Hierarchical Constraint Satisfaction in Spatial Databases

Dimitris Papadias, Panos Kalnis, Nikos Mamoulis

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/{~dimitris, ~kalnis, ~mamoulis}

Abstract

Several content-based queries in spatial databases and geographic information systems (GISs) can be modelled and processed as constraint satisfaction problems (CSPs). Regular CSP algorithms, however, work for main memory retrieval without utilizing indices to prune the search space. This paper shows how systematic and local search techniques can take advantage of the hierarchical decomposition of space, preserved by spatial data structures, to efficiently guide search. We study the conditions under which hierarchical constraint satisfaction outperforms traditional methods with extensive experimentation.

Introduction

Consider that a user is searching for a triplet (v_1,v_2,v_3) of a residential area, a commercial center and a park, such that v_1 *covers* v_2 and v_2 *meets* v_3. The query can be modeled as a CSP where: i) each query object corresponds to a CSP variable ii) a pair of variables is related by the respective query constraints (e.g., covers(v_1,v_2)) iii) the domain of each variable consists of the corresponding objects in the database (e.g., the domain of v_1 is the set of all stored residential areas). As opposed to other CSP applications, here the number of variables is relatively small (usually less than ten), while the domains are very large (geographic maps may contain more than 100,000 objects).

Because of the large amount of data involved, spatial databases and GIS employ indexing for efficient retrieval. The R-tree (Guttman 1984) and its variations is the most popular multi-dimensional access method, currently used in many commercial GISs and DBMS (e.g., Informix, Illustra, MapInfo). R-trees have been applied for a variety of queries including spatial selections, nearest neighbors, and spatial joins. This paper illustrates how the hierarchical decomposition of space, preserved by R-trees, can be utilized by CSP algorithms to accelerate search.

In order to provide a general framework of retrieval we use the 9-intersection model (Egenhofer 1991) as the basis for defining spatial constraints. This model, which is becoming a standard in commercial systems (e.g., Intergraph and Oracle spatial products), describes eight mutually exclusive topological relations between planar regions (Figure 1) using intersections of object interiors and boundaries. The same set of relations, called RCC-8 (region connection calculus) in AI literature, was defined independently in (Randell, Cui, and Cohn 1992). Alternatively, depending on the application needs, the proposed techniques could be used with other types of spatial constraints such as directions (e.g., *north*) and distances.

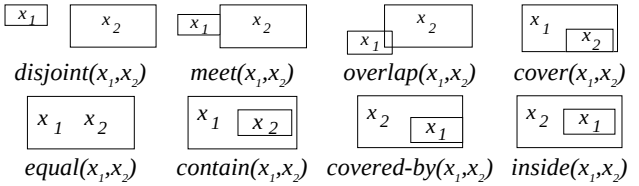


Figure 1 Topological relations

The rest of the paper is organized as follows: Section 2 describes R-trees and spatial query processing techniques. Section 3 defines hierarchical constraint satisfaction based on indexing and outlines pre-processing techniques. Section 4 experimentally evaluates the performance gain of hierarchical constraint satisfaction for systematic search and local search techniques. Finally, Section 5 concludes the paper.

Background

The R-tree data structure is a height-balanced tree that consists of intermediate and leaf nodes corresponding to disk pages in secondary memory (R-trees are extensions of B^+-trees to many dimensions). The root is at level *h*-1, where *h* is the height of the tree, and the leaf nodes at level 0. The minimum bounding rectangles (MBR) of the actual data objects are stored in the leaf nodes, and intermediate nodes are built by grouping rectangles at the lower level. Notice, that each object has a distinct identity and location in space. Furthermore, in most applications there is a separate R-tree for every type of object (e.g., residential areas, parks).

Figure 2 illustrates R-trees that index three sets of objects covering the same area. For this example we assume that the maximum node capacity *C* is 3 rectangles (in real 2D applications *C* is normally 50-400 depending on the page size). MBRs a_1, a_2 and a_3 of the first R-tree are grouped together in an intermediate node A_1, which is contained in the root. In the rest of the paper, we make the distinction between an R-tree node X_i and its *entries* $X_{i,1}$,.., X_{i,C_i}, (where $C_i \leq C$ is the capacity of X_i) which correspond to MBRs included in X_i. $X_{i,k}$.*ref* points to the corresponding node X_k at the next (lower) level. For instance, at level 1 of the first tree, the entries of the root are A_1 and A_2, which point to nodes at level 0. An entry of a leaf node X_i is an object MBR $x_{i,k}$.

