Approximate Processing of Multiway Spatial Joins in Very Large Databases

Dimitris Papadias

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

Dinos Arkoumanis

Dept of Electrical and Computer Engineering
National Technical University of Athens
Greece, 15773
dinosar@dbnet.ece.ntua.gr

Abstract. Existing work on multiway spatial joins focuses on the retrieval of all exact solutions with no time limit for query processing. Depending on the query and data properties, however, exhaustive processing of multiway spatial joins can be prohibitively expensive due to the exponential nature of the problem. Furthermore, if there do not exist any exact solutions, the result will be empty even though there may exist solutions that match the query very closely. These shortcomings motivate the current work, which aims at the retrieval of the best possible (exact or approximate) solutions within a time threshold, since fast retrieval of approximate matches is the only way to deal with the ever increasing amounts of multimedia information in several real time systems. We propose various techniques that combine local and evolutionary search with underlying indexes to prune the search space. In addition to their usefulness as standalone methods for approximate query processing, the techniques can be combined with systematic search to enhance performance when the goal is retrieval of the best solutions.

1. Introduction

Several specialized access methods have been proposed for the efficient manipulation of multi-dimensional data. Among the most popular methods, are the R-tree [G84] and its variations, currently used in several commercial products (e.g., Oracle, Informix). R-trees are extensions of B-trees in two or more dimensions. Figure 1 shows an image containing objects r_1,.., r_8, and the corresponding R-tree assuming capacity of three entries per node. Leaf entries store the minimum bounding rectangles (MBRs) of the actual objects, and nodes are created by grouping entries in a way that preserves proximity, and avoids excessive overlap between nodes. In Figure 1, objects r_1, r_2, and r_3, are grouped together in node e_1, objects r_4, r_5, and r_6 in node e_2, etc.

In this work we consider multimedia databases involving maps/images containing a large number (in the order of 10^5-10^6) of objects with well-defined semantics (e.g., maps created through topographic surveys, VLSI designs, CAD diagrams). Each map/image is not stored as a single entity, but information about objects is kept in relational tables with a spatial index for each type of objects covering the same area (e.g., an R-tree for the roads of California, another for residential areas etc). This facilitates the processing of traditional spatial selections (e.g., find all roads inside a query window) and spatial joins [BKS93] (e.g., find all pairs of intersecting roads and railroad lines in California).

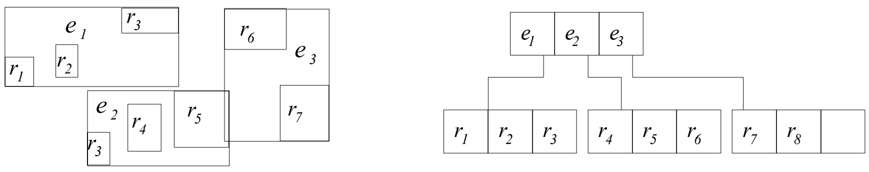


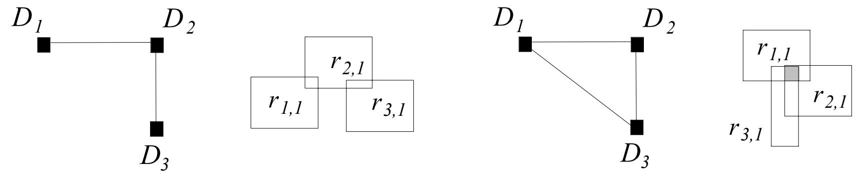
Fig. 1. R-tree example

A multiway spatial join is the natural extension of pairwise joins, combining more than two inputs (e.g., find all triplets $<c,r,a>$ of cities c, rivers r and industrial areas a such that c is *crossed by* r which also *crosses* a). Multiway spatial joins constitute a special form of content based retrieval, which focuses on spatial relations rather than visual characteristics such as color, shape or texture. In accordance with related work on content based retrieval, query processing mechanisms should be able to handle approximate retrieval using some measure of similarity. Unfortunately existing techniques (presented in the next section) only deal with exact retrieval, i.e., if there does not exist some stored configuration matching exactly the query constraints, the result will be empty (even if there exist very close matches). Furthermore, these techniques assume that there is unlimited time for query processing. This assumption may not always be valid, given the exponential nature of the problem and the huge amounts of multimedia information in several applications.

This paper deals with these problems by proposing methods that can handle approximate retrieval of multiway spatial joins under limited time. The methods combine well known search heuristics, such as local and evolutionary search, with spatial indexing (R-trees) in order to quickly locate good, but not necessarily optimal, solutions. In addition to their usefulness as standalone methods, they can be employed in conjunction with systematic search to speed up retrieval of the optimal solutions by orders of magnitude. The rest of the paper is structured as follows: section 2 overviews related work in the context of multiway spatial joins and content-based retrieval; sections 3, 4 and 5, present algorithms that combine R-trees with local search, guided local search and genetic algorithms, respectively. Section 6 contains the experimental evaluation and section 7 concludes with a discussion about future work.

2. Definitions and Related Work

Formally, a multiway spatial join can be expressed as follows: Given n multidimensional datasets $D_1, D_2, \dots D_n$ and a *query* Q, where Q_{ij} is the spatial predicate that should hold between D_i and D_j, retrieve all n-tuples $\{(r_{1,w},..,r_{i,x},..,r_{j,y},..,r_{n,z}) \mid \forall\ i,j : r_{i,x} \in D_i, r_{j,y} \in D_j \text{ and } r_{i,x}\ Q_{ij}\ r_{j,y}\}$. Such a query can be represented by a graph where nodes correspond to datasets and edges to join predicates. Equivalently, the graph can be viewed as a *constraint network* [DM94] where the nodes are problem variables, and edges are binary spatial constraints. In the sequel we use the terms variable/dataset and constraint/join condition interchangeably. Following the common methodology in the spatial database literature we assume that the standard join condition is *overlap* (*intersect*, *non-disjoint*). Figure 2 illustrates two query graphs joining three datasets and two solution tuples ($r_{1,1}$,$r_{2,1}$,$r_{3,1}$) such that $r_{i,1}$ is an object in D_i. Figure 2a corresponds to a chain query (e.g., "find all cities *crossed by* a river which *crosses* an industrial area"), while 2b to a clique ("the industrial area should also intersect the city").



