Querying by Spatial Structure

Dimitris Papadias[1], Nikos Mamoulis[1] and Vasilis Delis[2]

[1]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{dimitris, mamoulis}@cs.ust.hk

[2]Computer Engineering and Informatics Department
and Computer Technology Institute
University of Patras, Greece
delis@cti.gr

Abstract: Structural queries constitute a special form of content-based retrieval where the user specifies a set of spatial constraints among query variables and searches for all configurations of actual objects that (totally or partially) match these constraints. Processing of such queries can be thought of as a general form of spatial joins, i.e., instead of pairs, the result consists of n-tuples of objects, where *n* is the number of query variables. In this paper we propose a flexible framework which permits the representation of configurations in different resolution levels and supports the automatic derivation of similarity measures. We subsequently describe three algorithms for structural query processing which integrate constraint satisfaction with spatial indexing. For each algorithm we apply several optimization techniques and experimentally evaluate performance using real data.

Abstract Tracking Number: 628

Correspondence should be addressed to Dimitris Papadias.

Tel: ++852-2358-8218, Fax: ++852-2358-6971

1. INTRODUCTION

Several types of spatial queries have been the focus of active research in the database community: window queries [G84], nearest neighbors [RVK95], relation-based queries [PTSE95] etc. The above types retrieve all objects in the database that satisfy some spatial property with respect to a fixed reference object or window. Recently the focus has shifted towards spatial joins [R91][G93][BKS93], which involve the retrieval of pairs of objects that satisfy some spatial predicate (most often *overlap*). In general, proposed methods for the above queries can be classified in two categories: new indexing structures (e.g., [S90] [ELS95]), or optimization techniques that improve performance of standard access methods for the queries of interest (e.g., [HJR97]).

This work examines an alternative form of spatial information processing, namely, queries involving the retrieval of n-tuples ($n>2$) objects that satisfy some spatial structure. Structure can be described as a set of spatial constraints between query variables which can be expressed either by a "verbal" (e.g., *select* X, Y, Z *where* overlaps(X,Y) and north(Y,Z)) or pictorial language (e.g., by drawing a prototype configuration on a sketch-board). This type of queries can be thought of as the generalization of spatial joins (if the relation between all pairs of query variables is *overlap* it corresponds to traditional nested spatial join). From a different perspective, structural queries constitute a special class of image similarity retrieval, where similarity is based on relative locations and not on visual characteristics (e.g., colour, shape).

Let n be the query size (number of variables) and N be the data size (number of image objects): in the worst case (exhaustive search), all n-permutations of N objects have to be searched in order to find solutions (i.e., $N!/(N-n)!$). Because in real DBMSs N>>n, $N!/(N-n)! \approx N^n$, meaning that the retrieval of structural queries can be exponential to the query size. Query processing becomes more expensive if partial matches are to be retrieved, a situation which arises very often in practical applications. In order to avoid this problem, previous work on structural queries has focused on a specific instance of the problem where images are re-arrangements of the same set of objects. [CSY87] and [LH92] use symbolic object *projections* encoded in 2D strings and string matching algorithms to retrieve images that match direction relations (e.g., *north*) among a set of variables. [NAS96] describe another projection-based technique that applies *conceptual neighborhoods* [H84] to define structural similarity. [GR95] use the angles between centroids to retrieve images that contain all object configurations that satisfy some angular relations, e.g., "find all images where object A is northeast (i.e., the angle between the centroids is 45^{o}) of object B etc." (A, B, … are specific objects that exist in all images).

[PF97] move a step further and employ spatial indexing to solve structural queries for images that contain a constant number of expected objects (e.g., lungs) and a small number of unlabelled ones (e.g., tumours). They map each image onto a point in multi-dimensional space, where each dimension corresponds to a relation between a specific pair of objects (the number of dimensions is quadratic to the number of objects), and apply R-trees for nearest neighbor retrieval. In order to keep the number of dimensions stable, images containing various unlabelled objects are decomposed into combinations of images of fixed size. Although their method is efficient for domains involving relatively small images with few unlabelled objects (e.g., medical databases of X-rays) it is not applicable to large images of unlabelled objects.

In this paper we deal with the general problem where large images contain arbitrary objects and queries refer to object variables rather than instances. In order to provide a general solution, we propose a unified framework for structural similarity, which can represent various resolution levels and automates the derivation of similarity measures. We then apply several optimization techniques that can solve the problem for considerable data and query sizes. The techniques utilize ideas from related work in spatial databases (spatial join processing) and AI methods (constraint satisfaction algorithms). Although the problem of querying by structure is not a new one (it has been around since the early stages of computer vision [BB84]), to our knowledge this is the first approach to provide a solution which combines search algorithms with spatial indexing and can be applied for secondary memory retrieval. Our techniques have a wide range of potential applications in various areas (e.g., Geographic Information Systems, Multimedia Databases, VLSI).

The rest of the paper is organized as follows: Section 2 describes a binary string encoding for the representation of structure in multiple resolutions and dimensions. The new encoding permits the definition of similarity measures independently of the underlying domain and allows the retrieval of partial solutions. Sections 3 outlines the problem and provides examples of spatial queries and their processing. Sections 4, 5 and 6 describe three algorithms for structural query processing: the first one extends traditional spatial join methods to multiway (nested) joins. The second algorithm uses a search heuristic to prune the windows where query variables can be instantiated from, while the third one combines ideas from the first two algorithms. Section 8 concludes with future research directions.

2. A GENERAL FRAMEWORK FOR STRUCTURAL SIMILARITY

[F92] defined the concept of *conceptual neighborhood* as a cognitively plausible way to measure similarity among Allen's ([A83]) interval relations. A neighborhood is represented as a graph whose nodes denote temporal relations that are linked through an edge, if they can be directly transformed to each other by continuous interval deformations. In such a graph, similar relations are closer to each other than non-similar ones. Depending on the allowed deformation (e.g., movement, enlargement), several graphs may be obtained. The one in Figure 1, corresponds to what Freksa called *A-neighbors* (three fixed endpoints while the fourth is allowed to move). Starting from relation R_1 and extending the upper interval to the right, we derive relation R_2. With a similar extension we can produce the transition from R_2 to R_3 and so on. R_1 and R_3 are called 1st degree neighbors of R_2. The distance between two relations is equal to the length of the shortest path relating them in the neighborhood graph. Although hereafter we assume type A neighborhoods, extensions to other types are straightforward.

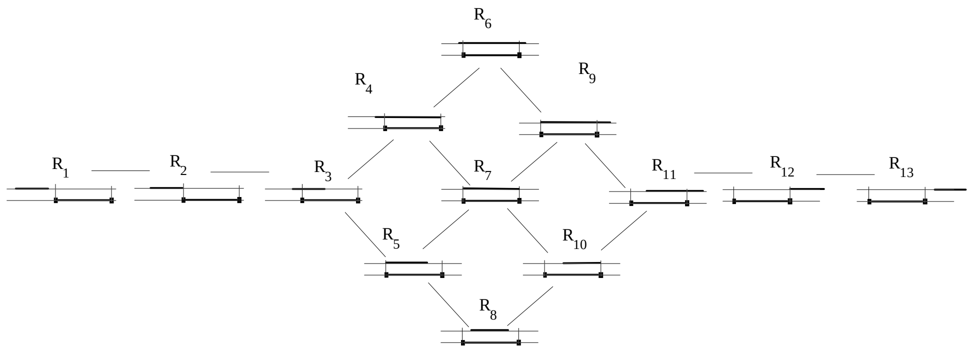


Figure 1 Conceptual Neighborhood for relations between intervals in 1D space

Related work on conceptual neighborhoods has been carried out for direction (e.g., *north*) [NNS96], topological relations (e.g., *inside, meet*) [EA92] and for classes of both topological and direction relations ([H94], [BE96]). Unlike previous methods that deal with application-dependent subclasses of spatial relations, we describe a framework that covers all three types of spatial relations (topological, directional, distance) under a unified multi-dimensional framework that supports multiple granularities. The proposed framework is easily adjustable to different application needs and may have a wide range of uses in query processing involving object and image retrieval.