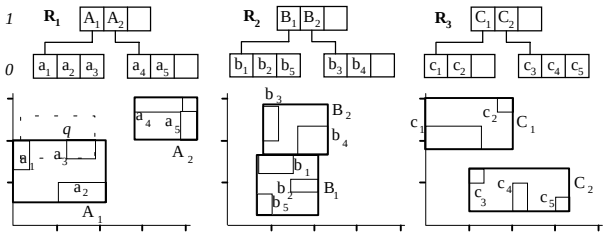


Figure 2 A set of objects and the corresponding R-tree

Traditionally, R-trees have been used for *window queries* which ask for a set of objects that *intersect*[1] a window *q*. The processing of a window query in R-trees involves the following procedures: Starting from the top node, exclude the entries that are *disjoint* with *q*, and recursively search the remaining ones. Among the entries of the leaf nodes retrieved, select the ones that are *non-disjoint* with *q*. For instance when searching for objects sharing common points with the dotted window in the first tree, we need not access A_2 since it cannot contain qualifying objects.

When two MBRs are *disjoint*, the objects that they approximate are also *disjoint*. If the MBRs however share common points, no conclusion can be drawn about the relation between the actual objects. For this reason, spatial queries involve the following two step strategy: (i) a *filter step*, in which the tree is used to retrieve a set of candidates that includes all the results and possibly some *false hits*, and (ii) a *refinement step* where each candidate is examined and false hits are eliminated. Here, as in most related literature, we will only consider MBRs, avoiding the refinement step (which is based on computational geometry techniques and is outside the scope of this paper).

The above method can be extended for the retrieval of the topological relations of Figure 1. In contrast to window queries where the retrieval condition is *non-disjoint* for all levels of the tree, in order to retrieve topological relations using R-trees one needs to define conditions for the intermediate nodes. For instance, A_1 encloses a_3 which is *covered-by q*, but the relation between A_1 and *q* is *overlap*. Table 1 presents, for each relation, the condition between an intermediate node *X* and *q*, so that *X* may contain qualifying objects *x*.

Relation *(x,q)*	**Condition for intermediate nodes (*X,q*)**
equal	equal ∨ cover ∨ contain
contain	contain
inside	overlap∨covered-by∨inside∨equal∨cover∨contain
cover	cover ∨ contain
covered-by	overlap∨ covered-by ∨ equal ∨ cover ∨ contain
disjoint	disjoint∨ meet∨ overlap∨ cover∨ contain
meet	meet∨ overlap ∨ cover ∨contain
overlap	overlap ∨ cover ∨ contain

Table 1 Conditions for intermediate nodes (window queries)

R-trees can also effectively support *intersection joins*, i.e., queries that select from two object sets, the pairs that satisfy some spatial predicate, usually *intersect* (e.g., "find all land parcels intersecting some forest area"). The most influential algorithm for processing intersection joins using R-trees, is *R-tree join* (*RTJ*) proposed in (Brinkhoff, Kriegel, and Seeger 1993). It is based on the *enclosure property* of R-trees: if two intermediate nodes X_i and Y_j (possibly belonging to different trees) are *disjoint*, then all pairs $(X_{i,k},Y_{i,l})$ of their entries are also *disjoint*. *RTJ* starts from the roots of the two trees to be joined (e.g., R_1 and R_2) and finds all pairs of *non-disjoint* entries inside them (e.g., (A_1,B_1) and (A_1,B_2)). These are the only pairs that may lead to solutions; for instance, there can be no pair (a_i,b_j) $a_i \in A_2$ and $b_j \in B_1$ such that (a_i,b_j) is a solution, since A_2 is *disjoint* with B_1. For each *non-disjoint* pair of entries, the algorithm is recursively called until the leaf levels where intersecting pairs constitute solutions.

Like window queries, in order to process arbitrary topological relations using *RTJ*, we need to define the conditions between intermediate nodes X_i and Y_j that could enclose (at any level below) objects x_k and y_l satisfying the join predicate. Table 2 contains the allowed relations between X_i and Y_j, so that they could contain qualifying pairs (x_k,y_l).