(a) chain query (b) clique query

Fig. 2. Examples of multiway spatial joins

We use the notation $v_i \leftarrow r_{i,x}$ to express that variable v_i is instantiated to rectangle $r_{i,x}$ (which belongs to domain D_i). A *binary instantiation* $\{v_i \leftarrow r_{i,x}, v_j \leftarrow r_{j,y}\}$ is *inconsistent* if there is a join condition Q_{ij}, but $r_{i,x}$ and $r_{j,y}$ do not *overlap*. A solution is a set of *n* instantiations $\{v_1 \leftarrow r_{1,w}, .., v_i \leftarrow r_{i,x}, .., v_j \leftarrow r_{j,y}, .., v_n \leftarrow r_{n,z}\}$ which, for simplicity, can be described as a tuple of values $(r_{1,w},..,r_{i,x},..,r_{j,y},..,r_{n,z})$ since the order of variables is implied. The *inconsistency degree* of a solution is equal to the total number of inconsistent binary instantiations, i.e., the number of join conditions violated. A solution with zero inconsistency degree is *exact*; otherwise it is *approximate*. The lower the inconsistency degree, the higher the *similarity* of the solution.

Approximate retrieval of the solutions with the highest similarity (i.e., solutions with the minimum number of inconsistencies / join condition violations) is desirable in several applications (e.g., architectural and VLSI design), where queries often involve numerous variables and the database may not contain configurations that match all the query constraints. Furthermore, since the problem is in general exponential, even if there exist exact solutions, there may not be enough processing time or computational resources to find them.

As in the case of relational joins, multiway spatial joins can be processed by combining pairwise join algorithms. The *pairwise join method* (*PJM*) [MP99] considers a join order that is expected to result in the minimum cost (in terms of page accesses). However, *PJM* (and any method based on pairwise algorithms) cannot be extended for approximate retrieval since if no exact solution exists, the results of some joins should contain non-intersecting pairs of objects. Two alternative methodologies for multiway spatial joins, motivated by algorithms for constraint satisfaction problems (CSPs), were proposed in [PMT99]. *Synchronous traversal* (*ST*) starts from the roots of the R-trees and finds combinations of entries that satisfy the query constraints. For each such combination the algorithm is recursively called, taking the references to the underlying nodes as parameters, until the leaf level is reached. The calculation of combinations of the qualifying nodes for each level is expensive, as their number can be as high as C^n (where *C* is the node capacity, and *n* the number of query variables). In order to avoid exhaustive search of all combinations, the authors use search algorithms and optimization techniques.

The second methodology, called *window reduction* (*WR*) integrates the ideas of backtracking and index nested loop algorithms. When the first variable gets a value $v_1 \leftarrow r_{1,w}$, this rectangle ($r_{1,w}$) is used as a query window to find qualifying entries in the second dataset. If such entries are found, the second variable is instantiated to one of them, i.e., $v_2 \leftarrow r_{2,w}$. The algorithm proceeds forward using the values of instantiated variables as query windows. When a window query yields no results, the algorithm backtracks to the previous variable and tries another value for it. Although *ST* and *WR* can be applied for retrieval of approximate solutions, they are not suitable for query processing within a time limit (see experimental evaluation), since they may initially spend a lot of time in regions with low quality solutions, failing to quickly find some good ones.

Also related to this paper, is previous work on spatial (or configuration) similarity retrieval. The corresponding queries describe some prototype configuration and the goal is to retrieve arrangements of objects matching the input exactly or approximately. Thus, these queries can be thought of as approximate multiway spatial joins with arbitrary predicates. Petrakis and Faloutsos [PF97] solve such problems by mapping images (datasets) and queries into high-dimensional points, which are then indexed by R-trees. Similarity retrieval is processed by nearest neighbor search. The method, however, assumes medical images with about 10 objects and cannot be employed for even the smallest datasets normally found in spatial databases. In general, techniques based on high-dimensional indexing and nearest neighbor search are not applicable due to the huge number of dimensions required to represent the problem.

A number of techniques are based on several variations of 2D strings [LYC92, LH92], which encode the arrangement of objects on each dimension into sequential structures. Every database image is indexed by a 2D string; queries are also transformed to 2D strings, and similarity retrieval is performed by applying appropriate string matching algorithms [CSY87]. Although this methodology can handle larger datasets (experimental evaluations usually include images with about 100 objects) it is still not adequate for real-life spatial datasets.

In order to deal with similarity retrieval under limited time, Papadias et al., [PMK+99] use heuristics based on local search, simulated annealing and genetic algorithms. Unlike *ST* and *WR,* which search systematically, guaranteeing to find the best solutions, these heuristics are non-systematic (i.e., random). The evaluation of [PMK+99] suggests that local search, the most efficient algorithm, can retrieve good solutions even for large problems (images with about 10^5 objects). In the next section we propose heuristics based on similar principles, for inexact retrieval of multiway spatial joins. However, unlike [PMK+99] where the algorithms were a straightforward adaptation of local and evolutionary search for similarity retrieval, the proposed methods take advantage of the spatial structure of the problem and existing indexes to achieve high performance.

For the rest of the paper we consider that all datasets are indexed by R*-trees [BKSS90] on minimum bounding rectangles, and we deal with intersection joins. We start with indexed local search in the next section, followed by guided indexed local search and a spatial evolutionary algorithm.

3. Indexed Local Search

The search space of multiway spatial joins can be considered as a graph, where each solution corresponds to a node having some inconsistency degree. If all *n* datasets have the same cardinality *N*, the graph has N^n nodes. Two nodes/solutions are connected through an edge if one can be derived from the other by changing the instantiation of a single variable. Excluding its current assignment, a variable can take *N*-1 values; thus, each solution has $n \cdot (N-1)$ neighbors. A node that has lower inconsistency degree than all its neighbors, is a *local maximum*. Notice that a local maximum is not necessarily a global maximum since there may exist solutions with higher similarity in other regions of the graph.