We will start by describing a new encoding of relations. Consider a (reference) interval [a,b]. We identify 5 distinct 1D regions of interest with respect to [a,b]: 1.(-∞,a) 2.[a,a] 3.(a,b) 4.[b,b] 5.(b,+∞). The relationship between a (primary) interval [z,y] and [a,b] (the reference interval) can be uniquely determined by considering the 5 empty or non-empty intersections of [z,y] with each of the 5 aforementioned regions, modelled by the 5 binary variables *t, u, v, w, x,* respectively, with the obvious semantics ("0" corresponds to an empty intersection while "1" corresponds to a non-empty one). Therefore, we can define relations in 1D to be binary 5-tuples (R_{tuvwx} : $t, u, v, w, x \in \{0,1\}$). For example, R_{00011} (t=0, u=0, v=0, w=1, x=1) corresponds to the relation of Figure 2 (R_{12} in Figure 1).

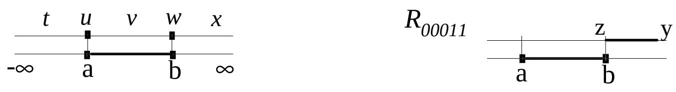


Figure 2 Encoding of spatial relations

A tuple of "1"s and "0"s represents a valid relation if it contains a list of consecutive "1"s (in the case of a single "1", this should not correspond to u or w, otherwise [z,y] collapses to a point) and the intervals of interest form a consecutive partition of (-∞,+∞). The above ideas can be extended in order to handle relations at varying resolution levels. We will initially illustrate the applicability of "binary string" encoding to a coarse resolution level, i.e. to a level where only a few relations can be distinguished. In the example of Figure 3, the 1D regions of interest are (-∞,a), [a,b] and (b,+∞), respectively. The corresponding relations are of the form R_{tuv}, $t,u,v \in \{0,1\}$. This allows for the definition of only 6 primitive relations since information content concerning the endpoints of [a,b] is reduced: R_{100} (*before*), R_{010} (*during*), R_{001} (*after*), R_{110} (*before_overlap*), R_{011} (*after_overlap*), R_{111} (*includes*). Figure 3 illustrates four configurations that correspond to R_{010} and cannot be distinguished in this resolution.

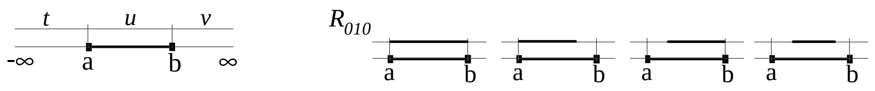


Figure 3 Encoding at a coarse resolution level

Increasing the resolution of relations in our model can be achieved simply by increasing the number of regions of interest (the number of bits in the binary string representation), thus refining the information level for a particular spatial relationship. In Figure 4 we use the middle point of [a,b] to define refinements of *overlap* relation such as *weak_overlap*, *strong_overlap*, etc.

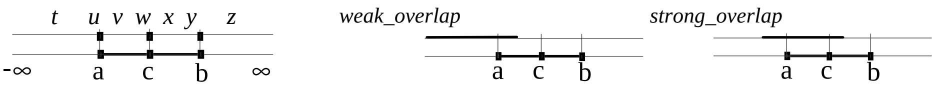


Figure 4 Refinement of *overlap* relation

Similarly, we can capture distance in our framework by refining *disjoint* relations, i.e., by splitting (-∞,a), (b,+∞) to several intervals of interest. Figure 5 illustrates a simple partitioning that allows the distinction between *far* and *near* relations (near defined as being in a distance up to δ and far otherwise). Nine bits are used for encoding such relations (*left_far,* for instance, can be defined as $R_{100000000}$). An arbitrary number of distance refinements can be used to define (for example a distance grid), according to the application needs. We call such consecutive partitionings of space *resolution schemes*; the ones we have seen so far are:

1. (-∞,a) [a,a] (a,b) [b,b] (b,+∞): the original resolution of Allen's relations.
2. (-∞,a) [a,b] (b,+∞): the coarse scheme where *meet* (at endpoints) relations can not be distinguished (Figure 3).
3. (-∞,a) [a,a] (a,c) [c,c] (c,b) [b,b] (b,+∞): (c being the middle point of [a,b]) which refines *overlap* relations (Figure 4).

4. (-∞,a-δ) [a-δ,a-δ] (a-δ,a) [a,a] (a,b) [b,b] (b,b+δ) [b+δ,b+δ] (b+δ,+∞): distance-enhanced scheme of Figure 5.

In general, if n is the number of bits used by the resolution scheme, the number of feasible relations in 1D is $n(n+1)/2 - k$, where k is the number of bits assigned to single points (i.e. intervals of the form [a,a]). If we fix the starting point at some bit then we can put the ending point at the same or some subsequent bit. There are n choices if we fix the first point to the leftmost bit, $n-1$ if we fix it to the second from the left, and so on. The total number is $n(n+1)/2$ from which we subtract the k single-point intervals. For the 1st scheme ($n=5$, $k=2$) we get 13 (Allen's) relations, while for ($n=9$, $k=4$) we get the 41 relations of Figure 5.

The new notation is more expressive, in the sense that given a relation, the user can easily infer the corresponding spatial configuration and vice versa (the only means to achieve this using the former notation is by referring to the neighborhood graph). More importantly, it permits the automatic calculation of relation distances and, consequently, of similarity measures. Figure 5 illustrates the neighborhood graph for the 4th resolution scheme. The edges are arranged horizontally and vertically and the semantics of traversing the graph in either direction are captured by the following "pump and prune" rule of thumb: Given a relation R_x, there are 4 potential neighboring relations, denoted *right*(R_x), *left*(R_x), *up*(R_x), *down*(R_x), respectively, with the obvious topological arrangement in the graph. *Right*(R_x) can be derived from R_x by "pumping" an "1" from the right, i.e., finding the first "0" after the rightmost "1" and replacing it by a "1". *Left*(R_x) can be derived from R_x, by "pruning" an '1' from the right, i.e. replacing the rightmost "1" by a "0". Similarly, *up*(R_x) can be derived from R_x by pumping an "1" from the left while *down*(R_x) can be derived by pruning the leftmost "1". Notice that not all neighboring relations are always defined. Consider, for example, the relation $R_{110000000}$, for which *up*($R_{110000000}$) is not defined since the leftmost digit is a "1" while *down*($R_{110000000}$)=$R_{010000000}$ is not a valid relation.