Relation (x_k,y_l)	**Condition for intermediate nodes (X_i,Y_j)**
equal, contain, inside, overlap, cover, covered-by	overlap ∨ covered-by ∨ inside ∨ equal ∨ cover ∨ contain
disjoint	disjoint ∨ meet ∨ overlap ∨ covered-by ∨ inside∨ equal∨ cover ∨ contain
meet	meet ∨ overlap∨ covered-by ∨ inside∨ equal ∨ cover ∨ contain

Table 2 Conditions for intermediate nodes (spatial joins)

Consider again the query given in the introduction: "find a triplet of objects (v_1,v_2,v_3) such that v_1 *covers* v_2 and v_2 *meets* v_3". This can be viewed as a *multi-way* spatial join and processed by computing the result of one pairwise join (e.g., v_1 *covers* v_2) using *RTJ*; then joining the results with v_3 by some spatial hash algorithm applicable when only one R-tree is available (since the results of the first join are not indexed). This approach is described in detail in (Mamoulis and Papadias 1999a). Alternatively, as shown in (Papadias, Mamoulis, and Delis 1998), the query could be processed as a CSP, where the query objects (variables) can take values from the corresponding domains.

The hierarchical structure of R-trees can be used to decompose the initial problem (with size $n \cdot \log_2(m)$, where *m* is the cardinality of the datasets) to smaller sub-problems (with size $n \cdot \log_2(C)$, where *C* is the node capacity) at each tree level. A solution of a sub-problem at an intermediate level is an instantiation of variables to entries that may contain objects satisfying the query constraints. Similarly to *RTJ*, the nodes pointed by these entries constitute the domains of the variables at the next (lower) level. In the sequel we describe hierarchical constraint satisfaction using R-trees, and evaluate its performance with systematic and local search.

Hierarchical CSPs using R-trees

Content-based queries like the previous one are transformed to two types of CSPs: one for the intermediate levels and one for the leafs. Formally, a hierarchical CSP using R-trees can be defined by:

[1] *Intersect* (or *non-disjoint*) is the complementary relation of *disjoint*.

- A set of n variables, $v_1,v_2,..,v_n$.
- For each variable v_i a domain D_i which i] for level 0, consists of the entries $\{x_{i,1},.., x_{i,C_i}\}$ of a leaf node X_i, and ii] for levels 1 to h-1, of the entries $\{X_{i,1},.., X_{i,C_i}\}$ of an intermediate node X_i.
- For each pair of variables (v_i,v_j) a binary constraint: i] for level 0, c_{ij} is a disjunction of topological relations restricting the relative positions of v_i and v_j as specified by the query ii] for levels 1 to h-1, C_{ij} is derived by replacing each relation in c_{ij} by the corresponding condition for intermediate nodes in Table 2.

Consider again the example query: the CSP for the top level of the tree in Figure 2 has three variables which can be instantiated to entries of the roots ($D_1=\{A_1,A_2\}$, $D_2=\{B_1,B_2\}$, $D_3=\{C_1,C_2\}$). C_{12} is the entry of Table 2 that corresponds to relation *cover* (i.e., *overlap* $\vee$ *covered-by* $\vee$ *inside* $\vee$ *equal* $\vee$ *cover* $\vee$ *contain*), while C_{23} is the entry that corresponds to *meet*. Out of the 8 possible combinations of root entries (e.g., (A_1,B_1,C_1), (A_1,B_1,C_2), .., (A_2,B_2,C_2)), only (A_1,B_1,C_2) and (A_1,B_2,C_1) may lead to actual solutions. The triplet (A_1,B_1,C_2) constitutes a solution at the root since *overlap*(A_1,B_1) and *overlap*(B_1,C_2) satisfy the intermediate level constraints. Then the algorithm will proceed to level 0 with $D_1=\{a_1,a_2,a_3\}$, $D_2=\{b_1,b_2,b_5\}$ and $D_3=\{c_3,c_4,c_5\}$. The constraints now become c_{12}= *cover* and c_{23}= *meet*. The only leaf level (i.e., actual) solution (a_2,b_2,c_4) is found. On the other hand, the root solution (A_1,B_2,C_1) does not lead to an actual one, i.e., it is a false hit. Figure 3 illustrates the above example, giving the domains, constraints and solutions at each level.

