Search Algorithms for Multiway Spatial Joins

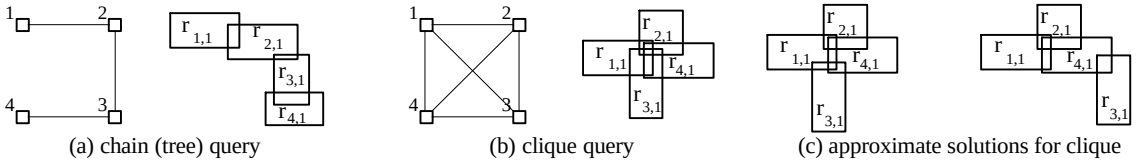
DIMITRIS PAPADIAS
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
Email: dimitris@cs.ust.hk

DINOS ARKOUMANIS
Department of Electrical and Computer Engineering
National Technical University of Athens
Greece, 15773
Email: dinosar@dbnet.ece.ntua.gr

Abstract. This papers deals with multiway spatial joins when (i) there is limited time for query processing and the goal is to retrieve the best possible solutions within this limit (ii) there is unlimited time and the goal is to retrieve a single exact solution, if such a solution exists, or the best approximate one otherwise. The first case is motivated by the high cost of join processing in real-time systems involving large amounts of multimedia data, while the second one is motivated by applications that require "negative" examples. We propose several search algorithms for query processing under theses conditions. For the limited-time case we develop some non-deterministic search heuristics that can quickly retrieve good solutions. However, these heuristics are not guaranteed to find the best solutions, even without a time limit. Therefore, for the unlimited-time case we describe systematic search algorithms tailored specifically for the efficient retrieval of a single solution. Both types of algorithms are integrated with R-trees in order to prune the search space. Our proposal is evaluated with extensive experimental comparison.

1. Introduction

A multiway spatial join can be expressed as follows: Given n datasets D_1, D_2, ... 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 $\{(r_{1,w},..,r_{i,x},..,r_{j,y},..,r_{n,z}) \mid \forall\ i,j : r_{i,x} \in D_i,\ r_{j,y} \in D_j$ 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* (Dechter and Meiri 1994) 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 standard terminology in the spatial database literature we assume that the standard join condition is *overlap* (*intersect*, *non-disjoint*). In this case the graph is undirected (Q_{ij}=Q_{ji}) and, if Q_{ij}=True, then the rectangles from the corresponding inputs i,j should overlap. Figures 1a and 1b illustrate two example queries: the first one has an acyclic (*tree*) graph, and the second one has a complete (*clique*) graph.



(a) chain (tree) query (b) clique query (c) approximate solutions for clique

Figure 1 Example queries and solutions

We use the notation $v_i \neg r_{i,x}$ to express that variable v_i is instantiated to rectangle $r_{i,x}$ (which belongs to domain D_i). A *binary instantiation* $\{v_i \neg r_{i,x},\ v_j \neg r_{j,y}\}$ 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 $\{v_1 \neg\ r_{1,w}, .., v_i \neg r_{i,x}, .., v_j \neg r_{j,y}, .., v_n \neg r_{n,z}\}$ 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*. Figures 1a and 1b illustrate an exact solution for each type of query. Figure 1c shows two approximate solutions for the clique. The left one violates only one condition (Q_{23}), while the right one violates two (Q_{13} and Q_{23}). The lower the inconsistency degree, the higher the *similarity* of the solution.

The goals of multiway spatial join processing are determined by the three parameters illustrated in figure 2. The x-axis refers to the type of solutions to be retrieved, the y-axis to the number of solutions and the z-axis specifies the time available for query processing. The three numbers in the diagram correspond to cases of particular interest:

1) The goal is the retrieval of *all exact solutions* with *no time limit* for query processing. Existing work on multiway spatial joins (discussed in the next section) deals mostly with this case. Depending on the query and data properties, however, exhaustive processing of multiway spatial joins can be prohibitively expensive. Furthermore, if there do not exist any exact solutions, the result will be empty. In architectural applications, for instance, queries often involve numerous variables and the database may not contain configurations that match all the query constraints. The best approximate solutions are then desirable.

2) The shortcomings of the first case motivate the second one. The goal is the retrieval of the *best* possible (exact or approximate) *solutions within a time threshold*. Fast retrieval of sub-optimal solutions is the only way to deal with the vast (and ever increasing) amounts of multimedia information for several real time systems. Related work has been carried out in the context of spatial similarity retrieval where queries can be modeled as multiway self-joins.

3) The goal in the third case, is the retrieval of the *best* (*single*) *solution with no time limit.* Some applications require the retrieval of negative examples. For instance, a possible VLSI design can be invalidated if it matches an existing faulty one. In some engineering and industrial CAD applications several object configurations are prohibited. If a proposed plan is topologically similar to an illegal one, it should be altered or rejected. Retrieval of all similar configurations is not needed, since a single match suffices.

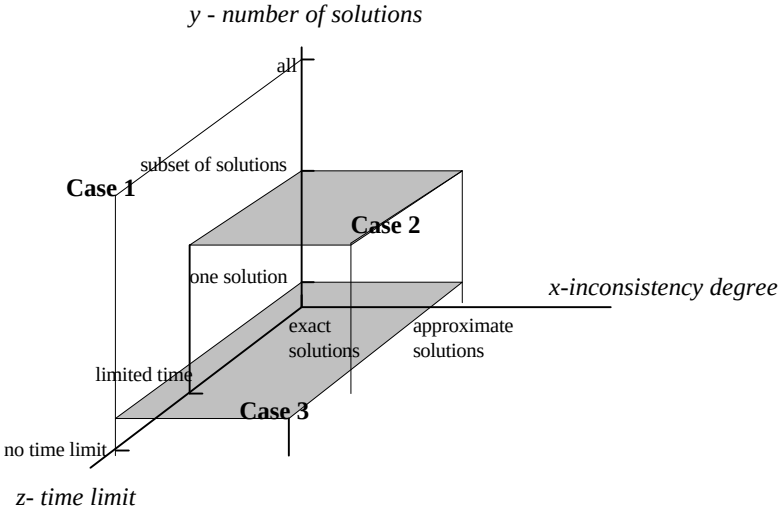


Figure 2: Goals of multiway spatial join processing

This paper deals with multiway spatial join processing involving the last two cases. For the second case we propose some non-deterministic search heuristics that can quickly retrieve good solutions. However, these algorithms are not guaranteed to find the best solutions, even without a time limit. Therefore, for the third case we propose systematic search algorithms tailored specifically for the efficient retrieval of a single exact solution (if such a solution exists), or the retrieval of the best approximate solution. For this case we also present a two-step method, which combines systematic and heuristic search, and may reduce processing time by orders of magnitude.

The rest of the paper is structured as follows: section 2 overviews related work and section 3 proposes search heuristics (case 2) based on the concepts of local search, guided local search and evolutionary algorithms. Section 4 discusses systematic search algorithms and their integration with R-trees for multiway spatial join processing. Section 5 contains a comprehensive experimental evaluation using several data sets and query combinations. Section 6 concludes the paper with a discussion and future work.

