Expressive Location-based Continuous Query Evaluation With Binary Decision Diagrams

Zhengdao Xu and Hans-Arno Jacobsen
zhengdao@cs.toronto.edu, jacobsen@eecg.toronto.edu

I. Introduction

The advances in location positioning technology and the pervasive presence of wireless networks give rise to an increasing number of location-aware applications. Due to the large amount of information, such as the continuously changing location position of moving objects, the large number of subscriber profiles, dynamic 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. The queries we consider monitor *location constraints*. A location constraint represents a proximity relation among moving objects and among moving and static objects.

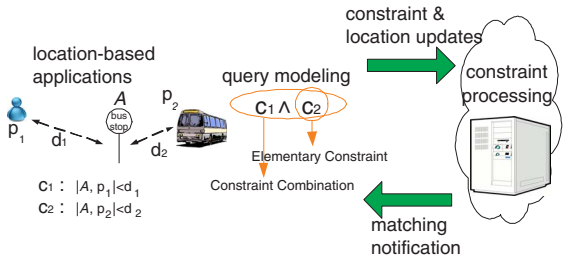

Fig. 1. Data Flow for Constraint Processing

Location constraint processing is like continuous query processing; once the location constrain is submitted to the system, it remains active until explicitly revoked. Fig. 1 shows the data flow of a typical location-based service employing location constraint processing as envisioned in this work. Applications register location constraints with the system, which processes location updates as input. Location updates that represent the movement of objects are streamed into the system and trigger the evaluation of all location constraints stored with the system. Matching constraints are communicated back to interested subscribers.

In the following, we briefly review the notation for representing *location constraints*, such as the *n-body constraint* and the *n-body static constraint*, previously introduced in [6]. The *n-body constraint* is of the form $|{p_1}^t, {p_2}^t, ..., {p_n}^t| < d$ (or $> d$.) It is *satisfied* if the smallest circle enclosing the n moving objects, identified by $p_1, p_2, ..., p_n$ has a diameter smaller (or larger) than d at time t. The position of object i for $1 \leq i \leq n$ is designated by p_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 $|A, {p_1}^t, {p_2}^t, ..., {p_n}^t| < d$ (or $> d$). A is the coordinate of some static point. The constraint is *satisfied* if the n moving objects $p_1, p_2, ..., p_n$, are within (or outside) 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.

We refer to the conjunction, disjunction and negation of location constraints, as a *constraint combination*. Combinations are more expressive than a single constraint, referred to as an *elementary constraint*. For instance, 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 A (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. We modeled this as the conjunction of two 1-body static constraints $c_1 \wedge c_2$ ($c_1 = |A, p_1| < d_1$ and $c_2 = |A, p_2| < d_2$.)

In many applications constraint combinations share popular elementary constraints. For example, in the query "*Send a notification when flight AC37 arrived at Pearson airport*," the result is popular among users who are waiting for the flight at the airport. The result of the constraints may logically dependent on each other. However, the underlying constraint would have to be evaluated multiple times for different users unless shared constraint represen-

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.189

tation and evaluation is possible.

Although constraint combinations are more expressive than a single constraint, to the best of our knowledge, there is no work on efficiently processing such kind of location constraint combinations. Existing data management and indexing techniques for moving objects [3], [4], [5] are mainly focusing on the efficient processing of a single spatio-temporal query, such as a range query and the k nearest neighbor query (kNN). In our earlier work [7], [6], a location constraint matching algorithm supporting the concurrent, continuous evaluation of many elementary location constraints (i.e., millions of location constraints) is developed. This work addresses the problem of evaluating large numbers of elementary constraints based on different space partitioning techniques. However, it 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 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 Diagram (BDD). We extend BDD to accommodate the location constraints underlying our query language. Our data structure exploits the shared execution by aggregating location constraints in the system, so that the redundant computation for repetitive elementary constraint is avoided. We also study how unnecessary computation can be skipped with transition links that infers the dependency relationship between the constraints.

II. Background on BDDs

A Binary Decision Diagram (BDD) is an abstract representation of a Boolean function [2]. BDDs represent Boolean functions as rooted, directed acyclic graphs (DAG). A BDD simplifies many Boolean operations on functions.