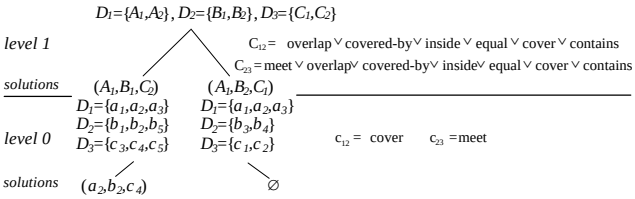


Figure 3 Path to solution (a_2,b_2,c_4)

Space-restriction (Brinkhoff, Kriegel, and Seeger 1993) is a pre-processing heuristic (employed before the application of the CSP algorithm at each level) that scans the domains of all variables, removing the entries that cannot satisfy the query constraints given their positions with respect to the other nodes. If an entry $X_{i,k} \in X_i$ is *disjoint* with Y_j, then it is *disjoint* with all entries contained in Y_j. In order to apply *space-restriction* for topological relations, Table 1 is used for entries at level 0, and Table 2 for the rest. In the example query, when the solution (A_1,B_1,C_2) is found at the top, entry a_1 can be safely pruned from D_1 at the next level since it is *disjoint* with node B_1, therefore it cannot *cover* any entry inside B_1.

Another type of pre-preprocessing is path consistency, which can be employed as a form of semantic query optimization to discard inconsistent queries. For instance, the query c_{12}= *cover*, c_{23}= *cover* and c_{13}= *disjoint* cannot have any solutions. For the detection of such inconsistencies prior to search, we use the composition table for topological relations in (Egenhofer 1991) and the algorithm in (Allen 1983), which does not check value consistency, but *constraint graph consistency*, and its complexity is, therefore, independent from the domain sizes. Note that the algorithm is not complete, i.e., depending on the query, it may not detect all inconsistencies.

Hierarchical constraint satisfaction can be applied with a variety of heuristics. In the next section we measure its performance using representative systematic and local search algorithms for the following three cases:

i] Hierarchical systematic search - Systematic search is applied at every level. This results in an exhaustive depth-first search of the trees.

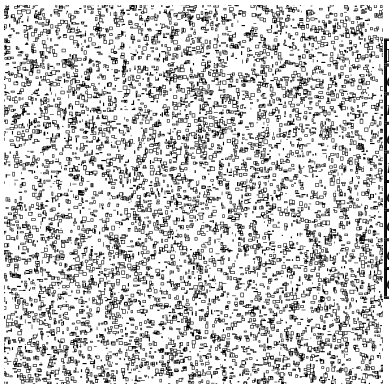
ii] Hierarchical local search - Local search is used at every level. Once a solution is found at level l the algorithm locally searches the references to l-1.

iii] Hierarchical local/systematic search - Local search is applied for the intermediate levels where there exist numerous solutions due to the non-restrictive constraints and the large areas of intermediate nodes. Systematic search is employed at the leafs.

Experiments

The problems were randomly generated by modifying the parameters $<n,m,p_1,p_2>$ (Dechter and Meiri 1994), where p_1 is the probability that a random pair of variables is constrained (*network density*), and p_2 the probability that a random assignment for a constrained pair is inconsistent (*constraint tightness*). The usual methodology for generating random CSPs is to specify each binary constraint as a subset of the domains' Cartesian product. In this case the same tightness for all constraints is achieved by filtering out $p_2 \cdot m^2$ value pairs for every constraint. In the current problem, where constraints are disjunctions of relations, this method cannot be applied.

In order to generate various values of tightness we created uniform datasets with different density values. The density D of a set of rectangles is the average number of rectangles that contain a given point in the workspace. Equivalently, D can be expressed as the ratio of the sum of the areas of all rectangles over the area of the workspace. Figure 4 shows one dataset with $m=10^4$ uniformly distributed rectangles where the average rectangle side in each dimension is $|x| = 0.0045$, resulting in $D \approx 0.2$ (assuming a $[0,1)\times[0,1)$ (unit) workspace, the density is defined as $D = m \cdot |x|^2$). It also illustrates the probability with which a random pair satisfies each topological relation. For uniform datasets, D is the single factor determining relation probabilities, which can be calculated by analytical models (Theodoridis and Sellis, 1996). Sampling and statistical information can be used for real data. Like Figure 4, in most real-life situations *disjoint* is satisfied by the vast majority of object pairs (D=0.2 is a typical value for real datasets). The tightness of a constraint, i.e. the probability that a random pair of variable assignments will violate it, can be calculated from the probabilities of the relations it consists of. For instance, the tightness of a constraint {*meet* $\vee$ *contain*} is 1-(*Prob*(*meet*) + *Prob*(*contain*)). Since in most problems, each constraint has a different tightness, we use the average tightness of all constraints to define p_2.