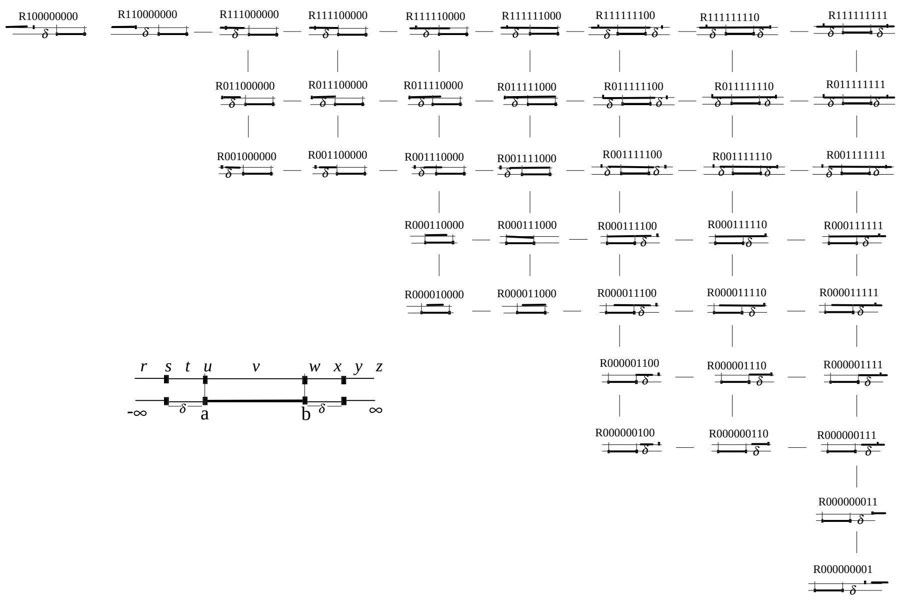


Figure 5 1D Conceptual neighborhood including distances

Since movement in the neighborhood graph is restricted to horizontal and vertical directions, the distance between two nodes is the sum of their vertical and horizontal distances. Equivalently, the distance between any two relations can be calculated by counting how many elementary movements we have to perform on an interval in order for the two relations to become identical. The larger the number of simple movements, the less similar the relations. The binary string representation enables the simplification of the previous procedure to the following one: we count the minimal number of

“0”s that we have to replace with "1"s in order to make the two notations two identical binary strings with consecutive “1”s. The corresponding pseudo code is given below:

```
INT distance(relation R1, relation R2) {
R = R1 OR R2;  //bitwise OR
count = 0;
FOR i:= R.leftmost_1() to R.rightmost_1 DO //for all bits between the leftmost and rightmost 1
       IF R1[i]=0 THEN count++;
       IF R2[i]=0 THEN count++;
RETURN (count);
}
```

For example $d(R_{\underline{000}11}, R_{1\underline{0000}}) = 7$ and $d(R_{\underline{0}11\underline{0}0}, R_{11110}) = 2$ (the underlined 0s are the ones counted during the calculation of distance). In order to extend the framework to multiple dimensions, we consider a N-dimensional relation as a N-tuple of 1D relations, e.g. $R_{110000000\text{-}111000000} = (R_{110000000}, R_{111000000})$ where each 1D relation corresponds to the relationship in one axis. So if s is the number of possible 1D relations at a particular resolution, the number of ND relations that can be defined at the same resolution is s^N. According to the requirements of the particular application, not all dimensions need to be tuned at the same resolution, in which case the maximum number of ND relations is the product of the corresponding numbers for each dimension.

Consider an N-dimensional relation $R_{i1\text{-}i2\text{-}\ldots\text{-}iN}$ where each i_k is a binary string. In order to derive a neighboring relation we have to replace one of its constituent 1D relations R_{ik} with its 1D neighbors, say R_{ik_j}, j=1..4, i.e.: $neighbor(R_{i1\text{-}i2\text{-}\ldots ik\ldots iN}) \in \{R_{i1\text{-}i2\text{-}\ldots ik_j\ldots\text{-}iN} \mid j=1..4, k=1..N\}$. As a result, computing ND neighbors is reduced to the already solved problem of computing 1D neighbors. ND neighborhood graphs are "fractal" images (graphs whose nodes are graphs, etc.). Figure 6 shows the 2D neighborhood for the distance-enhanced resolution scheme of Figure 5.

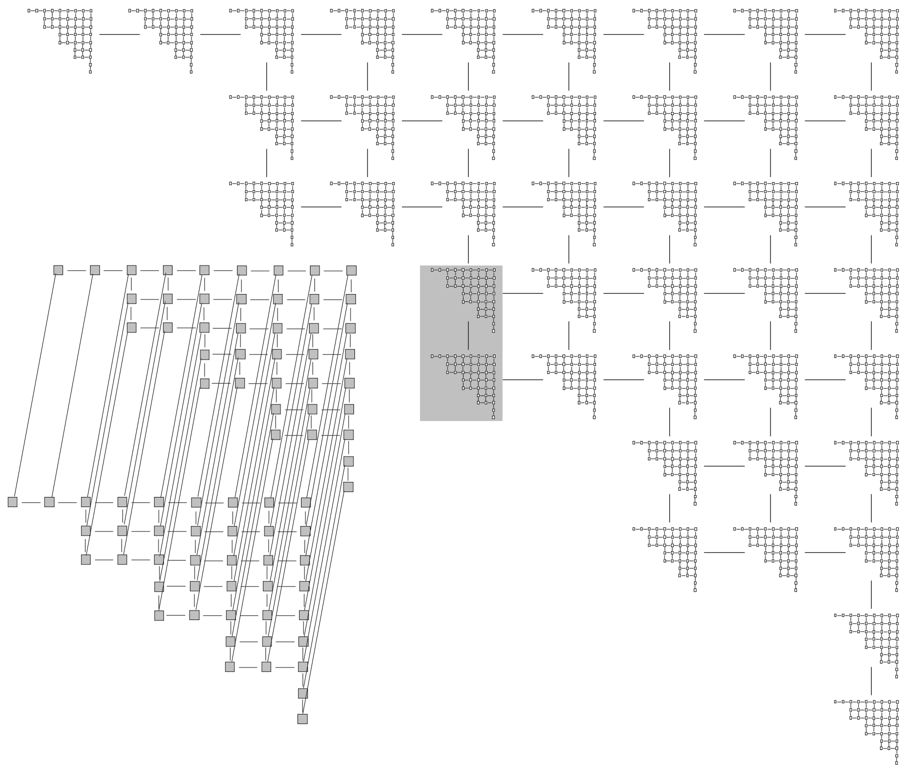


Figure 6 2D neighborhood graph for the distance-enhanced scheme

In this graph, 41 conceptual neighborhoods corresponding to one dimension are linked, forming a higher level conceptual neighborhood for the other dimension (each node in the big neighborhood graph is a small neighborhood graph). As the magnified section of the graph shows, in addition to the conceptual neighbors with the same x value, each relation R_{xj} has four more potential neighbors; i.e., the relations with the same j value and the x values related as shown in the large scale graph. According to the ND neighbourhood graph, the distance between two ND relations is the sum of the pair-wise distances between the corresponding constituent 1D relations, i.e. $d(R_{i1\text{-}i2\text{-}\ldots\text{-}iN}, R_{j1\text{-}j2\text{-}\ldots\text{-}jN}) = d(R_{i1}, R_{j1})+d(R_{i2}, R_{j2})+\ldots+ d(R_{iN}, R_{jN})$.

The advantages of the proposed framework are i) the expressiveness of the encoding in the sense that given the new notation, the corresponding spatial configuration can be easily inferred, and vice versa, ii) efficient automatic calculation of neighborhoods and relation distance, and iii) the uniform representation of all three types of relations (topological, directional, distance) in various resolution levels. For the sake of clarity, in the rest of the paper we use the distance enhanced resolution scheme of Figures 5 and 6. However for more realistic applications, sufficiently fine schemes (large encoding strings) can be used, while retaining the model's reasoning capabilities without adding any complexity.