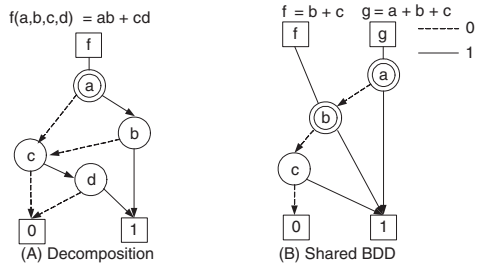

Fig. 2. Decomposition (A), Shared BDD (B)

Boolean functions are logic expressions based on predicates and operators. When a problem can be specified in terms of Boolean functions, it can take advantage of the BDD-based algebraic operations to determine satisfiability, equivalence and tautology.

The Boolean function can be expressed by the BDD. A BDD is a directed acyclic graph (DAG), which consists of two type of nodes, the non-terminal nodes and terminal nodes (see (Fig. 2(A)) as an example). Each non-terminal node v, expressed as a circle, has two outgoing edges, representing an if-then-else (ITE) operation. The outgoing edges are pointing at two children of v, f_v if the variable v is assigned 1, and $f_{v'}$ if v is assigned 0. Node v represents a partial Boolean expression rooted at node v, $ITE(v, f_v, f_{v'}) = f = v \cdot f_v + v' \cdot f_{v'}$. The root node (double circle) of the BDD represents the result of the whole function. A terminal node is denoted by a rectangular box, labeled either 0 or 1, and the terminal node represents the possible Boolean value a non-terminal node may be assigned.

The BDD for boolean expression can be 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 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). Fig. 2(A) shows the BDD of function $f(a, b, c, d) = ab + cd$. Notice that in this BDD, b, c and d are at the bottom and a becomes the root.

III. Indexing Constraint Combinations

In this section, we elaborate on how a BDD can be used to model constraint combinations. To share the execution of the elementary constraints, we further optimize the traditional BDD to include techniques of variable ordering, duplication removal and node sharing. We call the derived data structure the *Constraint Combination BDD* (CCBDD). It is explained in detail below.

A. Indexing With a BDD

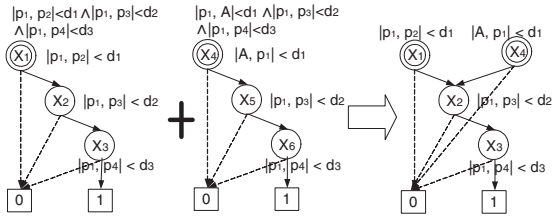

Fig. 3. Two BDDs Are Combined

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. The variable (elementary constraint) is represented in the

BDD as a node. Constraint combinations are regarded as Boolean operations on these variables. With Shannon's decomposition theorem, all the constraint combinations can be decomposed into if-then-else structures and modeled by the BDDs. As one example, suppose we have elementary constraints a, b, c and d, where $a : |p_1, p_2| < d_1$, $b : |p_1, p_3| < d_2$, $c : |p_1, p_4| < d_3$, $d : |p_1, p_5| < d_4$, then the logic of either a and b, or c and d is modeled as constraint combination $ab + cd$ and can be expressed with a BDD as in Fig. 2(A).

To serve the purpose of shared execution, we adopt some optimization based on the traditional BDD. When manipulating multiple functions, all functions can be represented using a single multi-rooted diagram, which is called a Shared BDD (SBDD). An SBDD provides 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(B) shows an 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.

A BDD allows 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 (negation) 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 to easily switch between both.

B. Constraint Sharing

When aforementioned optimizations are applied to process the constraint combinations, the resulting data structure is called the Constraint Combination BDD (CCBDD). In our implementation, each constraint combination is represented by one BDD, with variables (elementary constraints) ordered and reduced. BBDs are the basic components of the CCBDD. Several BDDs in a CCBDD can share nodes for the 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 to be reevaluated. However, our CCBDD ensures that 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, X_1 and X_4 ($X_1 : |p_1, p_2| < d_1$, $X_4 : |A, p_1| < d_1$). 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, i.e., when p_3 updates its location, the node X_2 for constraint $|p_1, p_3| < d_2$ is only evaluated once.

IV. Dependency of Elementary Constraints

Location constraints may logically depend on each other. The result of some constraint may be derived from the result of some other constraint that is already evaluated.

For instance, if $|p_1, p_2, p_3| < d$ is *satisfied*, then it follows that $|p_1, p_2| < d$ is also *satisfied*. Realizing implication relationships among constraints can therefore speed up evaluation considerably.

In CCBDD, 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. 4 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 or p_4, 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.