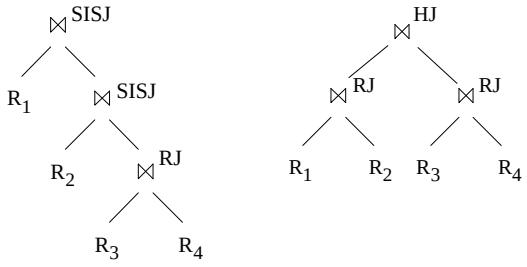
2. Related Work

Previous work on *pairwise*-join processing, can be classified in two categories. The first one includes algorithms applicable when both relations to be joined are indexed on the spatial attributes. The most influential technique in this category is *R-tree-based Join* (*RJ*) (Brinkhoff et al. 1993), which presupposes the existence of R-trees for both relations. *RJ* is based on the *enclosure property*: if two intermediate R-tree nodes do not intersect, there can be no objects below them that intersect. The algorithm starts from the roots of the trees to be joined and for each pair of overlapping entries inside them, it is recursively called until the leaf levels where overlapping entries constitute solutions. Huang et al. (1997), extend *RJ* by introducing an on-the-fly indexing mechanism to optimize, in terms of I/O cost, the execution order of matching at intermediate levels. Theodoridis et al. (2000) present formulae for cost estimation in terms of node accesses.

The methods of the second category treat non-indexed inputs (e.g., when there is another operation, such as selection, before the spatial join). If there is an R-tree for only one input, processing can be done by (i) indexed nested loop, (ii) building a second R-tree for the non-indexed input using bulk loading and then applying *RJ*, (iii) the *sort and match* algorithm (Papadopoulos et al. 1999) (iv) the *seeded tree* algorithm (Lo and Ravinshakar 1994) which works like (ii) but builds the second R-tree using the existing one as a skeleton (seed) (v) the *slot index spatial join* (*SISJ*) (Mamoulis and Papadias 1999) which is an improved version of (iv). If both inputs are non-indexed, some methods (Patel and DeWitt 1996, Koudas and Sevcik 1997) partition the space into cells (a grid-like structure) and distribute the data objects in buckets defined by the cells. The spatial join is then performed in a relational hash join fashion. A similar idea is applied by the spatial hash join algorithm (*HJ*) but, instead of regular space partitioning, the buckets are determined by the distribution of objects in the probe input (Lo and Ravinshakar 1994). Another method (Arge et al. 1998) first applies external sorting to both files and then uses an adaptable plane sweep algorithm, considering that in most cases the “horizon” of the sweep line will fit in main memory.

As in the case of relational joins, multiway spatial joins can be processed by combining pairwise join algorithms. The *pairwise join method* (*PJM*) (Mamoulis and Papadias 1999) considers a join order that is expected to result in the minimum cost (in terms of page accesses). Each join order corresponds to exactly one execution plan where: (i) *RJ* is applied when the inputs are leaves i.e., datasets indexed by R-trees, (ii) *SISJ* is employed when only one input is indexed by an R-tree and (iii) *HJ* when both inputs are intermediate results. As an example of *PJM*,

consider the query in figure 1a and the plans of figure 3. Figure 3a involves the execution of *RJ* for determining $R_3 \bowtie R_4$. The intermediate result, which is not indexed, is joined with R_2 and finally with R_1 using *SISJ*. On the other hand, the plan of figure 3b applies *RJ* for $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$, and *HJ* to join the intermediate results. Unfortunately, *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. Therefore, *PJM* cannot be applied for retrieval of cases 2 and 3.



(a) right-deep plan (b) bushy plan

Figure 3: Alternative plans using pairwise join algorithms

Two alternative methodologies for multiway spatial joins, motivated by algorithms for constraint satisfaction problems (CSPs), were proposed in (Papadias et al. 2001). Synchronous traversal (*ST*) can be thought of as the generalization of *RJ* to arbitrary query graphs, i.e., starting from the roots of the R-trees, *ST* 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 $v_1 \neg r_{1,w}$, 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., $v_2 \neg r_{2,w}$. The algorithm proceeds forward using the values of instantiated variables as query windows. Assuming for instance, that all variables v_1 to v_{i-1} have been instantiated, the consistent values for v_i, must overlap the assignments of all v_j, j=1..i-1 such that Q_{ij}=True. If no such rectangle can be found, the algorithm backtracks to the previous variable and tries another value for it.

ST and *WR* have been applied for approximate retrieval of multiway spatial joins involving various spatial predicates (Papadias et al. 1998); *WR* was found superior because *ST* induces numerous false hits at the high tree levels when the query involves relaxed constraints. Nevertheless, methods, like *ST* or *WR*, that search systematically in the solution space are not suitable for retrieval under limited time (case 2) because they may initially spend a lot of time in regions with low quality solutions, failing to quickly find some good ones. Furthermore, the algorithms of (Papadias et al. 1998) are restrictive for general retrieval in case 3 because they assume soft constraints (which can be partially violated) and that the user specifies a target similarity, which should be exceeded by the solutions to be retrieved. If the target similarity is too high no solutions will be found, while if it is too low the algorithms will retrieve too many useless solutions (unnecessarily deteriorating performance). In section 4 we propose alternative techniques, based on the main concept of *WR*, which find the best solution without assuming a target similarity.

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. Petrakis and Faloutsos (1997) 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 similarity 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 (Lee and Hsu 1992, Lee et al. 1992), 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 (Chang et al. 1987). 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.

Papadias et al. (1999) deal with approximate retrieval of similarity queries under limited time (i.e., case 2) by using search heuristics based on local search, simulated annealing and genetic algorithms. Their evaluation 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 multiway spatial joins. However, unlike (Papadias et al. 1999) where the presented algorithms were a straightforward adaptation of local and evolutionary search for similarity retrieval, the following 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 (Beckmann et al 1990) on minimum bounding rectangles (MBRs) and deal with the filter step of intersection joins. The proposed techniques are easily extensible to other spatial predicates, such as north, inside, meet etc.

3. Non-Systematic Search Heuristics

Retrieval of the best (exact or approximate) solutions of multiway spatial joins is essentially an optimization problem. Since such problems are in general exponential in nature, several search heuristics have been proposed for the efficient retrieval of sub-optimal solutions. These heuristics are non-systematic in the sense that their search method includes some randomness. Although they are not guaranteed to find the best solution, usually they perform well in limited time. Their performance depends on the application and the special characteristics of the problem. In our case, the existence of spatial indexes facilitates the processing of much larger problem sizes than usual. The integration of search heuristics with spatial indexes, however, is not trivial and requires some careful design in order to take full advantage of the indexes. In this chapter we propose three such heuristics: indexed local search, guided indexed local search and a spatial evolutionary algorithm.

3.1 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 domains 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(*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 high 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 4.