3. STRUCTURAL QUERIES

The multidimensional extension of 1D relations define *projection-based* spatial relations [PS94], as each 1D relation between ND objects corresponds to the relation between their corresponding 1D projections. Projection-based definitions of relations and similarity measures are particularly suitable for structural similarity retrieval because usually spatial databases utilize minimum bounding rectangles (MBRs) as a fast *filter step* to exclude the objects that could not possibly satisfy a query [O86]. Furthermore, structural queries do not always have exact matches and crisp results. Rather, the output should have an associated "score"[1] to indicate its similarity to the query. By adoption, this score is inversely proportional to the degree of neighborhood.

A structural query can be formalised as a tuple $(T, \{(X_i, X_j, r, \tau)\})$, where T is the global tolerance and $\{(X_i, X_j, r, \tau)\}$ is a finite set of 4-tuples, where X_i, and X_j are query variables and r is a set (disjunction) of relations. Each instantiated pair (X_i, X_j) must satisfy r (i.e. one of the relations that belong to r) or a relation whose distance from r is equal or less than τ. The result of such a query consists of a set of n-tuples of data objects such that each object instantiates a query variable. In order for a *complete* instantiation (n-tuple) to be retrieved, its total distance (the sum of distances produced by all instantiation pairs) should less or equal than T.

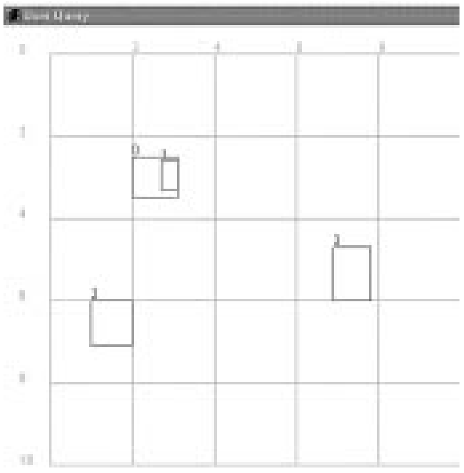
When T=0 only perfect matches will be retrieved, in which case we have *exact* retrieval. Retrieval with T>0 is called *partial*. τ ($\tau < T$) is used as a local threshold for partial retrieval in order to prune the search space (i.e., abandon search when the distance between an instantiated relation and the corresponding queried one exceeds τ). For example, assume that the global tolerance T for a query is 10 and τ=4. When the first pair of variables is instantiated all values that produce τ>4 will be eliminated even though they may eventually lead to complete instantiations whose total distance is less than 10.

Consider, for example, the query of Figure 7(a) which is a spatial arrangement of n=4 variables, expressed using a query-by-sketch language. Assuming the distance-enhanced resolution scheme, this query is represented as the set of relations of Figure 7(b)[2]. Although the particular language specifies relations between all pairs of variables, in some cases (e.g., verbal languages), queries may be

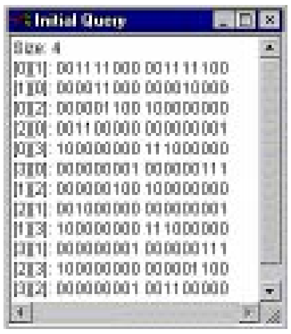
[1]Text information retrieval techniques, deal with the problem of similarity by associating the retrieved documents with a score proportional to the similarity of the query and the document.

[2] Our query language allows retrieval using area, shape (i.e., ratio of sides) and properties of objects. In general, structural queries could be combined with other forms of retrieval to express more complicated constraints (e.g., "find n-tuples with a structure similar to that of Figure 7(a) where X_0 is a park, X_1 a small lake" etc).

incomplete (some relations may be left unspecified), *indefinite* (a disjunction of relations) or even *inconsistent* (the relation between a pair of variables may contradict the relation of another pair). Figure 7(c) illustrates a solution where variable X_0 is instantiated to object 143, X_1 to object 207 and so on. The specific 4-tuple constitutes a partial solution whose distance from the query is 2 because some query relations (e.g., between X_0 and X_1) are not fully satisfied.

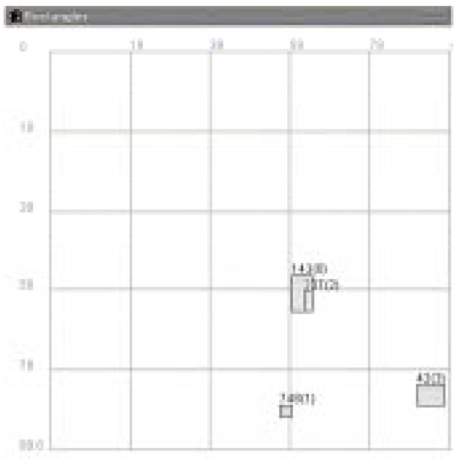


(a)



```
Initial Query
Size: 4
[0][1]: 001111000 001111100
[1][0]: 000011000 000010000
[0][2]: 000001100 100000000
[2][0]: 001100000 000000001
[0][3]: 100000000 111000000
[3][0]: 000000001 000000111
[1][2]: 000000100 100000000
[2][1]: 001000000 000000001
[1][3]: 100000000 111000000
[3][1]: 000000001 000000111
[2][3]: 100000000 000001100
[3][2]: 000000001 001100000
```

(b)



(c)

Figure 7 Example query

Depending on the viewpoint, the problem of processing structural queries can be modelled either as a *multiway spatial join* or as a *constraint satisfaction problem* (CSP). A multiway (or nested) spatial join, joins together *n* (>2) query variables and produces the same result as a series of two-way joins. Viewing structural query processing as a CSP, the query variables coincide with the CSP variables, whose domain consists of the image objects, and whose constraints are expressed in the query.

In both cases, indexing may take advantage of the special nature of spatial queries to facilitate efficiency. We use R-trees, one of the most widely-used spatial access methods, for structural query processing. The R-tree [G84] is a multidimensional extension of the height-balanced B-tree that stores the MBRs of the actual data objects in the leaf nodes; intermediate nodes are built by grouping rectangles at the lower level. Several variations of R-trees (e.g., [SRF87] [BKSS90]) have been proposed to enhance the performance of the original structure. Traditionally R-trees have been used for *window* queries, i.e., queries that return all objects overlapping a reference window. Other types of retrieval such as direction and topological queries can be transformed to window queries [PT97] and processed accordingly.

The retrieval of structural queries also requires some transformation to appropriate query windows. As an example consider the rectangles of Figure 8(a) which are organized in the R-tree of Figure 8(b) assuming a bucket size equal to three. If variable X_2 (example of Figure 7) is instantiated to object *a*, then X_3 must be instantiated to one of the objects inside the gray window of Figure 8(a) in order to satisfy the relation $R_{000000001\text{-}001100000}$ with respect to X_2. The lower side of potential instantiations for X_3 should coincide with the lower side of the window while the upper side must lie within the window. The tree nodes to be searched are the gray ones in Figure 8(b). In case of partial retrieval the window is extended on each axis depending on the tolerance τ. For instance, when τ=1, the query window is the one denoted by the dotted line. Candidate objects may touch one of the top or left sides of the new window but not the bottom one. In general, all queries involving projection relations can be transformed to query windows of various sizes. There exist a number of optimization techniques for pair-wise spatial joins based on R-trees (e.g., [BKS93,96][BKSS94][HJR97]) which could be extended for efficient processing of multiway joins. In addition, R-trees (and indexing in general) can be combined with CSP algorithms to effectively prune the search space and reduce the number of instantiations.