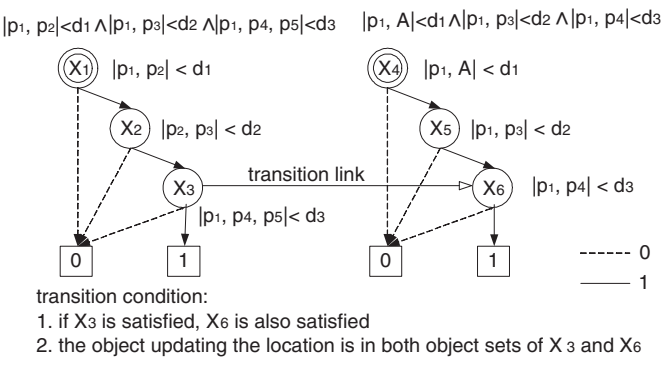

Fig. 4. Transition Link

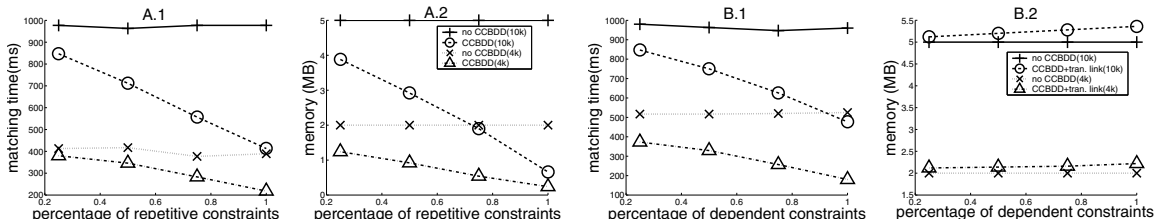

Fig. 5. Repetitive & Dependent Constraint Processing

V. Evaluation

In this section we provide an evaluation of the performance of constraint processing with CCBDDs. Our implementation is based on the JavaBDD package [1]. In our experiments, the constraint combinations are generated and associated with 20,000 objects moving with velocity uniformly distributed in the range 10-50m/s. On average, a constraint combination consists of 5 elementary constraints, each associated with 4 objects. To construct the constraint combinations, the elementary constraints are generated such that they contain a predetermined percentage of the repetitive elementary constraints. In the experiments, we measure the processing time and the memory consumption with and without CCBDD structure.

In the experiments, we increase the percentage of repetitive elementary constraints and measure the matching time and memory consumption for two constraint loads, with 4k and 10k constraint combinations, respectively. As can be observed from Fig. 5(A), the matching time without CCBDD is fix at around 400ms (for 4k) and 1000ms (for 10k), respectively. However, with the CCBDD, the repetitive elementary constraints are represented as the same node, and are therefore evaluated only once; the matching time decreases linearly as the percentage of repetitive constraints increases (A.1). Likewise, due to the reuse of the CCBDD nodes for the repetitive elementary constraints, the amount of memory to store all constraint combinations decreases linearly as the number of redundant elementary constraints increases, while it has no effect on the memory use when CCBDD is not used (A.2).

In Fig. 5(B), 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 increases (B.1). However, since transition links incur additional storage, the CCBDD structure plus transition links actually requires about 5% more memory (B.2).

VI. Conclusions

In this paper, we introduced algorithms based on a Binary Decision Diagram for the efficient processing of combinations of elementary location constraints. The combinations constitute more expressive spatio-temporal queries than a single elementary constraint alone. Under large query loads, it is conceivable that many of the elementary constraints involved in a combination are redundant, dependent or overlapped. To amortize processing, we propose the CCBDD to index constraint combinations so that repetitive constraints are not being stored and evaluated multiple times. This significantly improves system performance in terms of processing speed and memory use; both are reduced in proportion to the percentage of repeated elementary constraints. Moreover, with transition links across nodes, the CCBDD structure additionally prunes the computation of dependent constraints. Our experimental evaluation shows that when the percentage of overlapping constraints is high (e.g., 50%), the CCBDD structure dramatically reduces the processing time by up to 35%, and the memory use is reduced by up to 40%.

References

[1] Javabdd package. *http://javabdd.sourceforge.net*.
[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.
[3] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of the 18th ACM PODS Conference*, 1999.
[4] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.
[5] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R-Tree: A Dynamic Index for Multi-Dimensional Objects. In *The VLDB Journal*, 1987.
[6] Z. Xu and H.-A. Jacobsen. Adaptive location constraint processing. In *SIGMOD, 2007*.
[7] Z. Xu and H.-A. Jacobsen. Evaluating proximity relations under uncertainty. In *ICDE, 2007*.