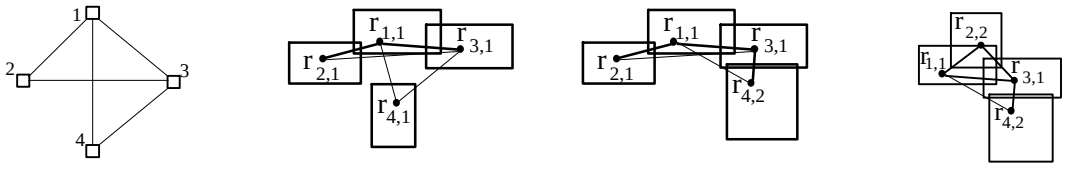
Indexed Local Search

```
1     WHILE NOT (Time limit) {
2       S := random seed
3       WHILE NOT(Local_Maximum) {
4           determine worst variable v_i
5           value := find best value (Root of tree R_i, v_i)
6           IF better value THEN
7                                   S = S ∧ { v_i ← Value }
8                                   IF S among the best solutions found so far THEN keep S
9       } /* END WHILE NOT Local_Maximum */
10    } /* END WHILE NOT Time Limit */
```

Figure 4: Indexed local search

Motivated by conflict minimization algorithms (Minton et al. 1992), we choose (line 4) 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 5a and the approximate solution of figure 5b. 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 5b 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.



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

Figure 5: Example of ILS

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 5b, 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 $r_{3,1}$. The pseudo-code for *find best value* is illustrated in figure 6; 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 $w_{3=}r_{3,1}$).

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.

maxConditions=0
bestValue=∅
Find best value (Node N, integer i)
1. FOR EACH entry e_x of N
2. FOR EACH Q_{ij} such that Q_{ij} = True
3. IF e_x intersects w_j THEN conditions$_x$=conditions$_x$+1
4. Sort all entries e_x such that conditions$_x$>0 with respect to conditions$_x$
5. IF N intermediate node
6. FOR each $e_x \in$ E
7. IF conditions$_x$ > maxConditions THEN **Find best value** (e_x, i)
8. ELSE *//leaf node*
9. FOR each $e_x \in$ E
10. IF conditions$_x$ > maxConditions THEN
11. maxConditions=conditions$_x$
12. bestValue=e_x

Figure 6: *Find best value*

Figure 7 illustrates this process for the example of figures 5a and 5b. *Find best value* will retrieve the rectangle 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 *maxConditions* 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 5c) is 2 ($Q_{3,4}$ is now satisfied). At the next step (figure 5d), 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 7, there is no object in the fourth domain that intersects both $r_{1,1}$ and $r_{3,1}$.

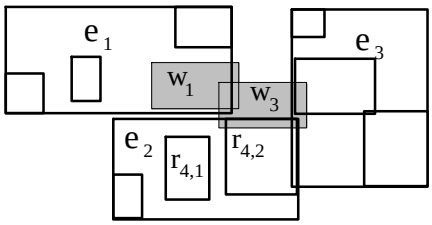


Figure 7: Example of *find best value*

3.2 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 (Glover and Laguna 1997). 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 (Davenport et al. 1994) 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, sometimes, unrelated nodes that share features with visited local maxima, are avoided too and some 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 $v_1 \neg r_{1,w}$, .., $v_i \neg r_{i,x}$, .., $v_j \neg r_{j,y}$, .., $v_n \neg r_{n,z}$ get a penalty. In particular, *GILS* penalizes the assignments with the minimum penalties so far; e.g., if $v_1 \neg r_{1,w}$ already has some punishment from a previous local maximum (while the others do not), only the rest of the assignments are penalized in order to avoid over-punishing $v_1 \neg r_{1,w}$. The code for *GILS* (figure 8) 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 (line 1, outside the while loops) and has some additional code (lines 10-13) for penalty assignment.

Guided Indexed Local Search

```
1  S := random seed
2  WHILE NOT (Time limit) {
3      WHILE NOT(Local_Maximum) {
4          determine worst variable v_i
5          value := find best value (Root of tree R_i, v_i)
6          IF better value THEN
7                      S = S ∧ { v_i ← Value }
8                      IF S among the best solutions found so far THEN keep S
9              } /* END WHILE NOT Local_Maximum */
10     P = among the assignments of the current local maximum, select the ones with the minimum penalty
11     FOR EACH assignment v_i ¬ r_{i,x} in P
12         penalty(v_i ¬ r_{i,x})= penalty(v_i ¬ r_{i,x}) + 1
13} /* END WHILE NOT Time limit */
```

Figure 8: Guided indexed local search

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

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

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.

The results of this punishment process are:

(i) 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.

(ii) 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 (nxN) array where the cell (i,j) stores the penalty of assigning the i^{th} variable with the j^{th} value in its domain ($v_i \neg r_{i,j}$). 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, we build an in-memory hash table which only stores the assignments with positive penalties.

3.3 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 (Blickle and Thiele 1996) 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.

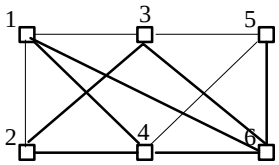
Crossover mechanism is the driving force of *exploration* in evolutionary algorithms. In the simplest approach (Holland 1975), pairs of solutions are selected randomly from the population. For each pair a *crossover point* is defined randomly, and the solutions beyond it are mutually exchanged with probability 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*, Goldberg 1989). 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 another solution.

Figure 9 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

Figure 9: Example of solution splitting during crossover

Notice that there exist several alternative options for crossover. For instance, instead of choosing the second solution at random, we could find the one that has the maximum number of satisfied conditions among v_2,v_3 and v_5. Going one step further we could perform exhaustive search in order to find the optimal way to combine solutions. However, such methods introduce significant computational overhead and should be avoided. The elementary operations, including crossover, must be as fast as possible in order to allow the initial population to evolve through a large number of generations.

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 10 illustrates the pseudo-code for *SEA*. 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

```
1   P := generate initial set of solutions {S_1,..,S_p}
2   WHILE NOT (Time limit) {
3       compute crossover point c /* increase the value of c by 1 every g_c generations */
4       FOR EACH S_i in P /*evaluation */
5           evaluate S_i
6           IF S_i among the best solutions found so far THEN keep S_i
7       FOR EACH S_i in P /* offspring allocation */
8           compare S_i with T other random solutions in P
9           replace S_i with the best among the T+1 solutions
10      FOR EACH S_i in P /*crossover*/
11          with probability μ_c change S_i as follows
12              determine set of c variables to keep their current values
13              re-instantiate the remaining variables using their values in S_j (S_j ∈ P and i ≠ j)
14      FOR EACH S_i in P /* mutation */
15          with probability μ_m change S_i as follows
16              determine worst variable v_k
17              value := find best value (Root of tree R_k, v_k)
18  }/* END WHILE NOT Time limit */
```

Figure 10: 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 interrelated 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 (Grefenstette 1986). In the next section we explore parameter tuning with respect to the problem characteristics.