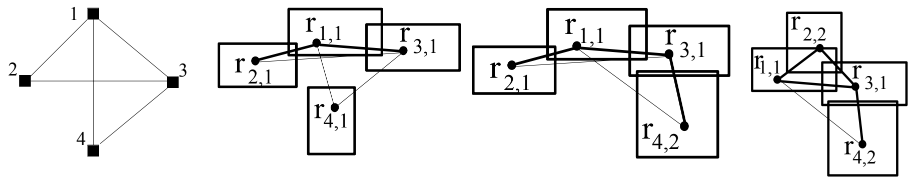
Local search methods start with a random solution called *seed*, and then try to reach a local maximum by performing uphill moves, i.e., by visiting neighbors with higher similarity. When they reach a local maximum (from where uphill moves are not possible) they restart the same process from a different seed until the time limit is exhausted. Throughout this process the best solutions are kept. Algorithms based on this general concept have been successfully employed for a variety of problems. *Indexed local search* (*ILS*) also applies this idea, but uses R*-trees to improve the solutions. The pseudocode of the algorithm is illustrated in Figure 3.

```
Indexed Local Search
WHILE NOT (Time limit) {
  S := random seed
  WHILE NOT(Local_Maximum) {
        determine worst variable $v_i$
        value := find best value (Root of tree $R_i$, $v_i$)
        IF better value THEN S = S ∧ { $v_i$ ← value }
        IF S is the best solution found so far THEN bestSolution=S
  } /* END WHILE NOT Local_Maximum */
} /* END WHILE NOT Time Limit */
```

Fig. 3. Indexed local search

Motivated by conflict minimization algorithms [MJP+92], we choose to re-instantiate the "worst" variable, i.e., the one whose current instantiation violates the most join conditions. In case of a tie we select the one that participates in the smallest number of satisfied constraints. If the worst variable cannot be improved, the algorithm considers the second worst; if it cannot be improved either, the third worst, and so on. If one variable can be improved, the next step will consider again the new worst one; otherwise, if all variables are exhausted with no improvement, the current solution is considered a local maximum.

Consider, for example the query of Figure 4a and the approximate solution of Figure 4b. The inconsistency degree of the solution is 3 since the conditions $Q_{1,4}$, $Q_{2,3}$, and $Q_{3,4}$ are violated (in Figure 4b satisfied conditions are denoted with bold lines and violated ones with thin lines). Variables v_3 and v_4 participate in two violations each; v_3, however, participates in one satisfied condition, so v_4 is chosen for re-assignment. *Find best value* will find the best possible value for the variable to be re-instantiated, i.e., the rectangle that satisfies the maximum number of join conditions given the assignments of the other variables. In the example of Figure 4b, the best value for v_4 should overlap both $r_{1,1}$ and $r_{3,1}$. If such a rectangle does not exist, the next better choice should intersect either $r_{1,1}$, or $r_{3,1}$.



(a) example query (b) initial solution (c) re-instantiation of v_4 (d) local max

Fig. 4. Example of ILS

The pseudo-code for *find best value* is illustrated in Figure 5; the variable to be re-instantiated is v_i. Essentially this is like a *branch-and-bound window query*, where there exist multiple windows and the goal is to retrieve the rectangle that intersects most of them. The windows are the assignments of all variables v_j such that Q_{ij}=True (for the current example there exist two windows $w_{1=}r_{1,1}$ and $w_{3=}r_{3,1}$).

```
maxConditions=0
bestValue=∅

Find best value (Node N, integer i)
FOR EACH entry $e_x$ of N
  FOR EACH $Q_{ij}$ such that $Q_{ij}$ = True
     IF $e_x$ intersects $w_j$ THEN $conditions_x=conditions_x+1$
Sort all entries $e_x$ such that $conditions_x>0$ with respect to $conditions_x$
IF N intermediate node
  FOR each $e_x$ in the sorted list
     IF $conditions_x$ > maxConditions THEN
               Find best value ($e_x$, i)
 ELSE //leaf node
  FOR each $e_x$ in the sorted list
               IF $conditions_x$ > maxConditions THEN
                       maxConditions=$conditions_x$
                       bestValue=$e_x$
```

Fig. 5. *Find best value* algorithm

The algorithm starts from the root of the corresponding tree and sorts the entries according to the conditions they satisfy (i.e., how many windows they overlap). The entries with the maximum number are visited first because their descendants are more likely to intersect more windows. At the leaf level, an entry is compared with the maximum number of conditions found so far (*maxConditions*). If it is better, then this is kept as *bestValue*, and *maxConditions* is updated accordingly. Notice that if an intermediate node satisfies the same or a smaller number of conditions than *maxConditions*, it cannot contain any better solution and is not visited.

Figure 6 illustrates this process for the example of Figures 4a and b. *Find best value* will retrieve the rectangle (in the dataset corresponding to v_4) that intersects the maximum number of windows, in this case $w_1=r_{1,1}$ and $w_3=r_{3,1}$. Suppose that the node (of R-tree R_4) considered has three entries e_1, e_2 and e_3; e_1 is visited first because it overlaps both query windows. However, no good values are found inside it so *max-Conditions* remains zero. The next entry to be visited is e_2 which contains a rectangle ($r_{4,2}$) that intersects w_3. *MaxConditions* is updated to 1 and e_3 will not be visited since it may not contain values better than $r_{4,2}$ (it only satisfies one condition). $r_{4,2}$ becomes the new value of v_4 and the inconsistency degree of the new solution (Figure 4c) is 2 ($Q_{3,4}$ is now satisfied). At the next step (Figure 4d), a better value (let $r_{2,2}$) is found for v_2 using *find best value* (R_2,2). At this point, the algorithm reaches a local maximum. The violation of $Q_{1,4}$ cannot be avoided since, according to Figure 6, there is no object in the fourth dataset that intersects both $r_{1,1}$ and $r_{3,1}$.