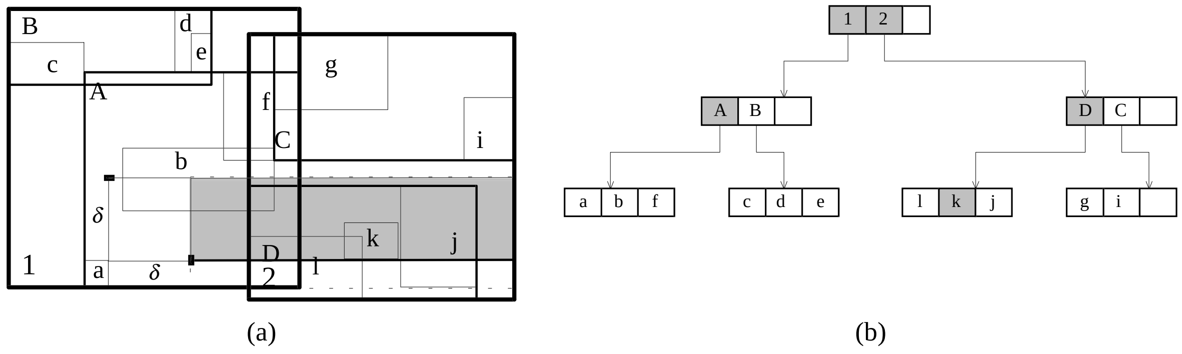


Figure 8 R-Tree and Query Window

Statistical information about the number of occurrences of each relation of interest in the data files may also improve performance significantly. Relations that occur rarely are of higher discriminative value, i.e., prune search space more effectively than frequent ones (information retrieval techniques take advantage of similar information by keeping the *inverse term frequency* [SAB94]). For instance, the relation $R_{001111000\text{-}001111100}$ between X_0 and X_1 in Figure 7(a) is more restrictive than the other relations (of the same query) because only a few pairs of objects satisfy it in normal data distributions. Therefore, during retrieval this relation should be processed first independently of the algorithm used (the *most-constrained-first* heuristic is utilized by many constraint satisfaction algorithms to determine the order of variables to be instantiated [D90]). Relation occurrences are calculated only once and stored with each image file as meta-data. When a file is searched for a particular structure, its associated meta-data is retrieved and used to determine the order of certain computations. In the sequel we describe several alternative approaches for structural query processing and quantify their performance through experimentation.

4. A MULTILEVEL FORWARD CHECKING ALGORITHM

An obvious way to perform multiway spatial joins is to execute pairwise joins and devise effective methods for the combination of the intermediate results. For instance, after performing the first join between X_i and X_j to use the potential values of X_j, in order to join it with X_k, etc. This approach faces the problem that the intermediate result may be too large to fit in main memory in order to be efficiently processed. In this paper we propose three alternative algorithms that avoid calculating intermediate results. The first one, multilevel forward checking (MFC), extends the techniques of [BKS93] to deal with n-tuples (instead of pairs) and cover a variety of relations defined by the resolution scheme used (instead of simple overlaps).

MFC finds the tuples of intermediate nodes (at each level of the R-tree) that may contain some solution objects and follows the references to the next level, until it reaches the leaves, where it outputs solutions. For instance, the path to solution (*d,e,a,k*) of the example query in the tree of Figure 8(b) is: (1,1,1,2) at the top, (B,B,A,D) at level 1 and (d,e,a,k) at level 0. In order to avoid searching all combination of nodes at each level (e.g., (1,1,1,1), (1,1,1,2), …., (2,2,2,2) for the top) the algorithm utilizes the query constraints. However, unlike traditional joins where *overlap* is used as the join condition throughout the tree, in our case the relations between intermediate nodes that may contain solution objects are different from the ones between object MBRs and not apparent. For instance, there is no obvious relation between the intermediate nodes that contain (d,e,a,k).

In order to join intermediate nodes we generate for each query *q*, a second one *Q*, which expresses the set of corresponding constraints between intermediate nodes given the constraints of *q*. *Q* contains the same number of variables as *q*, but related by more relaxed constraints (which are also encoded in bitstrings). For example, if *q* specifies that two variables are related by *inside* (e.g., $R_{000010000\text{-}000010000}$), then the intermediate nodes that contain qualifying objects may overlap (they are not necessarily related by inside). Although the corresponding constraints in *Q* are less restrictive than the ones in *q*, they do eliminate some values (e.g., the intermediate nodes cannot be disjoint).

At each level the calculation of combinations of the qualifying nodes may be expensive, as the number of total combinations can be as high as C^n, where *C* is the capacity of an R-tree bucket. Although the search space is not prohibitively large (usually $n \leq 10$ and $C \leq 200$), the computational burden is due to numerous appearances of the problem during query processing. Finding the subset of node combinations which is "legal" with respect to *Q* can be treated as a local CSP problem at each level. MFC uses a f*orward checking*[3] heuristic to solve this problem: every time a variable is instantiated to an entry in the corresponding intermediate node, the heuristic eliminates all potential values of subsequent (so-far uninstantiated) variables that are inconsistent with the current instantiation (i.e., values that violate some constraint of *Q*). Backtracking occurs when an instantiation eliminates all possible values for an uninstantiated variable.

The parameters of MFC are *q*, *Q*, and a n-tuple of references (*nodes*) to the domain of the variables for the current tree level. Variable *nodes* does not correspond to the actual tree nodes, but to a 2-dimensional array of references, where *nodes[i][j]* denotes the j-th entry of the domain node for the i-th variable. Although two variables X_k, X_m may be instantiated to values from the same bucket, their corresponding indices *nodes[k]* and *nodes[m]* have different entries which point to the same actual R-tree references. The pseudo code for MFC is:

```
MFC(Query q, Query Q, Node[] nodes) {
        index = 0; /* the current variable */
        WHILE (TRUE) {
                new_value = getNextValue(index);
                IF new_value = NULL THEN /* end of domain */
                        IF index=0 THEN RETURN;
                        ELSE index=index-1; CONTINUE;  /*Backtrack*/

                current_instantiations[index]=new_value; /*index is instantiated to new_value*/
                IF index = number_of_variables-1 THEN  /*last variable instantiated*/
                        IF level>0 THEN /*intermediate level*/
                                MFC(q, Q, current_instantiations);
                        ELSE /*leaf level*/
                                output_solution(current_instantiations);
                ELSE
                        IF level>0 THEN //intermediate level
                                variable_eliminated = check_forward(index, nodes, Q);
                        ELSE //leaf level
                                variable_eliminated = check_forward(index, nodes, q);
                        IF variable_eliminated THEN // unsuccessful instantiation
                                restore_eliminations(index);
                                CONTINUE;
                        index=index+1; /* successful instantiation: go forward */
        }
}

BOOLEAN check_forward(int index, Node[] nodes, Query q) {
        FOR i = index+1 TO num_variables-1 DO /*for all uninstantiated variables*/
                FOR j = 0 TO nodes[i].num_entries-1 /* for all potential values */
                        IF inconsistent (current_instantiations[index], nodes[i][j]) THEN
                                eliminate (nodes[i][j]); /*var. i cannot take value nodes[j] */
                IF (all_values_eliminated(i)) THEN RETURN FALSE;
```

[3] Forward checking [BG95] is considered as one of the most powerful ways for solving CSPs [N89][FD94].

```
        RETURN TRUE;
}
```

Initially *nodes* is set to an n-tuple that points to the tree root for all variables. Each time a query variable X_{index} takes a value, *check_forward()* iterates through all potential values of subsequent (uninstantiated) variables X_i and tests their validity against the current instantiation of X_{index}. Illegal values (the ones violating the constraint between X_{index} and X_i) are removed. If the domain of an uninstantiated variable is totally eliminated, the present instantiation of X_{index} becomes invalid, the values of future variables that have been eliminated due to X_{index} are restored, and a new value is chosen. If a new value for X_{index} cannot be found, the algorithm backtracks to the previous variable $X_{index-1}$. A solution for the current tree level is found when the last variable is instantiated. The algorithm is then recursively invoked for the next level, or the solution is reported if it refers to actual object MBRs. MFC finishes when it backtracks from the first variable, because all the possible instantiations have been tested for the current tree level.