3.4 Parameter Tuning

In order to provide a general set of parameters, applicable to a variety of problem instances, we perform the tuning based on the size of the problem. The size s of a problem denotes the number of bits required to represent the search space (Clark et al. 1998), i.e., the number of bits needed to express all possible solutions:

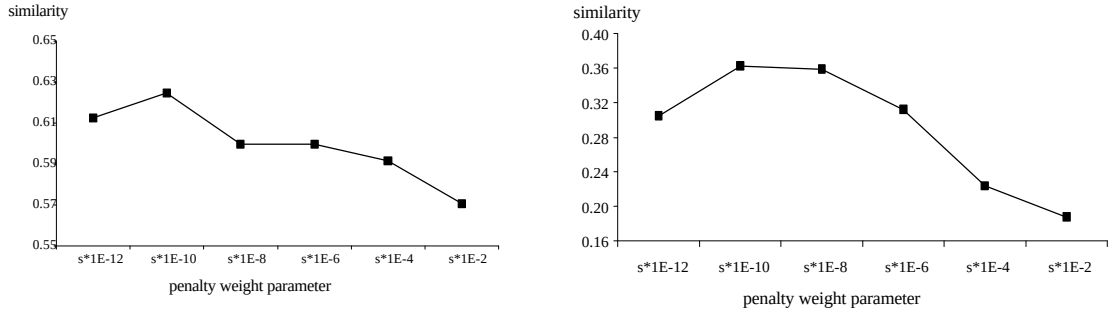
$$s = \log_2 \cdot \prod_{i=1}^{n} N_i$$

For all experiments on parameter tuning we used two queries; a chain applied on 25 uniform datasets with density d=0.155 and a clique applied on 5 uniform datasets with density d=0.025. All datasets contain 100,000 objects and the above density values are such that the resulting problems have one exact solution (the motivation is discussed in section 5). These settings were chosen so that they represent two extreme cases of constrainedness and query size. Acyclic queries are the most under-constrained, while cliques the most over-constrained. As shown in the sequel, the best parameter values are the same or very close for the two queries, implying that these values provide good performance for most other queries as well.

The following experimental results represent 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). The time of every execution is proportional to the query size and set to $10 \cdot n$ seconds. 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).

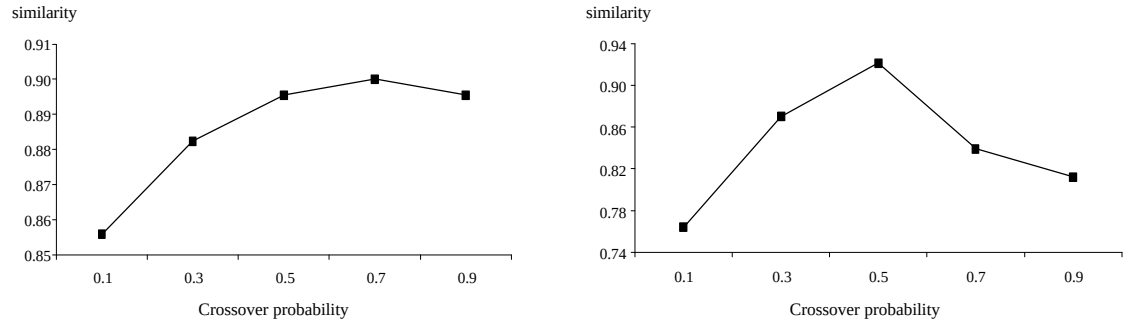
The first experiment aims at adjusting the penalty weight λ of *GILS*. We used values of λ in the range$10^{-12}\cdot s$, $10^{-10}\cdot s$, $10^{-8}\cdot s$ and so on until $10^{-2}\cdot s$. Figure 11 displays the average (over 100 executions) best similarity found for each value of λ. In both query types the best performance is achieved for $\lambda=10^{-10}\cdot s$. Smaller values of λ will cause the algorithm to spend most of the available time in the neighborhood of the first local maximum visited. On the other hand, large values will prevent exploration of the neighborhoods around local maxima missing some good solutions. The value $10^{-10}\cdot s$ provides a good trade-off between these two cases.



(a) Chain query with 25 variables (b) Clique query with 5 variables

Figure 11: Penalty weight parameter tuning for *GILS*

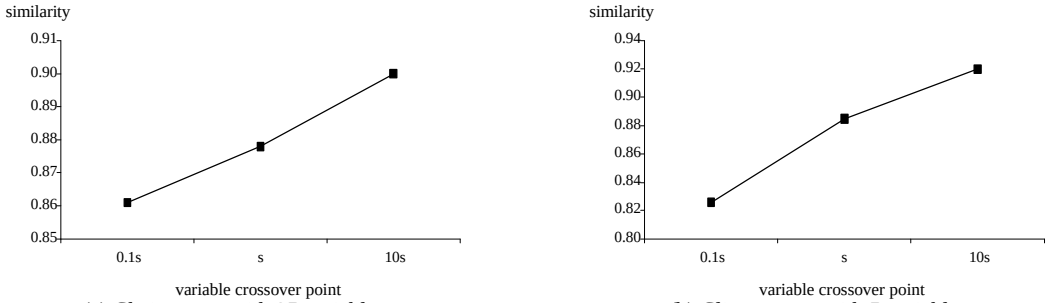
The rest of the experiments involve the *SEA* algorithm. Due to the large number of parameters and their interrelationships, we performed the experiments in two rounds. Through the first round (omitted here) we determined a starting point for the parameter values as follows: population size $p=100\cdot s$, number of solutions in tournament $T=0.05\cdot s$, crossover rate $\mu_c= 0.5$, $g_c=10\cdot s$ and mutation rate $\mu_m=0.8$. Then a second round of experiments was conducted to fine-tune each parameter individually by keeping the values of the other ones fixed. The first experiment (of the second round) attempts to find the best value for the crossover rate by trying values from 0.1 to 0.9. Figure 12 shows the average best similarity as a function of μ_c. The highest similarity is achieved for values between 0.5 and 0.7 and for the following experiments we set $\mu_c= 0.6$.



(a) Chain query with 25 variables (b) Clique query with 5 variables

Figure 12: Crossover probability tuning for *SEA*

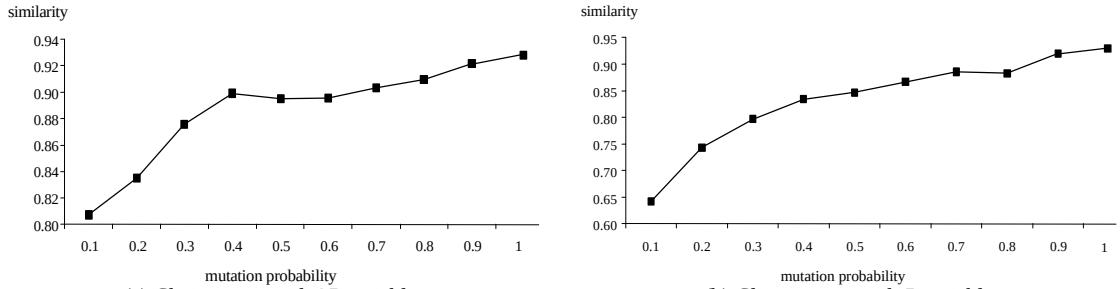
Given the value of crossover rate, we try to find the best value of g_c, i.e., the number of generations that the population must evolve before the variable crossover point is increased by one. We experimented with orders of magnitude of s: $0.1\cdot s$, s and $10\cdot s$. Larger values are pointless since the algorithm reaches the time limit before $100\cdot s$ generations. As shown in figure 13, the best results are obtained for $g_c=10\cdot s$, meaning that *SEA* works better when it only keeps the instantiations of a few, but very good variables. This implies that the crossover has a rather strong effect in the performance of the algorithm.