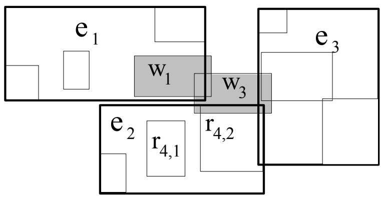


Fig. 6. Example of *find best value*

4. Guided Indexed Local Search

There have been many attempts to include some deterministic features in local search and achieve a more systematic exploration of the problem space. "Memory" mechanisms guarantee that the algorithm will not find the same nodes repeatedly by keeping a list of visited nodes [GL97]. These nodes become forbidden (*tabu*) in the graph, forcing the algorithms to move to new neighborhoods. A limitation of this approach for the current problem is the huge number of nodes, since there exist N^n solutions a significant percentage of which may be visited. Other approaches [DTW+94] try to avoid revisiting the same maxima by storing their features (e.g., the route length in traveling salesman problem). Solutions matching these features, are not rejected, but "punished". As a result, the probability of finding the same maximum multiple times is decreased. The trade-off is that unrelated nodes that share features with visited local maxima are avoided too, and good solutions may be missed.

Guided indexed local search (*GILS*) combines the above ideas by keeping a memory, not of all solutions visited, but of the variable assignments at local maxima. When a local maximum ($r_{1,w}$,..,$r_{i,x}$,..,$r_{j,y}$,.., $r_{n,z}$) is found, some of the assignments $v_1 \leftarrow r_{1,w}$, .., $v_i \leftarrow r_{i,x}$, .., $v_j \leftarrow r_{j,y}$, .., $v_n \leftarrow r_{n,z}$ get a penalty. In particular, *GILS* penalizes the assignments with the minimum penalties so far; e.g., if $v_1 \leftarrow r_{1,w}$ already has a punishment from a previous maximum (while the others do not), only the other assignments are penalized in order to avoid over-punishing $v_1 \leftarrow r_{1,w}$.

The code for *GILS* (Figure 7) is similar to *ILS* since both algorithms re-instantiate the worst variable for improving the current solution. Their difference is that *GILS* only generates one random seed during its execution and has some additional code for penalty assignment. The penalty is used to increase the inconsistency degree of the current local maximum, and to a lesser to degree of solutions that include a subset of the assignments. In particular, for its similarity computations *GILS* applies the *effective inconsistency degree* which is computed by adding the penalties

$$\lambda \cdot \sum_{i=1}^{n} \text{penalty}(v_i \leftarrow r_{i,x})$$

to the actual inconsistency degree (i.e., the number of condition violations) of a solution. The *penalty weight parameter* λ is a constant that tunes the relative importance of penalties and controls the effect of memory in search. A large value of λ will punish significantly local maxima and their neighbors causing the algorithm to quickly visit other areas of the graph. A small value will achieve better (local) exploration of the neighborhoods around maxima at the expense of global graph exploration.

```
Guided Indexed Local Search
S := random seed
WHILE NOT (Time limit) {
     WHILE NOT(Local_Maximum) {
        determine worst variable $v_i$
        value := find best value (Root of tree $R_i$, $v_i$)
        IF better value THEN S = S ∧ {$v_i$ ← value}
        IF S is the best solution found so far THEN bestSolution=S
     /* END WHILE NOT Local_Maximum */
     P = among the assignments of the current local maximum, select the
     ones with the minimum penalty
     FOR EACH assignment  $v_i$ ← $r_{i,x}$ in  P
        penalty($v_i$ ← $r_{i,x}$)= penalty($v_i$ ← $r_{i,x}$) + 1
/* END WHILE NOT Time limit */
```

Fig. 7. Guided indexed local search

The results of this punishment process are:

- search does not restart from various random seeds but continues from local maxima. This is because the penalty increases the effective inconsistency degree of the current local maximum (sometimes repeatedly) and eventually worse neighbors appear to be better and are followed by *GILS*. The intuition behind this is to perform some downhill moves, expecting better local maxima in subsequent steps.
- solutions that share many common assignments with one or more local maxima have high effective inconsistency degrees and usually are not chosen during search. Thus, possible visits to the same regions of the search space are avoided.

Like *ILS*, *GILS* uses *find best value*, to select the new object for the variable to be re-instantiated. The process is modified in order to deal with penalties as follows: after the calculation of the inconsistency degree of a leaf object, the penalty value of this assignment is added, and compared with the best found so far. *Find best value* is identical with the one for *ILS* when it operates at intermediate nodes.

For small problems, the penalties are kept in a two dimensional ($n \cdot N$) array where the cell (i,j) stores the penalty of assigning the i^{th} variable with the j^{th} value in its domain ($v_i \leftarrow r_{i,j}$). This array is, in general, very sparse since only a small subset of the possible assignments are penalized (most of the cells contain zeros). For large problems, where there is not enough main memory to keep such an array, a hash table (which only stores the assignments with positive penalties) can be built in-memory.

5. Spatial Evolutionary Algorithm

Evolutionary algorithms are search methods based on the concepts of natural mutation and the survival of the fittest individuals. Before the search process starts, a set of p solutions (called initial population P) is initialized to form the first generation. Then, three genetic operations, *selection, crossover* and *mutation*, are repeatedly applied in order to obtain a population (i.e., a new set of solutions) with better characteristics. This set will constitute the next generation, at which the algorithm will perform the same actions and so on, until a stopping criterion is met. In this section we propose a *spatial evolutionary algorithm* (*SEA*) that takes advantage of spatial indexes and the problem structure to improve solutions.

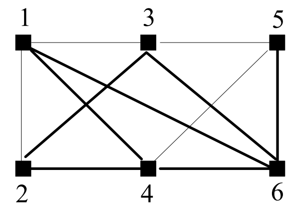
Selection mechanism: This operation consists of two parts: *evaluation* and *offspring allocation*. Evaluation is performed by measuring the similarity of every solution; offspring generation then allocates to each solution, a number of offspring proportional to its similarity. Techniques for offspring allocation include *ranking*, *proportional selection*, *stochastic remainder* etc. The comparison of [BT96] suggests that the *tournament* method gives the best results for a variety of problems and we adopt it in our implementation. According to this technique, each solution S_i competes with a set of T random solutions in the generation. Among the T+1 solutions, the one with the highest similarity replaces S_i. After offspring allocation, the population contains multiple copies of the best solutions, while the worst ones are likely to disappear.

Crossover mechanism is the driving force of *exploration* in evolutionary algorithms. In the simplest approach [H75], pairs of solutions are selected randomly from the population. For each pair a *crossover point* is defined randomly, and the solutions beyond it are mutually exchanged with probability μ_c (*crossover rate*), producing two new solutions. The rationale is that after the exchange of genetic materials, the two newly generated solutions are likely to possess the good characteristics of their parents (*building-block hypothesis* [G89]). In our case randomly swapping assignments will most probably generate multiple condition violations. Especially for latter generations, where solutions may have reached high similarities, random crossover may lead to the removal of good solutions.

