pointer alias analysis in Java [22]. Multi-terminal BDDs (MTBDDs) are suggested as representation of matrices. This allows the efficient processing of term-wise, row, column, block, and diagonal selection over matrices, such as required by Strassen matrix multiplication and LU factorization [7]. The multi-valued decision diagrams (MDDs) support fast discrete function evaluation [12]. When extending BDDs by allowing values from an arbitrary finite domain to be associated with terminal nodes, algebraic decision diagrams (ADDs) are applicable to shortest path algorithms and linear algebra [15]. Zero-suppressed BDDs (ZBDDs) are applied to reduce the computational complexity of clustering techniques for analyzing gene expression data [17]. BDDs have also been applied to the publish/subscribe matching problem [11] by representing predicates in subscriptions as variables, and modeling subscriptions as Boolean functions. All these applications exploit the shared execution to reduce the computing effort, however, none of them targets at the shared execution of database queries, letting alone the combination of spatio-temporal queries.

8. Conclusions

In this paper, we introduce algorithms based on BDDs for the efficient processing of combinations of elementary location constraints. The combinations constitute more expressive queries than a single elementary constraint. Under large query loads, it is conceivable that many of the elementary constraints involved in a combination are redundant or overlapping. To amortize the processing cost, we propose the CCBDDs to index constraint combinations so that the repetitive constraints are not being stored and evaluated multiple times. This greatly improves the system performance in terms of processing speed and memory use. With transition links across nodes, the CCBDDs structure successfully prunes the computation of dependent constraints. Variable ordering based on the popularity of constraints speeds up the matching process by enhancing the chance of node reuse. Incremental maintenance of an additional lattice structure allows us to determine the dependent constraints of the newly inserted constraints efficiently.

Our experimental evaluation shows that when the percentage of overlapping constraints is high (e.g. 50%), the CCBDDs struc-

ture dramatically reduces the processing time (in our experiments by up to 35%) and memory use is reduced by up to 40%. Furthermore, we find that an advantageous variable ordering must be used to make CCBDDs effective. Moreover, with 10% more memory used, the lattice structure helps to identify dependent constraint ten times quicker than without the lattice.

9. REFERENCES

- [1] Javabdd package. <http://javabdd.sourceforge.net>.
- [2] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 1978.
- [3] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 1996.
- [4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.
- [5] Randal E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of Conference on Computer-Aided Design (ICCAD '95)*, 1995.
- [6] 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.
- [7] M. Fujita, P.C. McGeer, and J.C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Formal Methods in System Design*, 2004.
- [8] Marios Hadjieleftheriou, Nikos Mamoulis, and Yufei Tao. Continuous constraint query evaluation for spatiotemporal streams. In *International Symposium on Spatial and Temporal Databases (SSTD 2007)*.
- [9] Shin ichi Minato. Binary decision diagrams and applications for vlsi cad. *Springer; 1 edition*, 2001.
- [10] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of the 18th ACM PODS Conference*, 1999.
- [11] G. Li, S. Hou, and H.-A. Jacobsen. A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems based on Modified Binary Decision Diagrams. In *Proceedings of ICDCS*, 2005.
- [12] Patrick C. McGeer, Kenneth L. McMillan, Alexander Saldanha, Alberto L. Sangiovanni-Vincentelli, and Patrick Scaglia. Fast discrete function evaluation using decision diagrams. In *Proceedings of International Conference on Computer Aided Design*, 1995.
- [13] M. Mokbel, X. Xiong, and W. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *International Conference on Management of Data (SIGMOD 2004)*.
- [14] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 2002.
- [15] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE/ACM International Conference on CAD*, 1993.
- [16] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.
- [17] S. Yoon and G. De Micheli. An application of zero-suppressed binary decision diagrams to clustering analysis of DNA microarray data. In *International Conference of Engineering in Medicine and Biology Society, 2004*, 2004.
- [18] 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.
- [19] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R-Tree: A Dynamic Index for Multi-Dimensional Objects. In *The VLDB Journal*, 1987.
- [20] G. Trajcevski, O. Wolfson, K. Hinrichs, and S. Chamberlain. Managing uncertainty in moving objects databases. In *ACM Transactions on Database Systems (TODS)*, 2004.
- [21] A. Amir, A. Efrat, J. Myllymaki, L. Palaniappan, K. Wampler. Buddy tracking - efficient proximity detection among mobile friends. In *INFOCOM*, 2004.
- [22] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of Programming Language Design and Implementation*, 2004.
- [23] Kun-Lung Wu, Shyh-Kwei Chen, and Philip S. Yu. Efficient Processing of Continual Range Queries for Location-Aware Mobile Services. In *Information Systems Frontiers*, 2005.
- [24] Xiaopeng Xiong, Mohamed F. Mokbel, Walid G. Aref, Susanne E. Hambrusch, and Sunil Prabhakar. Scalable spatio-temporal continuous query processing for location-aware services. In *International Conference on Scientific and Statistical Database Management (SSDBM 2004)*.
- [25] Z. Xu and H. A. Jacobsen. Adaptive location constraint processing. In *International Conference on Management of Data (SIGMOD 2007), Beijing, China*.
- [26] Z. Xu and H. A. Jacobsen. Evaluating proximity relations under uncertainty. In *IEEE 23rd International Conference on Data Engineering (ICDE 2007)*.
- [27] Z. Xu and H. A. Jacobsen. Efficient constraint processing for highly personalized location based services. In *VLDB04*, 2004.
- [28] 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.
- [29] Jun Zhang, Manli Zhu, Dimitris Papadias, Yufei Tao, and Dik Lun Lee. Location-based spatial queries. In *SIGMOD Conference*, 2003.
- [30] Y. Zhao. Standardization of mobile phone positioning for 3G systems. *IEEE Communication Magazine*, 2002.
- [31] Baihua Zheng, Jianliang Xu, Wang chien Lee, and Dik Lun Lee. Grid-Partition Index: A Hybrid Approach to Nearest-Neighbor Queries in Wireless Location-Based Services. In *The VLDB Journal*, 2006.