(a) Chain query with 25 variables (b) Clique query with 5 variables

Figure 13: Tuning of g_c for *SEA*

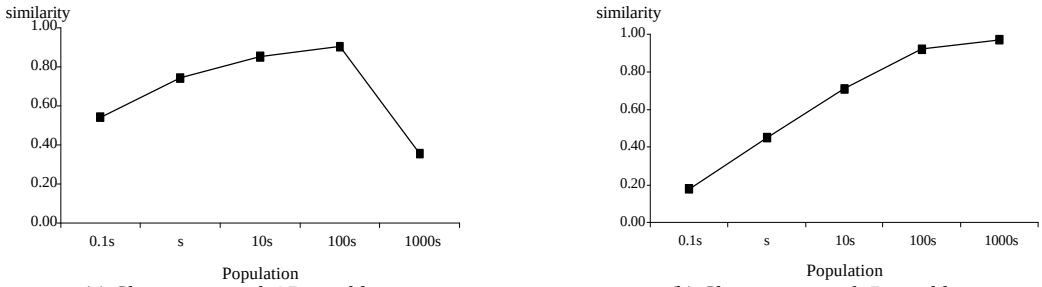
The following experiment involves the mutation rate. *SEA* was executed for values of μ_m in the range [0.1, 1], and according to figure 14 the best results were obtained for μ_m=1. This was expected since mutation uses the index to quickly re-instantiate the worst variable to the best value in its domain, significantly improving solutions. We also experimented with mutation of a random (but violating at least a constraint) variable. The motivation is, given that we may have multiple copies of the same solution in one generation (due to offspring allocation), it might pay off to re-instantiate different variables in order to create larger variety in the population. The results, however, have shown that mutation on the worst variable is significantly more effective. Crossover provides a better way for altering solutions and achieving variety than mutation on random variables.



(a) Chain query with 25 variables (b) Clique query with 5 variables

Figure 14: Mutation probability tuning for *SEA*

After having obtained the optimal values for the other parameters, in the last experiment we examine values 0.1·*s*, *s*, 10·*s*, 100·*s* and 1000·*s* as potential population sizes *p*. The larger the population, the higher the percentage of the search space covered by solutions, but the lower the number of generations permitted for evolution within a time limit. As shown in figure 15, the best value is *p*=100·*s* for chains and *p*=1000·*s* for cliques. This is because, due to the larger problem size in the case of chains, values larger than 100·*s* only permit an insufficient number of generations and evolution is not complete within the time limit. Since *p*=100·*s* provides good results for cliques as well, we choose this value.



(a) Chain query with 25 variables (b) Clique query with 5 variables

Figure 15: Population size tuning for *SEA*

We could continue the process of fine-tuning *SEA* using some additional rounds of experiments (possibly in higher granularity) in order to find even better parameter values. This, however, is beyond the scope of the paper, since our goal is mainly to measure the applicability of the algorithm and observe its relative performance with respect to the others. Summarizing, the following set of values will be used in the experimental comparison: $\lambda = 10^{-10} s$, $T=0.05 \cdot s$, $\mu_c = 0.6$, $g_c=10 \cdot s$, $\mu_m=1$, and $p=100 \cdot s$. The next section describes a different type of algorithms that, unlike the previous search heuristics, are deterministic (given a query and a set of inputs they always produce the same output following the same steps at each execution) and do not require any parameter tuning.

4. Systematic Search Algorithms

The two algorithms proposed in this section are aimed at the retrieval of the solution(s) with the highest similarity (without a time limit). Both search systematically in the solution space and can be classified under the window reduction methodology in the sense that they apply variations of backtracking and query windows to instantiate variables. The algorithms differ in the way that they instantiate variables: (i) the first one chooses the first good value in the domain, quickly moving forward to the next variable (ii) the second algorithm spends more time at the current variable, trying to instantiate it to the best value in its domain.

4.1 Indexed Look Ahead

Indexed look ahead (*ILA*) combines backtracking with the tree search process, in order to move forward and instantiate all variables with the minimum amount of computation. The outcome is a doubly recursive algorithm without any memory requirements. As shown is figure 16, *ILA* starts by initializing two variables: *minViolations* (initially set to ∞) is the total number of violations of the best solution found so far, and *violations[i]* is the current number of violations when the first *i* variables get instantiated (initially *violations[1]*=0). The algorithm then assigns to the first variable each value in its domain, and for every instantiation, it calls *go ahead* function which performs the search. The current assignment of v_1, becomes a query window w_1 for all subsequent variables v_j such that Q_{ij}=True.

Indexed Look Ahead

```
1   i := 1 /* pointer to the current variable */
2   violations[1] := 0; minViolations := ∞; bestSolution := ∅
3   FOR EACH object r1,x of the first domain D1
4           Assign v1 ¬ r1,x
5           Go Ahead (root of R2, 2)
```

Go Ahead (Node N, integer i)

```
1    FOR EACH entry ex of N
2        FOR EACH Qij such that Qij = True and 1≤j<i
3            IF ex does not intersect wj THEN violationsx=violationsx+1
4            IF violationsx + violations[i-1]≥ minViolations THEN GOTO to next entry
5        IF N is intermediate node THEN Go Ahead (ex, i)
6         ELSE // leaf node
7            Assign vi ¬ ex
8            violations[i] := violations[i-1] + violationsx
9            IF i < n THEN Go Ahead (root of Ri+1, i+1)
10             ELSE /* the last variable has been instantiated */
11               minViolations := violations[i]
12               bestSolution=current assignment
13               IF minViolations=0 THEN return current solution and stop
```

Figure 16: Indexed look ahead

Go ahead starts from the root of the next variable to be instantiated (initially the second one) and for each entry e_x checks whether it intersects the existing windows of instantiated variables. Every violation increases its $violations_x$ count by one. Entries producing a total number of violations equal or greater than the best solution so far, are rejected. The rest are followed recursively to the leafs. Once a qualifying leaf entry (i.e., actual object) is found, it gets immediately assigned to the current variable and the same process is repeated for the next variable. When the last variable gets instantiated, *minViolations* is updated.