Relation	probability
disjoint	$9.999182 \cdot 10^{-1}$
meets	$3.760376 \cdot 10^{-6}$
equal	$2.000200 \cdot 10^{-8}$
inside	$2.280228 \cdot 10^{-6}$
covered by	$2.500250 \cdot 10^{-7}$
contains	$2.280228 \cdot 10^{-6}$
covers	$2.500250 \cdot 10^{-7}$
overlap	$7.298730 \cdot 10^{-5}$

Figure 4 A sample dataset used in the experiments

The datasets were organized in R*-trees (Beckmann et al. 1990) with height h=3 and node capacity C=50-200 (depending on cardinality). Performance is measured in terms of consistency checks, i.e., number of object pairs checked for the satisfaction of a topological constraint. In case of hierarchical CSPs, comparisons involving intermediate nodes are also counted as consistency checks.

Hierarchical Systematic Search

We first compare hierarchical and flat versions of systematic search through the whole solution space. In the following experiments we use forward checking (*FC*) with the *fail first* dynamic variable ordering heuristic (Haralick and Elliott 1980), because of its efficiency and relatively simple implementation.

For the first experiment, a series of problem ensembles was generated. An ensemble contains 50 random problems with complete constraint graphs (cliques), i.e., p_1=1, and the same average constraint tightness. The number of variables in all problems is n=5, m=10^4 and D≈0.2. Figure 5a shows the performance of *FC* and hierarchical *FC* (with *space-restriction*) for problems where none of the constraints contain *disjoint* (since *disjoint* is very loose, its existence in a binary constraint is almost equivalent to the absence of the corresponding constraint edge from the graph). The x-axis corresponds to the value of p_2 for each ensemble, and the y-axis shows the mean consistency checks of the algorithms. The number over each ensemble presents the percentage of problems that were soluble.

As a general observation, in the current experimental settings, *H-FC* outperforms *FC* by two orders of magnitude. Observe that in the first ensemble all problems are easy and soluble. This is due to the fact that all constraints contain the relation *overlap*, which is also relaxed compared to the others. The rest of the ensembles were harder and only the second contained soluble problems. This is because the third, and subsequent ensembles involve two or more constraints with *contain*, *inside*, *cover*, *covered-by* or *equal* (but without *disjoint* or *overlap*). The low probability of these relations renders the existence of object pairs (in the same neighborhood) satisfying the two constraints highly unlikely.

The performance of *FC* and *H-FC* was also tested for graphs containing constraints with *disjoint*. Figure 5b illustrates the consistency checks of the two algorithms for a series of problem ensembles in this category. As expected, ensembles with smaller constraint tightness (i.e., many *disjoint* constraints) contain more soluble problems than "dense" ones. For dense graphs the difference is again about two orders of magnitude, but as the tightness decreases the performance of the algorithms converges. This happens because the large number of *disjoint* constraints results in numerous solutions at the intermediate levels, many of which do not lead to actual solutions.

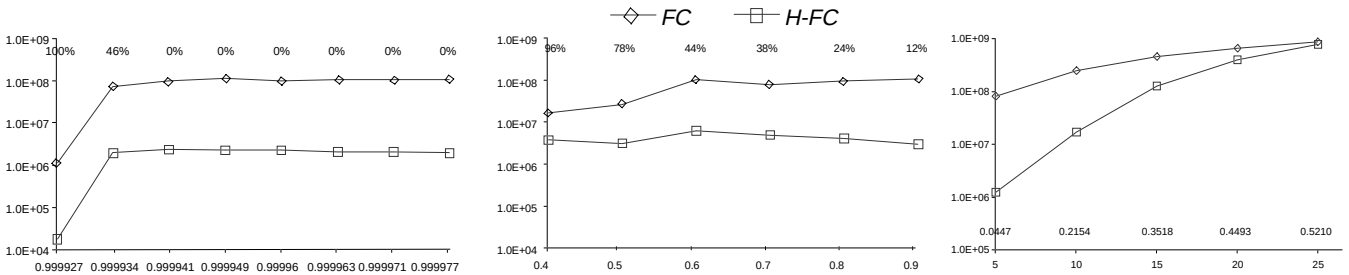
Observe that, due to the special nature of *disjoint*, the problem does not have the *easy-hard-easy* behaviour with respect to the value of p_2, usually observed in other CSPs (Prosser 1996). Rather, both hierarchical and plain systematic search appear to have two distinct behaviors depending on the existence of *disjoint*.