Before forward checking, MFC applies a variation of the *space restriction heuristic* [BKS93] to avoid unnecessary instantiations. Consider, for example, the R-tree of Figure 8 and the query of Figure 7. Assume the qualifying 4-tuple (1,2,2,2) for the top level of the tree. In the middle level, after the references are followed, the candidate values for X_0 are {A,B}. However, due to the relative positions of B and intermediate node 2 (disjoint), there can't be any instantiations of X_1 below node 2 that lead to solutions (valid instantiations for X_0 and X_1 should be inside intersecting nodes). Therefore, we can safely prune value B from X_0's domain and avoid useless instantiations. The following *Space_Restriction* routine takes the entries (e.g., A, B) of a node (e.g., 1) one by one and tests them against the MBRs of the rest of the nodes (e.g., 2), eliminating the ones that do not satisfy the constraints in *Q*:

```
Space_Restriction(Query Q, Node[] nodes){
FOR i=0 TO number_of_variables-1 DO
        FOR j=0 TO nodes[i].num_entries-1 DO
                FOR k=0 TO number_of_variables-1 DO
                        IF (i≠k) AND (inconsistent(n[i][j], n[k].MBR)) THEN
                                eliminate (nodes[i][j]);
}
```

5. A WINDOW-REDUCTION ALGORITHM

When the constraints between intermediate nodes are not tight, MFC suffers from the large number of pages that must be visited at the higher levels of the tree (most of which are false hits). An alternative approach that overcomes this problem is to directly use the data MBRs for the instantiation of the query variables and employ regular forward checking. The problem with this method is that it cannot be applied for large data files because the domain of a variable (the data MBRs) may be too large to fit in main memory[4]. In this section we propose another forward checking-based algorithm, window reduction (WR), which avoids this problem.

Normally, forward checking tests the legal values of future variables against the current instantiation and eliminates the inconsistent ones. On the other hand, WR takes advantage of spatial constraints to maintain a *domain window* (instead of the actual domain) that encloses all potential values for each variable (and possibly some false hits). When X_i is instantiated, the domain window of every uninstantiated variable X_j is updated (reduced) according to the current value of X_i and the constraint between X_i and X_j.

[4] Forward checking in MFC is applied locally within each bucket; therefore the size of each domain is limited to the bucket size.

```
WR(Query q) {
        index = 0; /* the current variable */
        FOR i=0 TO num_variables-1 DO
                domainWindow[0][i] = U; /* initially all domains are universal */
        WHILE (TRUE) {
                new_value = getNextValue(index, domainWindow[index][index]);
                IF new_value = NULL THEN
                        IF index = 0 THEN BREAK;
                        ELSE index=index-1; CONTINUE; /*Backtrack*/

                IF inconsistent(index) THEN CONTINUE; /* eliminate false hit */
                current_instantiations[index]=new_value; /*index is instantiated to new_value*/
                IF index = number_of_nodes-1
                        THEN output_solution(current_instantiations);
                ELSE
                        IF window_reduction(index, domainWindow, q) = FALSE THEN CONTINUE;
                        reorder_variables(index+1,domainWindow);/*get var with smallest window*/
                        index=index+1;
        }
}

BOOLEAN window_reduction(int index, Rectangle domainWindow[][], Query q) {
        FOR i=index+1 TO num_variables-1 DO
                domainWindow[index+1][i] = domainWindow[index][i] ∩ getWindow(index,i,q);
                IF domainWindow[index+1][i] = ∅ THEN RETURN FALSE; /* empty domain window*/
        RETURN TRUE;
}
```

DomainWindow is a 2-dimensional array, whose rows correspond to instantiation levels and columns to query variables. *DomainWindow*[i][j] stores the coordinates of the window where X_j should fall in after instantiating X_i (i<j). When X_i takes a new value, a new window W_j for X_j is computed taking into account the value of X_j and the constraints between X_i and X_j. The intersection of W_j with (existing) *domainWindow[i][j]* is stored at *domainWindow[i+1][j]*, in order to be used at the next instantiation level (previous windows are kept for backtracking). If this intersection is empty (∅), the current instantiation is invalid and the algorithm proceeds to the next value for X_i. WR can be thought of as a "lazy" version of forward checking because the domain windows are calculated but no values are retrieved until the variable gets instantiated.

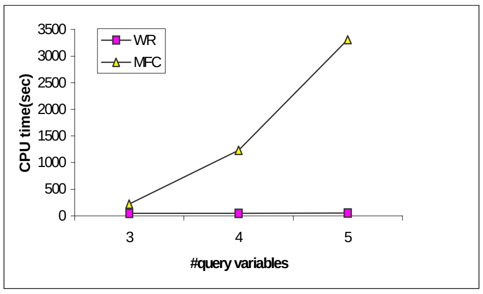
The next value for a variable X_i is retrieved via *getNextValue()*, which uses *domainWindow[i][i]* as the query window for X_i. *GetNextValue()* does not perform a window query every time it is invoked, but the whole search path for each variable is maintained in memory. The overhead for this path-holding technique is the maintenance of the search path of all query variables, i.e. $n \cdot h$ nodes -a considerable number of pages. *Inconsistent()* checks whether the value is consistent with previous instantiations since not all values that fall inside the domain window of X_i are necessarily legal.

In addition to domain windows and path maintenance techniques, WR uses *dynamic variable reordering* [BvR95]. As opposed to MFC, where the order is pre-defined using information about relation frequency (stored as image meta-data), *reorder-variables()* dynamically determines the instantiation order. When the domain windows of the future variables are calculated after an instantiation, the variable with the smallest domain window becomes the next to be examined. This is lead by the intuition that a small window is more likely to contain the least number of instantiations and minimize redundant consistency checks (this is the reason why the order of variables X_1 and X_2 has changed in Figure 8(b)).

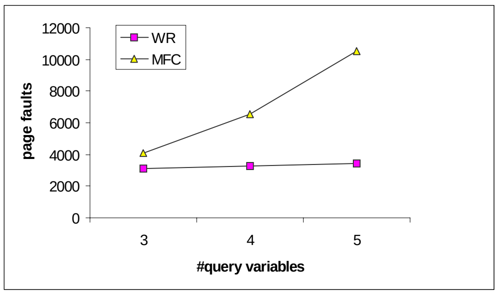
In order to compare the performance of WR with MFC, we implemented and tested the algorithms for several query types. For our experiments we used LB datafile [T94] which contains 53,145 rectangles representing road segments of Long Beach county. The maximum distance of the rectangles in each axis is 10000, and the data density 0.25. From the above file we built several R*-

trees of different block sizes, i.e. 512 bytes, 1K, 2K, and 4K. The LRU buffer size of the R*-trees during the experiments was set to 128. We constructed 5 artificial sets of 30 queries: the number of variables in the queries of each set was fixed to 3, 4, …, 7. In order to avoid trivial queries, each variable was set to intersect with some other variable on at least one axis. The distance between two variables on each axis did not exceed δ, which was set to 100. The implementation was done using Symantec Café JIT compiler v210.065 and the experiments were run on 5 Pentium PCs 200MHz with 64Mb RAM.