In order to comprehend the functionality of *ILA*, consider the three domains of figure 17, and a clique query ($Q_{1,2}$=$Q_{1,3}$=$Q_{2,3}$=True). Each domain is indexed by an R*-tree with node capacity 3. Assume that entries in the same node are accessed in the order specified by their ids (e.g., $r_{1,1}$, $r_{1,2}$, $r_{1,3}$ or $N_{1,1}$, $N_{1,2}$, $N_{1,3}$). Variable v_1 is first instantiated to $r_{1,1}$, and calls *go ahead*(root of tree R_2, 2). Because at this point the value of *minViolations* is ∞, no entries can be pruned out since they can all lead to qualifying solutions. Thus *go ahead*($N_{2,1}$, 2) is called and the first object ($r_{2,1}$) inside $N_{2,1}$ becomes the assignment of v_2. Notice that since $r_{2,1}$ does not intersect $r_{1,1}$, the partial solution has one violation (violations[2]=1). Rectangles $r_{1,1}$ and $r_{2,1}$ become windows w_1 and w_2, respectively, for search in the third tree. Then, the calls *go ahead*(root of tree R_3, 3) and *go ahead*($N_{3,1}$, 3) produce the instantiation v_3 ¬ $r_{3,1}$. Since $r_{3,1}$ intersects both $r_{1,1}$ and $r_{2,1}$, *minViolations* =1. From now on, the algorithm will only search for exact solutions.

Then the other entries of $N_{3,1}$ are rejected since they both produce a larger number of violations than *minViolations*. *Go ahead*($N_{3,1}$, 3) terminates and control resumes to *go ahead*(root of tree R_3, 3). Entry $N_{3,2}$ is immediately rejected and the algorithm backtracks to *go ahead*($N_{2,1}$, 2). Because no entry in $N_{2,1}$ can generate an exact solution, *go ahead*($N_{2,1}$, 2) terminates and *go ahead*(root of tree R_2, 2) resumes. Since neither $N_{2,2}$, nor $N_{2,3}$, intersect $r_{1,1}$, the algorithm backtracks once more and re-instantiates the first variable to a new value ($r_{1,2}$).

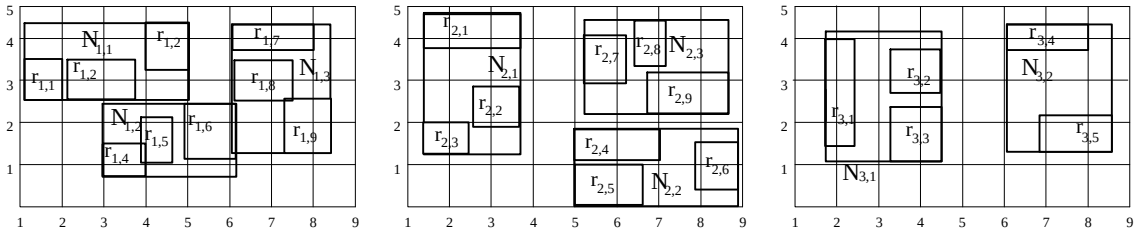


Figure 17: Example of *ILA*

The order in which variables get instantiated is very important for the efficiency of *ILA* (and backtracking algorithms in general). A good order can quickly guide to a solution with a small number of violations so that a large part of the entries can be pruned out early during search. For generating such an order we follow again a greedy approach: the variable that participates in the largest number of conditions is instantiated first. The i^{th} variable to be selected is the one, which participates in the largest number of conditions with respect to the already chosen i-1 variables. In case of a tie, the variable with the most join conditions is favored.

4.2 Indexed Branch-and-Bound

When instantiating a variable, *ILA* selects the first value in its domain that may possibly lead to solutions with similarity higher than the best solution found so far. Although this strategy allows the algorithm to quickly move forward, it may not be a good choice in the long term since it may cause unnecessary work. Consider, for instance, that the similarity of the best solution found so far is rather low, and there exist two potential instantiations for the current variable that can exceed this similarity: the first satisfies 5 constraints with respect to already instantiated

variables and the other satisfies 10. *ILA* will assign the first value (if it is found before the second one) although the second will most probably lead to a better solution.

Indexed Branch-and-Bound (*IBB*) on the other hand, will sort all qualifying values according to the number of conditions that they satisfy with respect to rectangles of instantiated variables, and then assign the one that satisfies the most conditions. Figure 18 illustrates the pseudo-code for *IBB*. Like *ILA*, the algorithm works in a backtracking-based function; their main difference is that, through the *similarity search* function, *IBB* computes the complete domain of the current variable v_i , i.e., the set of values that can lead to a solution better than the one found so far. Then, this set is sorted so that the best values are assigned first, leading quickly to good solutions (a similar idea is applied by *find best value* in non-systematic search). The trade-off is the space requirements for keeping the domains and the computational cost for sorting. The order of variables is determined using the same heuristic as *ILA*.

Indexed Branch and Bound

```
1   i := 1 /* pointer to the current variable */
2   violations[0] := 0; minViolations := ∞ ; bestSolution := ∅
3   domain[1] := All objects of the first domain D1
4   WHILE (TRUE) {
5       IF domain[i] = ∅ THEN GOTO 18 /* backtrack */
6       ex := get next value in domain[i]
7       Assign vi ¬ ex
8       violations[i] := violations[i-1] + violationsx
9       IF violations[i] < minViolations THEN
10          IF i < n-1 THEN
11              i := i+1 /* successful instantiation: go forward */
12              Similarity Search(root of R, i)
13              sort entries ex in domain[i] in ascending order with respect to violationsx
14          ELSE /* the last variable is been instantiated */
15              minViolations := violations[i]
16              bestSolution=current assignment
17              IF minViolations=0 THEN RETURN
18      IF i=1 THEN RETURN /* end of algorithm */
19      i:=i-1/* Backtrack */
20  } /* END WHILE */
```

Similarity Search (Node N, integer i)

```
1   FOR EACH entry ex of N
2       FOR EACH Qij such that Qij = True and 1≤j<i
3           IF ex does not intersect wj THEN violationsx=violationsx+1
4           IF violationsx + violations[i-1]≥ minViolations THEN GOTO to next entry
5       IF N is intermediate node THEN Similarity Search (ex, i)
6           ELSE
7               Domain[i]=Domain[i]∪{ex}
```

Figure 18: Indexed branch and bound

As an example of *IBB*, consider again the clique query and the datasets of figure 17. Assume that initially v_1 is instantiated to object $r_{1,6}$. *Similarity search* (R_2,2) will be called to retrieve the possible values for v_2. Since at this point no solution has been found, v_2 could take any value in its domain (*ILA* would pick the first rectangle found). *Similarity search*, however, performs a window query (using the value $r_{1,6}$) and retrieves $r_{2,4}$ (it is the only object that overlaps $r_{1,6}$) which gets assigned to v_2. The next step (instantiation of v_3) will try to find rectangles in the third dataset that intersect $r_{1,6}$ and/or $r_{2,4}$. These rectangles are then sorted according to the number of conditions that they satisfy (i.e., the number of windows they overlap). In the current example, there is no rectangle that intersects both windows, so $r_{3,5}$ (which intersects $r_{2,4}$, but not $r_{1,6}$) gets assigned to v_3, and the value of

minViolations becomes 1. Since no other value for the second variable can lead to a better solution, the algorithm will immediately backtrack to re-instantiate the first one. The experimental evaluation will show whether the overhead of computing and sorting the domains in *IBB* pays off.