In order to limit the number of violations, we propose a variable crossover point c ($1 \leq c < n$) which is initially 1 and increases every g_c generations. When a solution S is chosen for crossover, c variables will retain their current assignments, while the remaining n-c will get the assignments of another solution. A small value of c means that S will change dramatically after crossover, while a value close to n implies that only a small part will be affected (e.g., if $c=n-1$ only one variable will change its assignment). This leads to the desired result that during the early generations, crossover has a significant effect in generating variety in the population, but this effect diminishes with time in order to preserve good solutions.

Given the value of c, a greedy crossover mechanism uses a set X to store the c best variables in S, i.e., the ones that have relatively low inconsistency degrees and should not change their assignment during crossover. Initially variables are sorted according to the number of satisfied join conditions in S. In case of ties the variable with the smallest number of violations has higher priority. The first variable in the ordered list is inserted into X. From this point on, the variable inserted, is the one that satisfies the largest number of conditions with respect to variables already in X. Ties are resolved using the initial order. The process stops when c variables are in X. The rest of the variables are re-instantiated using the corresponding values of a another solution.

Figure 8 illustrates a solution where satisfied (violated) conditions are denoted with bold (thin) lines. Assume that c=3, meaning that three variables will keep their current assignments. The initial sorting will produce the order (v_6,v_4,v_2,v_1,v_3,v_5). The insertion order in X is v_6, then v_4 (because of $Q_{4,6}$) and finally v_1 (because of $Q_{1,6}$ and $Q_{1,4}$). Intuitively this is a very good splitting because the sub-graph involving v_1,v_4 and v_6 is already solved. Now another solution is chosen at random and v_2,v_3 and v_5 obtain the instantiations of this solution.



Variable	Conditions Satisfied	Conditions Violated
v_6	4	0
v_4	3	1
v_2	2	1
v_1	2	2
v_3	2	2
v_5	1	2

Fig. 8. Example of solution splitting during crossover

Mutation mechanism: Although it is not the primary search operation and sometimes is omitted, mutation is very important for *SEA* and the only operation that uses the index. At each generation, mutation is applied to every solution in the population with probability μ_m, called the *mutation rate*. The process is similar to *ILS*; the worst variable is chosen and it gets a new value using *find best value*. Thus, in our case mutation can only have positive results.

Figure 9 illustrates the pseudo-code for *SEA*. The algorithm first computes the similarity of the solutions, and then performs offspring allocation (using the tournament approach), crossover and mutation (following the methods described above) in this order. During the initial generations crossover plays an important role in the formation of new solutions. As time passes its role gradually diminishes and the algorithm behaves increasingly like *ILS*, since mutation becomes the main operation that alters solutions.

```
Spatial Evolutionary Algorithm
P := generate initial set of solutions {$S_1$,..,$S_p$}
WHILE NOT (Time limit) {
  compute crossover point c /* increase c every $g_c$ generations */
  FOR EACH $S_i$ in P /*evaluation */
     evaluate $S_i$
     IF $S_i$ is the best solution found so far THEN keep $S_i$
  FOR EACH $S_i$ in P /* offspring allocation */
     compare $S_i$ with T other random solutions in P
     replace $S_i$ with the best among the T+1 solutions
  FOR EACH $S_i$ in P /*crossover*/
     with probability $\mu_c$ change  $S_i$ as follows
     determine set of c variables to keep their current values
     re-instantiate the remaining variables using their values in an-
     other solution $S_j$ ($S_j \in$ P)
  FOR EACH $S_i$  in P  /* mutation */
     with probability $\mu_m$ change  $S_i$ as follows
     determine worst variable $v_k$
     $v_k \leftarrow$ find best value (Root of tree $R_k$, $v_k$)
}/* END WHILE NOT Time limit */
```

Fig. 9. Spatial evolutionary algorithm

Unlike *ILS* (which does not include any problem specific parameters), and *GILS* (which only contains λ), *SEA* involves numerous parameters, namely, the number T of solutions participating in the tournament, the crossover (μ_c) and mutation (μ_m) rates, the number of generations g_c during which the crossover point remains constant, and the number p of solutions in each population. Furthermore, these parameters are inter-related in the sense that the optimal value for one depends on the rest. Careful tuning of the parameters is essential for the good performance of *SEA*, and evolutionary algorithms in general [G86].

Based on extensive experiments (described in the long version of this paper [PA]) we chose the following set of parameters to be used in the subsequent experimental comparison: $\lambda=10^{-10}\cdot s$, $T=0.05\cdot s$, $\mu_c= 0.6$, $g_c=10\cdot s$, $\mu_m=1$, and $p=100\cdot s$, where s is the size of a problem and corresponds to the number of bits required to represent the search space [CFG+98], i.e., the number of bits needed to express all possible solutions:

$$s = \log_2 \cdot \prod_{i=1}^{n} N_i$$

The tuning of parameters as a function of s, provides good performance independently of the problem size. Although even better parameter values could be obtained for specific problem instances, the above set achieves good overall performance for a variety of query graphs and datasets.

6. Experimental Evaluation

In this section we compare the proposed techniques, according to the common CSP and optimization methodology, using problem instances in the, so-called, *hard region*. It is a well known fact, that over-constrained problems do not have exact solutions and it is usually easy to determine this. On the other hand, under-constrained problems have numerous solutions which can be easily found. Between these types occurs a *phase transition*. Several studies on systematic [CA93] and non-systematic search [CFG+98] in a variety of combinatorial problems, experimentally demonstrate that the most difficult problems to solve are in the (hard) region defined by the phase transition. This hard region occurs when the expected number of exact solutions is small, i.e., in the range [1,10].