Figure 9 shows the mean CPU-time (a) and page faults (b) after the application of MFC and WR for the query-sets of 3,4 and 5 variables on the R*-tree with 1KB block size. WR clearly outperforms MFC by orders of magnitude (although in this scale CPU time and page faults appear to be constant for WR, they actually increase with the number of variables as shown in Figure 10). The performance gap widens with the query size because the domain windows of WR are continuously decreasing as new variables are instantiated. Therefore the domains of the last variables are relatively small and do not constitute a significant overhead to the algorithm. On the other hand, the relaxed constraints between intermediate nodes do not permit MFC to prune the search space at the higher levels of the tree; thus, MFC cannot avoid the combinatorial explosion of possible instantiations as the number of variables increases. Although MFC is not an appropriate algorithm for the current resolution scheme, it could perform better in applications involving strict relations between intermediate nodes.



(a)



(b)

Figure 9 Comparison of WR and MFC for 1KB page size

6. A JOIN WINDOW-REDUCTION ALGORITHM

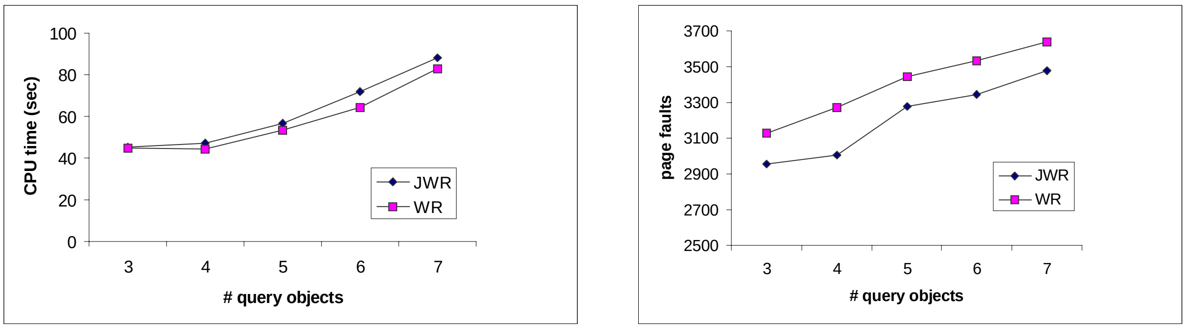
WR essentially searches the whole space in order to instantiate the first variable, but after doing so it performs only window queries which are cheap operations in R-trees. The disadvantage of blindly instantiating the first variable in the whole universe could be avoided by an algorithm that combines properties of both MFC and WR. The third algorithm (JWR) first applies a pairwise spatial join to pin down values for a pair of variables (the ones related by the most infrequent relation) and then uses window reduction to instantiate the rest of the variables. The subsequent variables are instantiated in the same way as in WR:

```
JWR(Query q, Query Q) {
        index = 1;
        FOR i=1 TO num_variables-1 DO
                domainWindow[0][i] = U; /* initially all domains are universal */
        WHILE (TRUE) {
                IF (index = 1) /* values of first two variables */
                        IF getNextPair(q, Q) = NULL THEN BREAK;

                ELSE  /* values of third and subsequent variables */
                        new_value = getNextValue(index, domainWindow[index][index]);
                        IF new_value = NULL THEN
                                index=index-1; CONTINUE; /* Backtrack*/
                        IF inconsistent(index) THEN CONTINUE; /* Eliminate false hit */
                current_instantiations[index]=new_value;
                IF index = number_of_nodes-1 THEN
                        output_solution(current_instantiations);
                ELSE
                        IF window_reduction(index, domainWindow, q) = FALSE THEN CONTINUE;
                        reorder_variables(index+1, domainWindow);
                        index=index+1;
        }
}
```

Function *getNextPair()* returns the next pair that satisfies the relations between the first two variables, utilizing the CPU-time tuning techniques proposed in [BKS93], namely search *space restriction* and *plane sweep*. The search space heuristic in *getNextPair* uses *Q* (like MFC) to filter the candidate sets of nodes to be joined. Plane sweep is a well known technique [PS85] that efficiently computes intersections of rectangles on the plane. Because we are interested in various relations, and not only intersections, we apply a variation of plane sweep which can deal with the whole set of relations of the current resolution scheme.

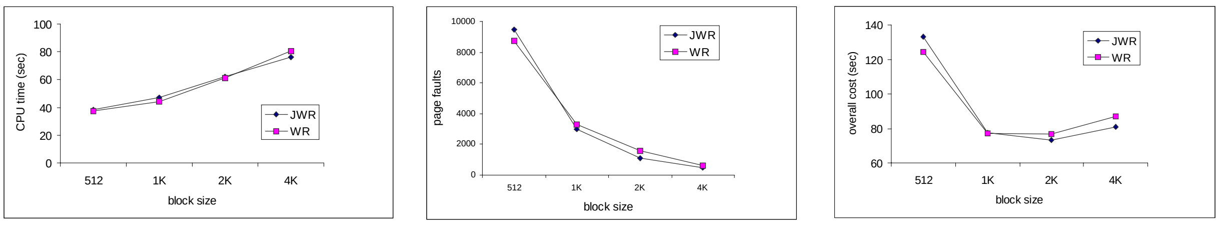
We compared the performance of JWR and WR under several conditions. Figure 10 shows the mean CPU-running time and page faults of the algorithms for the R*-tree of block size 1K, and a variable number of query sizes. JMW maintains a small performance gain on the number of page faults, whereas the CPU-time of both algorithms is almost identical. The performance gap is not affected by query size, because the only difference of the algorithms is the instantiation method for the first pair of variables.



(a) (b)

Figure 10 Comparison of JWR and WR for 1KB page size

In order to evaluate the algorithms for various block sizes we executed the 4-variable query set using R*-trees of 512, 1K, 2K, and 4K bucket sizes. The results are shown in Figure 11. The overall cost was estimated by charging 10ms for each page access (a typical value [HJR97]). The algorithms perform better for page sizes 1K and 2K where JWR has a slight advantage over WR. For small page sizes the overhead of frequent page accesses is very high.



(a) (b) (c)

Figure 11 Comparison of JWR and WR for various page sizes

Finally, we tested the performance of JWR over queries with non-zero tolerance. In all experiments the global tolerance *T* was set to 10. Figure 12 illustrates the overall cost of JWR for the 2K page size R*-tree. Each line corresponds to a different value of local tolerance τ. Because partial retrieval is equivalent to exact retrieval using a larger window, the domain windows of JWR get larger as τ increases. Larger windows imply more potential legal values and more consistency checks.