The next experiment tests the effect of the number of variables, when the number of solutions remains constant. We use datasets of 10^4 objects and clique graphs where all constraints are *non-disjoint*. In order to keep the number of solutions stable, the density of the datasets has to be modified for each value of n. According to the analysis in (Papadias, Mamoulis, and Theodoridis 1999), the expected number of solutions for a 2-dimensional problem with a clique constraint graph where all constraints are *non-disjoint*, is given by the following formula:

$$Sol = \prod_{i=1}^{n} m_i \cdot \left(\sum_{i=1}^{n} \cdot \prod_{j=1, j\neq i}^{n} | x_i | \right)^2 \quad (1)$$

where $|x_i|$ is the average MBR extent in each dimension for dataset i. Assuming that all datasets have the same cardinality m and extent $|x|$, eq. 1 can be re-written as:

$$Sol = m^n \cdot n^2 \cdot | x |^{2\cdot(n-1)} \quad (2)$$



(a) checks/p_2 for graphs without *disjoint* (b) checks/p_2 for graphs with *disjoint* (c) checks/n for problems with 1 solution

Figure 5 Comparison of hierarchical and flat FC

Finally, after replacing $|x|$ by the density using $D = m \cdot |x|^2$, eq.2 becomes:

$$Sol = m \cdot n^2 \cdot D^{n-1} \quad (3)$$

Solving eq. 3 with respect to D, one can create synthetic variable domains such that the number of solutions can be controlled. Figure 5c illustrates the consistency checks for *FC* and *H-FC*, as a function of the number of variables for problems that have one solution (such problems usually belong to the hard region). Density is set according to:

$$D = 1 \Big/ \sqrt[n-1]{m \cdot n^2} \quad (4)$$

and its value for each experiment appears on top of *n*. The performance of the algorithms converges as the number of variables increases. For $n>25$, *FC* outperforms *H-FC*.

As the number of variables increases, the performance of *FC* does not deteriorate significantly because most inconsistent instantiations are detected during the early check-forwards. On the other hand, as shown in Table 3, the number of intermediate level solutions explodes with *n*. In general, the percentage of combinations that constitute solutions increases as we go up the levels of the trees because of the large node extents. Since a part of node area corresponds to "dead space" (space not covered by object MBRs) most high level solutions (in this case all but one) are false hits.

#solutions at l=2	159	1559	13567	27331	128781
#solutions at l=1	6430	49670	340480	2314492	15166017
#solutions at l=0	1	1	1	1	1
#variables *n*	5	10	15	20	25

Table 3 Number of solutions as a function of *n*

Similar behaviour is expected for other CSP algorithms including backtracking-based and hybrid algorithms. As a conclusion, hierarchical systematic search significantly outperforms flat search when the domains are large and the number of variables is small (as in most spatial database applications). Some preliminary experiments indicate that *H-FC* also outperforms methods based on pairwise join algorithms (Mamoulis and Papadias 1999a) for finding all solutions of multi-way intersection joins involving dense queries and datasets. The performance gain is higher when only a small subset of the solutions is required. In the next section we apply hierarchical constraint satisfaction with local search.

Hierarchical Local Search

For local search we used hill-climbing with the *min-conflicts* (*MC*) heuristic (Minton et al. 1992; Sosic and Gu 1994). *MC* starts with a random initial assignment for all variables. At each step, it chooses a variable that is currently in conflict and reassigns its value so that the number of conflicts is minimized. This step is repeated until a solution is found or until a deadlock is met, i.e. a local minimum where the number of conflicts can not be further minimized. In this case, the algorithm is restarted.