In order to generate such problem instances we need analytical formulae for the number of exact solutions. The general formula for the expected output size of multiway spatial joins is: *Sol* = #(possible tuples)·*Prob*(a tuple is a solution), where the first part of the product equals the cardinality of the Cartesian product of the *n* domains, while the second part corresponds to multiway join selectivity. According to [TSS98] the selectivity of a pairwise join over two uniform datasets D_i and D_j that cover a unit workspace is $(|r_i|+|r_j|)^2$, where $|r_i|$ is the average MBR extent in each dimension for D_i. For acyclic graphs, the pairwise probabilities of the join edges are independent and selectivity is the product of pairwise join selectivities. Thus, in this case the number of exact solutions is:

$$Sol = \prod_{i=1}^{n} |N_i| \cdot \prod_{\forall i,j:Q(i,j)=TRUE} \left(|r_i| + |r_j|\right)^2$$

When the query graph contains cycles, the pairwise selectivities are not independent anymore and the above equation is not accurate. Based on the fact that if a set of rectangles mutually overlap, then they must share a common area, [PMT99] propose the following estimation for *Sol*, in case of clique joins:

$$Sol = \prod_{i=1}^{n} |N_i| \cdot \left(\sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} |r_j| \right)^2$$

The above formulae are applicable for queries that can be decomposed to acyclic and clique graphs. For simplicity, in the rest of the paper we assume that all (uniform) datasets have the same cardinality *N* and MBR extents $|r|$. Under these assumptions, and by substituting average extents with density[1] values, the formulae can be transformed as follows. For acyclic queries, there are *n*-1 join conditions. Thus, the number

[1] The density *d* of a dataset is the average number of rectangles that contain a 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. Density is related with the average rectangle extent $|r|$ by the equation $d = N \cdot |r|^2$ [TSS98]. Obviously the number of solutions increases with density (since larger MBRs have a higher chance to overlap) and decreases with the number of join conditions in the query.

of solutions is: $Sol = N^n \cdot (2 \cdot |r|)^{2\cdot(n-1)} = N \cdot 2^{2\cdot(n-1)} \cdot d^{n-1}$. Similarly, for cliques the number of solutions is: $Sol = N^n \cdot n^2 \cdot |r|^{2\cdot(n-1)} = N \cdot n^2 \cdot d^{n-1}$. The importance of these equations is that by varying the density of the datasets we can create synthetic domains such that the number of solutions can be controlled. In case of acyclic graphs, for instance, the value of density that produces problems with one expected solution is $d = 1 / 4 \cdot \sqrt[n-1]{N}$, while for cliques this value is $d = 1 / \sqrt[n-1]{N \cdot n^2}$.

The following experiments were executed by Pentium III PCs at 500 MHz with 512MB Ram. For each experimental result we measure the average of 100 executions for each query (since the heuristics are non-deterministic the same query/data combination usually gives different results in different executions). In order to have a uniform treatment of similarity, independent of the number of the constraints, similarity is computed as 1-(#violated constraints/#total constraints).

The first experiment measures the quality of the solutions retrieved by the algorithms as a function of the number of query variables. In particular we constructed uniform datasets of 100,000 objects and executed acyclic and clique queries involving 5, 10, 15, 20 and 25 variables[2]. Depending on the number of variables/datasets involved and the query type, we adjusted the density of the datasets so that the expected number of solutions is 1. The time of every execution is proportional to the query size and set to $10 \cdot n$ seconds. Figure 10a illustrates the similarity of the best solution retrieved by the algorithms as a function of n, for chain and clique queries (average of 100 executions). The numbers in italics (top row) show the corresponding density values.

The second experiment studies the quality of the solutions retrieved over time. Since all algorithms start with random solutions which probably have very low similarities, during the initial steps of their execution there is significant improvement. As time passes the algorithms reach a *convergence point* where further improvement is very slow because a good solution has already been found and it is difficult for the algorithms to locate a better one. In order to measure how quickly this point is reached, we used the data sets produced for the 15-variable case and allowed the algorithms to run for 40 (chains) and 120 (cliques) seconds. Since chain queries are underconstrained, it is easier for the algorithms to quickly find good solutions. On the other hand, the large number of constraints in cliques necessitates more processing time. Figure 10b illustrates the (average) best similarity retrieved by each algorithm as a function of time.

The third experiment studies the behavior of algorithms as a function of the expected number of solutions. In particular, we use datasets of 15 variables and gradually increase the density so that the expected number of solutions grows from 1, to 10, 100 and so on until 10^5. Each algorithm is executed for 150 seconds (i.e., $10 \cdot n$). Figure 10c shows the best similarity as a function of *Sol*.

[2] We used synthetic datasets because, to the best of our knowledge, there do not exist 5 or more real datasets covering the same area publicly available. The query types were chosen so that they represent two extreme cases of constrainedness: acyclic queries are the most underconstrained, while cliques the most over-constrained.

The ubiquitous winner of the experiments is *SEA* which significantly outperforms *ILS* and *GILS* in most cases. *The solutions retrieved by the algorithm are often perfect matches*. This is very important since as we will see shortly, systematic search for exact solutions may require several hours for some problem instances. According to Figure 10c the performance gap does not decrease considerably as the number of solutions increases, meaning that the structure of the search space does not have a serious effect on the relative effectiveness of the algorithms.