5. Experimental Evaluation

Since the goal is not retrieval of all exact solutions, we chose to compare the algorithms, not using the standard spatial database settings, but according to the common CSP and optimization methodology with 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*. Many studies on systematic search algorithms (Crawford and Auton 1993, Smith 1994) and non-systematic heuristics (e.g., local search, Clark et al. 1998) 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 generalized formula for the expected output size of multiway spatial joins is:

Sol = #(possible tuples)·Prob(a tuple is a solution)

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 (Theodoridis et al. 2000) 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 equation 2 is not accurate. Based on the fact that if a set of rectangles mutually overlap, then they must share a common area, Papadias et al. (1998) 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 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:

[1] 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. Density is related with the average rectangle extent by the equation $d = N|r|^2$ (Theodoridis et al. 2000).

$Sol = N^n \cdot n^2 \cdot |r|^{2\cdot(n-1)} = N \cdot n^2 \cdot d^{n-1}$

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 $d = 1/\sqrt[n-1]{N \cdot n^2}$. In the rest of the section we experimentally evaluate the proposed algorithms starting with non-systematic heuristics and continuing with systematic search. The following experiments were executed by Pentium III PCs at 500 MHz with 512MB Ram.

5.1 Comparison of Non-Systematic Heuristics

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[3] number of solutions is 1. Each query was allowed to execute for $10 \cdot n$ seconds. Figure 19 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 under-constrained, 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 20 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 21 shows the best similarity as a function of *Sol*.

[2] To the best of our knowledge, there do not exist 5 or more real datasets covering the same area publicly available.
[3] Since the expected number of solutions does not always coincide with the actual number, we had to repeat the generation of the datasets until the actual number was also 1. This is needed for the comparison of systematic algorithms.

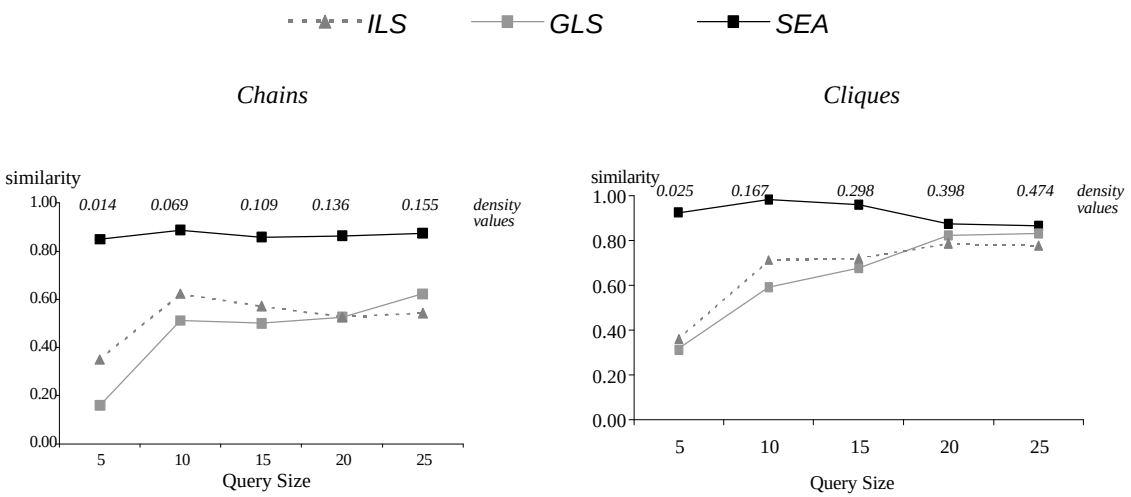


Figure 19: Best similarity retrieved as a function of *n*

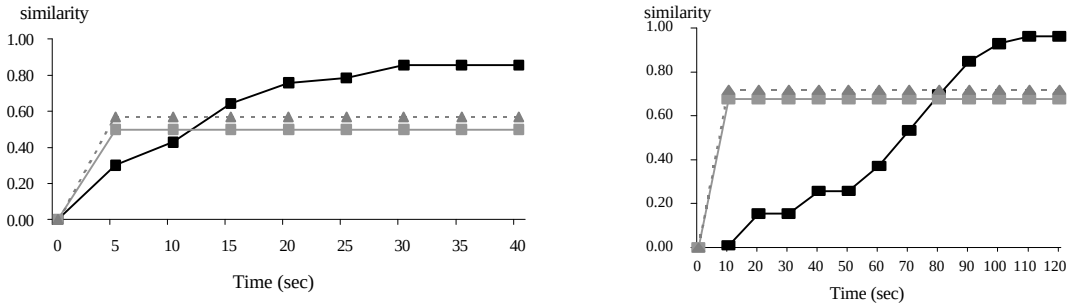


Figure 20: Best similarity retrieved as a function of time

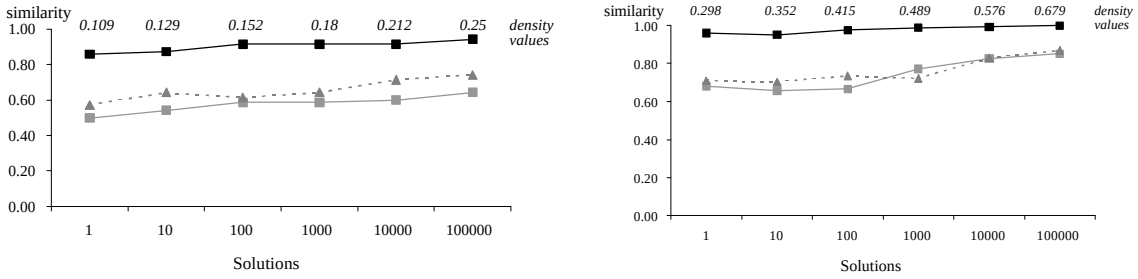


Figure 21: Best similarity retrieved as a function of the expected number of solutions

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 21 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.

The poor performance of *ILS* (and *GILS*) is rather surprising considering that local search significantly outperformed a genetic algorithm in the experimental evaluation of (Papadias et al. 1999) 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 proposed 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 (Papadias et al. 1999) 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 (Papadias et al. 1999) 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 20, 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.

Summarizing, *SEA* shows the best overall performance provided that it has sufficient time to converge. Since the performance gap is significant for both chains and cliques and all query sizes and data densities tested, the same results are expected for any problem instance. Furthermore, we anticipate further improvements by more refined, and possibly application-dependent, parameter tuning. *ILS* and *GILS* are preferable only in applications where the response time is more crucial than the quality of the solutions. If, for instance, in a military application an urgent decision is to be taken according to the best information available, *ILS* could be executed and return the first local maximum found within milliseconds. *GILS* outperforms *ILS* only for large queries. In addition to their usefulness as independent retrieval techniques, the above heuristics can be applied as a preliminary step to speed up systematic algorithms.

5.2 Comparison of Systematic Algorithms

In order to measure the performance of *ILA* and *IBB* we performed another set of experiments, using the datasets of the previous section. In particular, the first experiment (figure 22) illustrates the time required by the algorithms to locate the best solution as a function of *n*, for chain and clique queries. Similar to figure 21, the second experiment in figure 23 illustrates the processing time as a function of the expected number of solutions using the same datasets and queries of 15 variables.