The hierarchical version of *MC* applies this procedure for each level of the tree. Since a deadlock in the current level may occur because of a false hit at a previous level, the algorithm will backtrack to a higher lever after a number of restarts. In our experiments, this number was proportional to the depth and the size of the problem, i.e., the number of restarts at level l was set to: $(h-l) \cdot n \cdot \log_2(C)$ (in other words, the number of restarts decreases for the upper levels in order to avoid searching false hits). We experimented with the following variations of local search:

a] Flat *MC*: *MC* is applied directly at the leaf level without using the trees.

b] Hierarchical uninformed *MC* (*HU-MC*): *MC* is used at every level. Once a solution is found at level l the algorithm follows the references to l-1. If no solution can be found at l-1, it will backtrack to l, attempting to find another solution and repeat the same process.

c] Hierarchical informed *MC* (*HI-MC*): this is similar to b] but the algorithm keeps a memory of already visited solutions, so when it backtracks from l-1, it will avoid retrieving a solution already found at l.

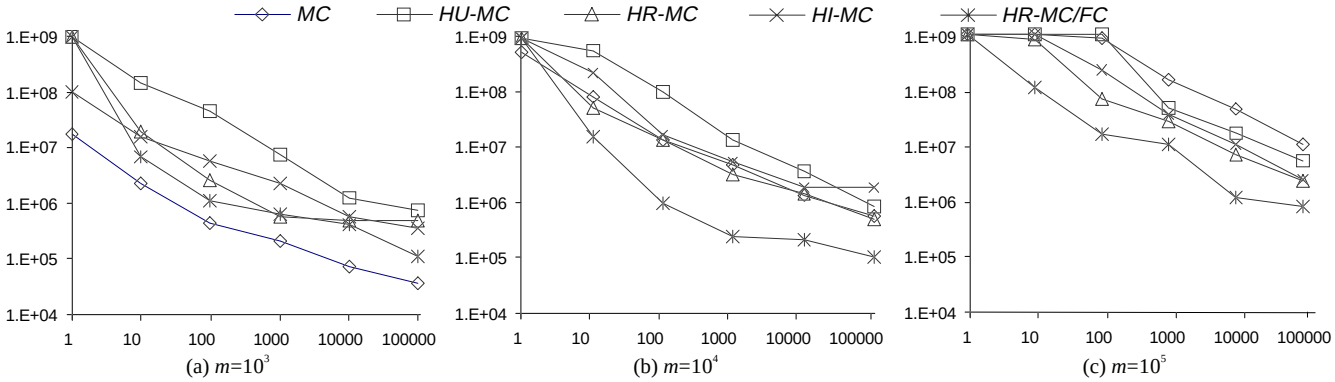
d] Hierarchical root *MC* (*HR-MC*): *MC* is used for every level but when a solution cannot be found at l-1, the algorithm re-starts again directly from the root.

e] Hierarchical root *MC/FC* (*HR-MC/FC*): this is similar to d] but *FC* is used for systematic search at the leaf level.

We experimented with three domain sizes of 10^3, 10^4 and 10^5 objects. In all experiments, the clique query graph contains five variables (n=5) related by *non-disjoint* constraints. The expected number of solutions for each domain size ranges from 1 to 10^5. In order to generate problems with a desired number of solutions we used eq. 3, varying the value of D. All algorithms were executed 10 times for every setting; their execution was terminated if a solution could not be found after 10^9 checks.

Figure 6 shows the mean consistency checks as a function of the number of solutions. Among the pure hierarchical local search techniques (b], c] and d]), *HR-MC* performs best in most cases. Recall that *HR-MC* restarts directly from the root when a deadlock occurs, so it can explore the whole domain more extensively. On the other hand, *HU-MC* has the worst behavior because it consumes a considerable amount of time at the lower levels, misled by false hits. In some cases *HU-MC* is at least one order of magnitude slower that *HR-MC*. *HI-MC*'s performance lies between the previous mentioned algorithms. *HI-MC* searches the domain in the same way as *HU-MC* but, since it keeps a history of the already visited nodes at each level, it avoids entering the same combination of sub-trees more than once.

The comparison of hierarchical local search versus flat search indicates that for small domains (m=10^3), *MC* performs almost one order of magnitude better that the hierarchical algorithms. This happens because flat *MC* avoids the overhead of searching false hits; in addition, it can easily escape from a local minimum since it focuses on the whole domain. This situation changes when dealing with larger domains. For m=10^4, *MC*'s performance is very similar to *HR-MC*. For m=10^5, *MC* is outperformed by *HR-MC* by one order of magnitude because the overhead imposed by the hierarchical structure is less than the effort required for searching in a large unstructured domain.