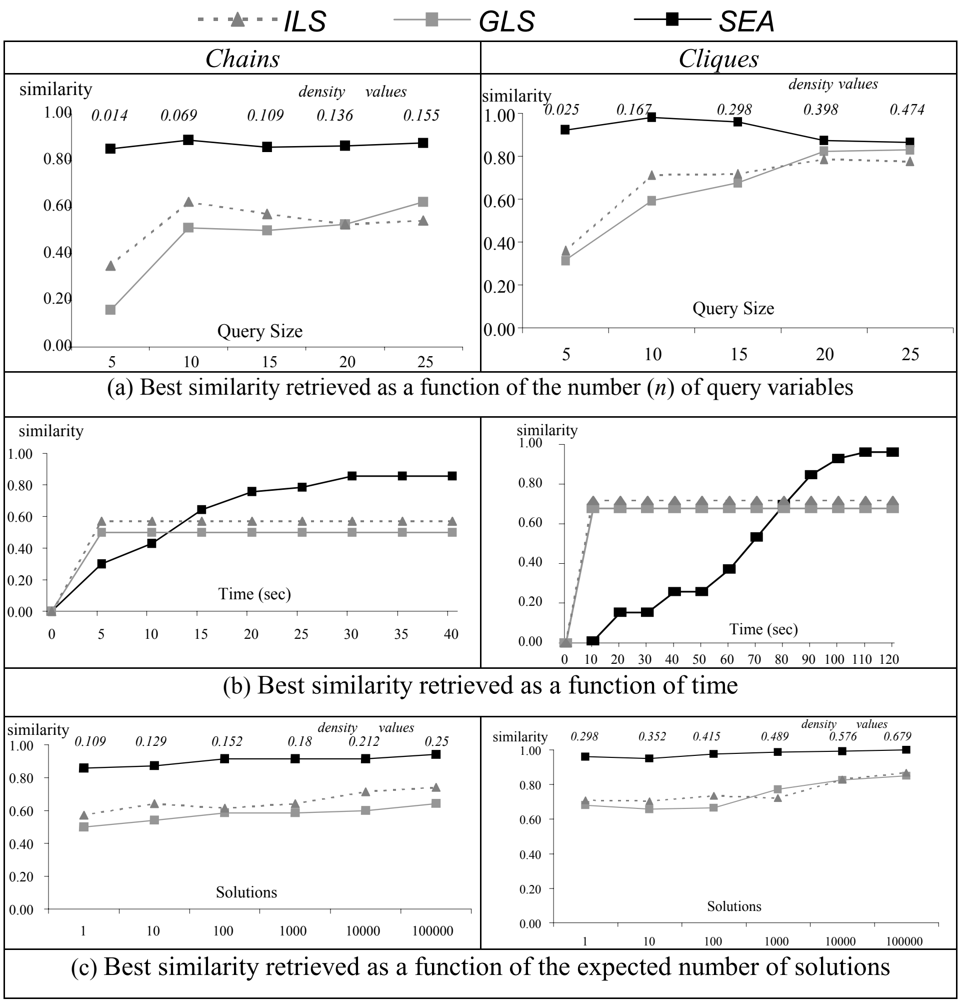


Fig. 10. Comparison of algorithms

The poor performance of *ILS* (and *GILS*) is rather surprising considering that local search significantly outperformed a genetic algorithm in the experimental evaluation of [PMK+99] for configuration similarity. This can be partially caused by the different images sizes (on the average, about an order of magnitude smaller than the datasets used in our experiments), version of the problem (soft spatial constraints that can be partially violated), implementation and parameter choices. The main reason, however, is that the current approach has some substantial improvements that affect relative performance: (i) we use indexes to re-assign the worst variable with the best value in its domain, while in [PMK+99] variables were re-assigned with random values, and (ii) we apply a sophisticated crossover mechanism that takes into account the quality of assignments in order to split solutions, while the genetic algorithm of [PMK+99] involves a random crossover mechanism. The first improvement enhances the performance of both local and evolutionary search, since indexes are used by *SEA*, for mutation, and by *ILS* (and *GILS*) for variable re-instantiation. Therefore, the main difference in relative efficiency is generated by the crossover mechanism. The careful swapping of assignments between solutions produces some better solutions which in subsequent generations will multiply through offspring allocation and mutate to better solutions.

ILS and *GILS* are still useful in cases where there is very limited time for processing since, as shown in Figure 10b, they reach their convergence point before 5 and 10 seconds for chains and cliques respectively (for chains, within 5 seconds *ILS* visits about 60,000 local maxima!). Although *SEA* will eventually outperform them, it requires longer time to reach high quality solutions due to the application of the genetic operations on a large population of solutions. Especially in the case of cliques, the crossover mechanism is not as efficient as for chains, because the constraints between all pairs of variables are very likely to generate large numbers of inconsistencies during the early steps where solutions have low similarities. *ILS,* in general, performs better than *GILS,* except for queries involving 20 and 25 variables. For large queries the similarity difference between a local maximum and its best neighbors is relatively small (due to the large number of constraints, each violation contributes little to the inconsistency degree of the solution), and good solutions are often found in the same neighborhood. So while *ILS* will retrieve one of them and then restart from a completely different point, the punishment process of *GILS* leads to a more systematic search by achieving gradual transitions between maxima and their neighbors.

In addition to their usefulness as standalone retrieval techniques, the above heuristics can be applied as a preliminary step to speed up systematic algorithms. In order to demonstrate this, we implemented a variation of the *WR* technique [PMT99], that finds the best approximate solutions, if no exact solutions exist. The resulting algorithm, *Indexed Branch and Bound* (*IBB*), instantiates variables by applying window queries in the corresponding tree. If there does not exist an object satisfying all join conditions with respect to already instantiated variables, the algorithm does not backtrack, but continues searching if the existing partial solution can potentially lead to a higher similarity than the best solution found so far. As in the case of *find best value* (Figure 5), objects that satisfy the largest number of join conditions are tried first (for details see [PA]).

IBB, and similar systematic search algorithms, can quickly discover the best solutions, if they have some "target" similarity to prune the search space. Otherwise, they may initially spend a lot of time exhaustively exploring areas of the search space with low quality solutions. A good value for this target similarity is very difficult to estimate because it depends on the dataset and the query characteristics. The proposed heuristics can be applied as a pre-processing step to provide such a value. In the next experiment we test the effectiveness of methods that first apply some search heuristic (*ILS* and *SEA*) to find a solution with high similarity, which is then input to *IBB*.

Figure 11 compares these two-step methods with the direct application of *IBB*. In particular we use "clique" queries over the datasets presented in Figure 10a. The datasets are such that the actual number of exact solutions is 1, and we measure the time (in seconds) that it takes for each method to retrieve the exact solution. The results are averaged over 10 executions because the target similarity returned by the heuristics differs each time due to their non-deterministic nature. Notice that often, especially for small queries, the exact solution is found by the non-systematic heuristics (usually *SEA*) in which case systematic search is not performed at all. The threshold for *SEA* is again $10{\cdot}n$ (i.e., 150) seconds which are sufficient for its convergence (see Figure 10b), while *ILS* is executed for 1 second (during this time it visits about 12,000 maxima and returns the best).