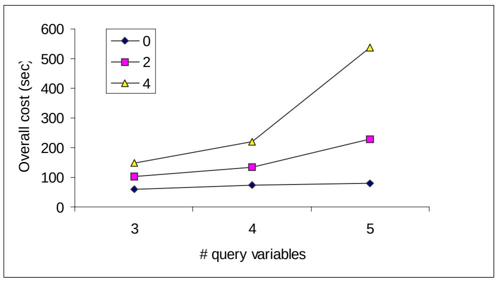


Figure 12 Overall cost of JWR for partial retrieval

7. CONCLUSION

This paper addresses the issue of spatial structural queries, i.e., queries that ask for all n-tuples of objects that satisfy some spatial constraints. We first proposed a formal yet practical framework for encoding 1D relations in a way that allows efficient generation of similarity measures. We subsequently extended the model in a uniform way to arbitrary dimensions and multiple resolution levels, thus covering many potential applications. Then we presented three algorithms for structural query processing:

– MFC which applies hierarchical constraint satisfaction to eliminate tuples of intermediate nodes that cannot lead to solutions.
– WR which gradually reduces the domain windows of uninstantiated variables based on the values of instantiated ones.
– JWR which performs a pairwise join to instantiate the first pair of variables and then applies the same window reduction technique as WR.

Finally we experimentally evaluated their performance and found that the last two clearly outperform the first one. All algorithms are independent of the resolution scheme used so they can be used to process any type of spatial predicates (including multiway overlap joins).

Future research could be carried out on alternative algorithms and optimization methods. A promising method involves parallel processing of nested spatial joins. Some algorithms, which do not perform well for uni-processor systems may have significant improvements on certain parallel architectures. For instance, we could independently process pairwise joins and devise effective methods for the combination of results (a method that is problematic for uni-processor systems due to the large size of intermediate results). Another important topic relates to the development of appropriate semantic models and query languages for structural similarity retrieval. Although there already exist some pictorial languages [SC96] that can express structural similarity, they have rather limited capabilities (e.g., they cannot express disjunction "find all configuration where X_1 is north or meets X_2 …"). On the other hand, verbal query languages, such as SQL, may be complex and counter intuitive. For instance, the expression of structural query involving *n* variables could involve as many as *n(n-1)/2* predicates if all binary constraints must be specified.

REFERENCES

[A83] Allen, J., "Maintaining Knowledge About Temporal Intervals", CACM, 26(11), 1983.

[BG95] Bacchus, F., Grove, A. "On the Forward Checking Algorithm", International Conference on Principles and Practice of Constraint Programming, 1995.

[BvR95] Bacchus, F., van Run, P. "Dynamic Variable Ordering in CSPs", International Conference on Principles and Practice of Constraint Programming, 1995.

[BB84] Ballard, D., Brown, C. "Computer Vision". Prentice Hall, 1984.

[BKSS90] Beckmann, N., Kriegel, H.P. Schneider, R., Seeger, B. "The R*-tree: an Efficient and Robust Access Method for Points and Rectangles". ACM SIGMOD, 1990.

[BKS93] Brinkhoff, T., H.-P. Kriegel, B. Seeger "Efficient processing of spatial joins using R-trees". ACM SIGMOD, 1993.

[BKSS94] Brinkhoff, T., H.-P. Kriegel, R. Schneider, and B. Seeger "Multi-step Processing of Spatial Joins". ACM SIGMOD, 1994.

[BKS97] Brinkhoff T., Kriegel H.-P., Seeger B. "Parallel Processing of Spatial Joins Using R-trees". Proceedings of 12th International Conference on Data Engineering, 1996.

[BE96] Bruns, T.H., Egenhofer, M.J., "Similarity of Spatial Scenes", 7th Symposium on Spatial Data Handling, 1996.

[CSY87] Chang, S.K., Shi, Q.Y., Yan C.W. "Iconic Indexing by 2-D String". IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-9, no 3, pp. 413-428, 1987.

[D90] Dechter, R. "Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition". Artificial Intelligence 41, 273-312, 1990.

[EA92] Egenhofer, M.J., Al-Taha, K. "Reasoning about Gradual Changes of Topological Relations", *International Conference GIS- From Space to Territory.* Springer Verlag LNCS, 1992.

[ELS95] Evangelidis, G., D. Lomet, and B. Salzberg (1995). "The hB$^{\Pi}$-tree: A Modified hB- tree Supporting Concurrency, Recovery and Node Consolidation". VLDB, 1995.

[F92] Freksa, C., "Temporal Reasoning based on Semi Intervals", Artificial Intelligence, Vol 54, pp. 199-227, 1992.

[FD94] Frost, D., Dechter, R. "In Search of the best Constraint Satisfaction Search". AAAI, 1994

[GR95] Gudivada, V., Raghavan, V. "Design and evaluation of algorithms for image retrieval by spatial similarity". ACM Transactions on Information Systems, 13(1):115-144, 1995.

[G93] Guenther, O. "Efficient computation of spatial joins". IEEE International Conference on Data Engineering, 1993.

[G84] Guttman, A. "R-trees: A Dynamic Index Structure for Spatial Searching". ACM SIGMOD, 1984.

[H94] Hernandez, D., "Qualitative Representation of Spatial Knowledge". Springer Verlag LNAI, 1994.

[HJR97] Huang, Y-W, Jing, N, Rundensteiner, E. "Spatial Joins using R-trees: Breadth First Travesral with Global Optimizations". VLDB, 1997.

[LH92] Lee S-Y., Hsu F-J "Spatial Reasoning and Similarity Retrieval of Images using 2D C-string Knowledge Representation". Pattern Recognition, 25(3):305-318, 1992.

[NNS96] Nabil, M., Ngu, A., Shepherd, J., "Picture Similarity Retrieval using 2d Projection Interval Representation", IEEE TKDE, 8(4), 1996.

[N89] Nadel. B. "Constraint Satisfaction Algorithms". Computational Intelligence, 5, pp. 188-224, 1989.

[O86] Orenstein, J. A. "Spatial Query Processing in an Object-Oriented Database System. ACM SIGMOD, 1986.

[PS94] Papadias, D., Sellis, T., "Qualitative Representation of Spatial Knowledge in Two-Dimensional Space", VLDB Journal, Vol. 3(4), pp. 479-516, 1994.

[PTSE95] Papadias, D., Theodoridis, Y., Sellis, T., Egenhofer, M. "Topological Relations in the World of Minimum Bounding Rectangles: A study with R-trees". ACM SIGMOD, 1995.

[PT97] Papadias, D., Theodoridis, Y., "Spatial Relations, Minimum Bounding Rectangles, and Spatial Data Structures". International Journal of Geographic Information Systems, 11(2), pp. 111-138, 1997.

[PF97] Petrakis, E., Faloutsos, C. "Similarity Searching in Medical Image Databases". IEEE TKDE, 9 (3) 435-447, 1997.

[PS88] Preparata F, Shamos, M. "Computational Geometry". Springer, 1988.

[R91] Rotem, D. "Spatial Join Indices". IEEE International Conference on Data Engineering, 1991.

[RKV95] Roussopoulos, N., Kelley, F., Vincent, F., “Nearest Neighbor Queries”, ACM SIGMOD, 1995.

[S90] Samet, H., "The Design and Analysis of Spatial Data Structures". Addison Wesley, 1990.

[SAB94] Salton, G., Allan, J., Buckley, C., "Automatic Structuring and Retrieval of Large Text Files". *CACM*, 37(2): 97-108, 1994.

[SRF87] Sellis, T., N. Roussopoulos, and C. Faloutsos "The R+ -Tree: A Dynamic Index for Multi-Dimensional Objects". VLDB, 1987.

[SC96] Smith, J. R., Chang, S-F., “Searching for Images and Videos on the World-Wide Web”, Technical Report CU/CTR 459-96-25, Columbia University, 1996.

[T94] TIGER/Line Files, 1994 Technical Documentation / prepared by the Bureau of the Census, Washington, DC, 1994.