(a) $m=10^3$ (b) $m=10^4$ (c) $m=10^5$

Figure 6 Performance of local search algorithms (consistency checks/number of solutions)

Due to the large number of solutions at the upper tree levels, hierarchical local search succeeds fast, but spends more time trying to find a solution at the leaf level. This motivated the replacement of *MC* at leaf levels with *FC*. For a small domain, *HR-MC/FC* achieves only a marginal performance gain with respect to *HR-MC*, while for larger domains it is almost an order of magnitude faster.

Conclusion

This paper describes a methodology for hierarchical constraint satisfaction in spatial databases using R-trees. Instead of processing content-based queries as flat CSPs, computation can be decomposed in smaller problems at each tree level. The experimental evaluation suggests that systematic search is significantly faster in the case of hierarchical CSPs for typical conditions ($m \geq 10^4$ and $n \leq 10$). On the other hand, hierarchical local search pays-off only for very large domains ($m \approx 10^5$).

Although we experimented with two representative algorithms (*FC*, *MC*), hierarchical constraint satisfaction can be used with a variety of systematic and local search techniques. Several heuristics, like plain sweep, can take advantage of the inherent order of domains to restrict search. Furthermore, these methods can be applied with other spatial access methods based on the hierarchical decomposition of space.

An alternative approach for solving spatial CSPs, is to exploit the data structure in order to avoid exhaustive search of domains while assigning or pruning values. Going back to the example query, once v_1 is instantiated to some MBR x_1, x_1 becomes the query window for retrieval of all objects satisfying $cover(x_1,v_2)$. In this way linear scan of domains at each instantiation is replaced by window queries which are very cheap operations in R-trees. An application of this technique with forward checking and backtracking in the context of temporal CSPs can be found in (Mamoulis and Papadias 1999b). Furthermore, this method can be combined with hierarchical constraint satisfaction, e.g., for the example query we could use hierarchical search to retrieve qualifying pairs of values for (v_1,v_2), and for each such pair apply window queries to retrieve consistent values for v_3. The optimal combination can be based on cost models and appropriate analytical formulae (Papadias, Mamoulis, and Theodoridis 1999) for the expected number of solutions.

Acknowledgements

This work was supported by grant HKUST 6151/98E from Hong Kong RGC and grant DAG97/ 98.EG02. We would like to thank Kostas Stergiou and Marios Mantzourogiannis for their comments.

References

Allen, J.F. Maintaining Knowledge about Temporal Intervals. *CACM 26* (11), 832-843, 1983.

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.

Brinkhoff, T., Kriegel, H. Seeger, B. Efficient Processing of Spatial Joins Using R-trees. *ACM SIGMOD*, 1993.

Dechter, R., Meiri, I. Experimental Evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence* 68(2): 211-241, 1994.

Egenhofer, M. Reasoning about Binary Topological Relations. In Günther, O. and Schek, H.J. (eds.) *Advances in Spatial Databases*. Springer Verlag LNCS, 1991.

Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *ACM SIGMOD*, 1984.

Haralick, R., Elliott, G. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* 14(3): 263-313, 1980.

Minton, S., Johnston, M., Philips, A., Laird, P. Minimizing Conflicts: A Heuristic Method for Constraint-Satisfaction and Scheduling Problems. *Artificial Intelligence* 58, 161-205, 1992.

Mamoulis, N., Papadias, D. Integration of Spatial Join Algorithms for Processing Multiple Inputs. *ACM SIGMOD*, 1999a.

Mamoulis, N., Papadias, D. Improving Search Using Indexing: a Study with Temporal CSPs. *IJCAI*, 1999b.

Papadias, D., Mamoulis, N., Delis, V. Querying by Spatial Structure. *VLDB*, 1998.

Papadias, D., Mamoulis, N., Theodoridis, Y., Processing and Optimization of Multi-way Spatial Joins Using R-trees. *ACM PODS*, 1999.

Prosser, P. An Empirical Study of Phase Transitions in Binary Constraint Satisfaction Problems. *Artificial Intelligence*, 81 (1-2), 1996.

Randell, D., Cui, Z., Cohn., A. A Spatial Logic Based on Regions and Connection. *Knowledge Representation and Reasoning*, 1992.

Sosic, R., Gu, J. Efficient Local Search with Conflict minimization: A Case Study of the n-Queens Problem. *IEEE Transactions on Knowledge and Data Engineering*, 6(5): 661-668, 1994.

Theodoridis, Y., Sellis, T. A Model for the Prediction of R-tree Performance. *ACM PODS*, 1996.