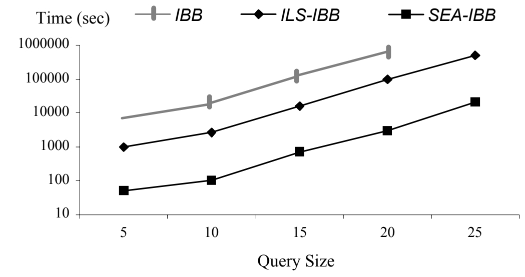


Fig. 11. Combinations of systematic with heuristic search

An important observation about Figure 11, refers to the processing time required for systematic search. *IBB* needs more than 100 minutes to process the smallest query (5 datasets). Cliques involving 25 variables take several days to terminate! This motivates the need for efficient retrieval of sub-optimal solutions through heuristic search. But, even when the optimal solutions are required, the incorporation of *SEA* speeds up *IBB* 1-2 orders of magnitude with respect to simple systematic search. This improvement is not so significant for *ILS* due to the lower similarity of the solutions retrieved.

7. Discussion

In this paper we propose several heuristics for multiway spatial join processing when the goal is retrieval of the most similar solutions within limited time. The best algorithm, *SEA*, can usually find optimal solutions even for difficult problems. In addition, we integrate systematic and non systematic search in a two-step processing method that boosts performance up to two orders of magnitude. To the best of our knowledge, our techniques are the only ones applicable for inexact retrieval of very large problems without any restrictions on the type of datasets, query topologies, output similarities etc. As such, they are useful in a variety of domains involving multiway spatial join processing and spatial similarity retrieval, including VLSI design, GIS and satellite imagery etc. Another potential application is the WWW, where the ever increasing availability of multimedia information will also require efficient mechanisms for multi-dimensional information retrieval. The methods are easily extensible to other spatial predicates, such as *northeast*, *inside*, near etc. Furthermore, they can be applied for cases where the image contains several types of objects and the query asks for configurations of objects within the same image (i.e., self-joins).

Regarding future directions, first we believe that the excellent performance of *SEA*, could be further improved in many aspects. An idea is to apply variable parameter values depending on the time available for query processing. For instance, the number of solutions *p* in the initial population may be reduced for very-limited-time cases, in order achieve fast convergence of the algorithm within the limit. Other non-systematic heuristics can also be developed. Given that using the indexes, local search can find local maxima extremely fast, we expect its efficiency to increase by including appropriate deterministic mechanisms that lead search to areas with high similarity solutions. Furthermore, several heuristics could be combined; for instance instead of generating the initial population of *SEA* randomly, we could apply *ILS* and use the first *p* local maxima visited as the *p* solutions of the first generation. Although we have not experimented with this approach, we expect it to augment the quality of the solutions and reduce the convergence time. For systematic search, we believe that the focus should be on two-step methods, like *SEA-IBB*, that utilize sub-optimal solutions to guide search for optimal ones. Finally another interesting direction is the application of our techniques to other (e.g., direction, distance) spatial predicates [ZSI01], as well as the incorporation of optimization heuristics similar to the ones that have been proposed for exhaustive processing of multiway spatial joins [PLC00].

Acknowledgments

This work was supported by grants HKUST 6081/01E and HKUST 6070/00E from Hong Kong RGC.

References

[BKS93] Brinkhoff, T., Kriegel, H., Seeger B. Efficient Processing of Spatial Joins Using R-trees. *ACM SIGMOD*, 1993.

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

[BT96] Blickle, T., Thiele, L. *A Comparison of Selection Schemes used in Genetic Algorithms*. TIK-Report No. 11, ETH, Zurich, 1996.

[CA93] Crawford, J., Auton, L. Experimental Results on the Crossover Point in Satisfiability Problems. *AAAI*, 1993.

[CFG+98] Clark, D., Frank, J., Gent, I., MacIntyre, E., Tomov, N., Walsh, T. Local Search and the Number of Solutions. *Constraint Programming*, 1998.

[CSY87] Chang, S, Shi, Q., Yan C. Iconic Indexing by 2-D String. *IEEE PAMI* 9(3), 413-428, 1987.

[DM94] Dechter R., Meiri I. Experimental Evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68: 211-241, 1994.

[DTW+94] Davenport, A., Tsang, E., Wang, C., Zhu, K. GENET: A Connectionist Architecture for Solving Constraint Satisfaction Problems by Iterative Improvement. *AAAI*, 1994.

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

[G86] Grefenstette, J. Optimization of Control Parameters for Genetic Algorithms. *IEEE Trans. on Systems, Man and Cybernetics*, 16 (1), 1986.

[G89] Goldberg, D. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, Mass., 1989.

[GL97] Glover F., Laguna, M. *Tabu Search*. Kluwer, London, 1997.

[H75] Holland, J. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

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

[LYC92] Lee, S, Yang, M, Chen, J. Signature File as a Spatial Filter for Iconic Image Database. *Journal of Visual Languages and Computing*, 3, 373-397, 1992.

[MJP+92] Minton, S. Johnston, M., Philips, A., Laird P. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence* 58(1-3), 161-205, 1992.

[MP99] Mamoulis, N, Papadias, D., Integration of Spatial Join Algorithms for Processing Multiple Inputs. *ACM SIGMOD*, 1999.

[PA] Papadias, D., Arkoumanis D. Search Algorithms for Multiway Spatial Joins. To appear in the *International Journal of Geographic Information Science (IJGIS)*. Available at: http://www.cs.ust.hk/~dimitris/

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

[PLC00] Park, H-H., Lee, J-Y., Chung, C-W. Spatial Query Optimization Utilizing Early Separated Filter and Refinement Strategy. *Information Systems* 25(1): 1-22, 2000.

[PMK+99] Papadias, D., Mantzourogiannis, M., Kalnis, P., Mamoulis, N., Ahmad, I. Content-Based Retrieval Using Heuristic Search. *ACM SIGIR*, 1999.

[PMT99] Papadias, D., Mamoulis, N., Theodoridis, Y. Processing and Optimization of Multiway Spatial Joins Using R-trees. *ACM PODS*, 1999.

[TSS98] Theodoridis, Y., Stefanakis, E., Sellis, T., Cost Models for Join Queries in Spatial Databases, *ICDE*, 1998.

[ZSI01] Zhu, H, Su, J, Ibarra, O. On Multi-way Spatial Joins with Direction Predicates. *SSTD*, 2001.