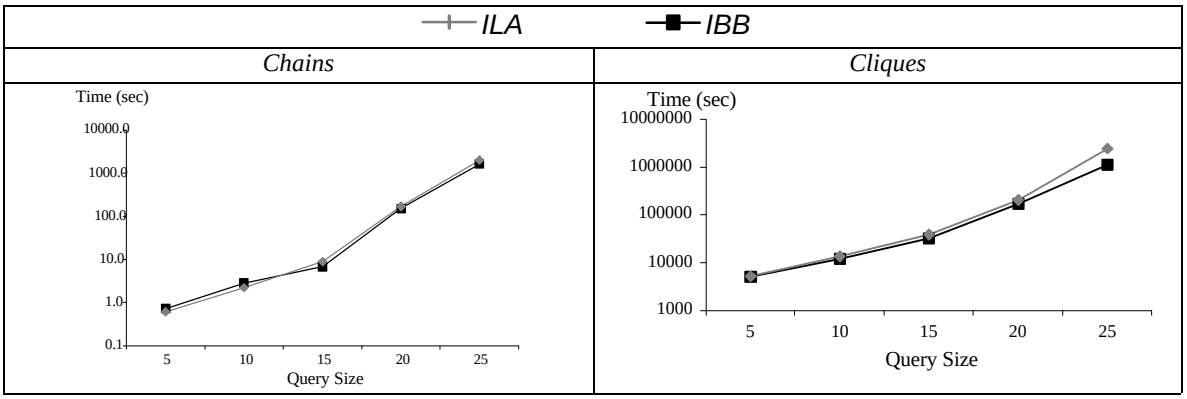


Figure 22: Time to retrieve the exact solution as a function of *n*

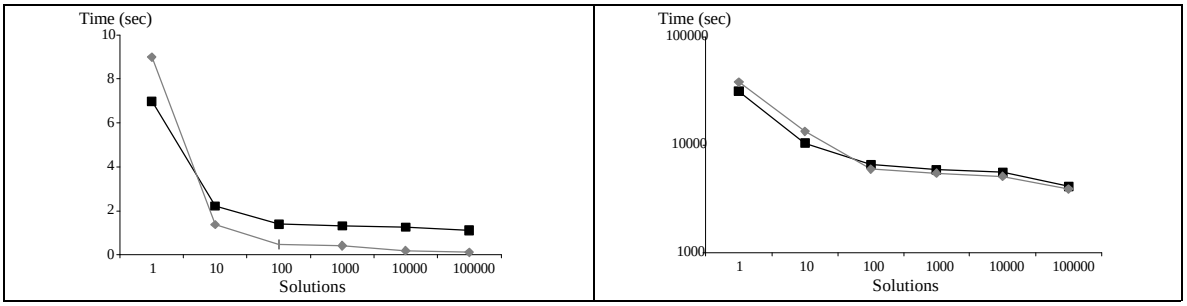


Figure 23: Time to find an exact solution as a function of the expected number of solutions (*n*=15)

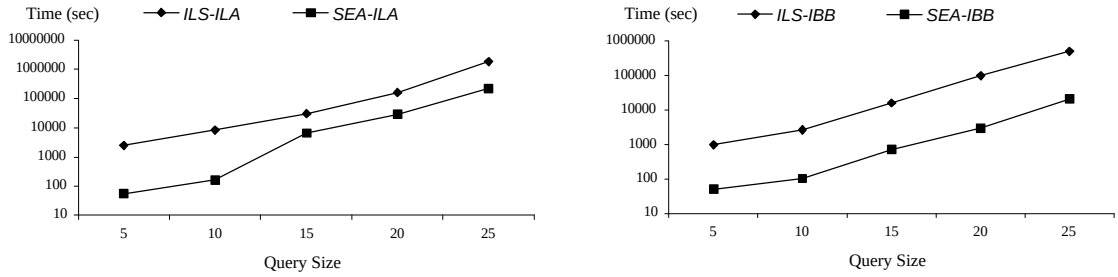
The performance of the algorithms is similar (in the same order of magnitude) with *IBB* being slightly better. As shown in figure 23, both algorithms are sensitive to the expected number of solutions; the processing time drops significantly as solutions multiply. For large number of solutions *ILA* has a slight advantage over *IBB* because by choosing "blindly" the first good value for a variable there is a high chance that it will eventually reach a good solution without the overhead of sorting. The important outcome of these experiments, however, refers more to the actual processing times rather than the relative performance of the algorithms. Notice in figure 22, that chain queries involving up to 15 datasets are answered very fast (in less than 10 seconds); thus non-systematic heuristics are not necessary for these cases. Even chains with 25 variables require about 1000 seconds which may be acceptable time given the size of the problem. On the other hand, even the smallest clique (5 datasets) needs more than 100 minutes. Cliques involving 25 variables take several days to terminate!

Approximate[4] retrieval of over-constrained queries is extremely time consuming because (i) it is difficult for the algorithms to quickly find good solutions that will prune the search space and (ii) even solutions with relatively high similarity may involve numerous violations. Thus, partial solutions can only be abandoned during the instantiation of the last few variables. In order to overcome the first deficiency (i.e., the *slow-start problem*) and speed up performance of *ILA* and *IBB* for difficult problem instances, we first apply a non-systematic heuristic to quickly locate a good solution. This solution provides an initial target for systematic search; any partial solution that cannot reach this target will be abandoned. We follow two approaches in this two-step search method. The first one executes *ILS* and returns the best local maximum visited, while the second executes *SEA* for a predetermined time threshold during which *SEA* has enough time to retrieve a very good solution. *ILS* is very fast,

[4] In contrast, if the goal were retrieval of exact solutions only, the processing time would decrease with the constrainedness of the query graph because once a violation were found the algorithms would immediately backtrack.

while *SEA* needs more time, which is hopefully compensated by the quality of the target solution (the higher the similarity, the more pruning of the search space during systematic search).

Figure 24 illustrates the total time (sum for systematic and non-systematic search) required to retrieve the exact solution using the two-step approaches for clique queries of 15 variables. The results are averaged over 10 executions because the quality of the solution returned by non systematic search 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 20), while *ILS* is executed for 1 second (during this time it visits about 12,000 maxima and returns the best).



(a) *ILA* (a) *IBB*

Figure 24: Processing time of two-step strategies as a function of n for clique queries

Although even the incorporation of *ILS* improves the performance of systematic search significantly (compare with figure 22b), the major difference is caused by *SEA*. The high similarity of the solution returned by *SEA* speeds up both algorithms 1-2 orders of magnitude with respect to simple systematic search. The improvement is more substantial for *IBB* because once a good target similarity is found, its branch-and-bound search strategy will lead it fast to the best solution. For instance, the 25-variable clique can now be processed in about 10^4 seconds as opposed to about 10^6 seconds in figure 22b. On the other hand, *ILA* is more sensitive to the total number of violations in the solution returned by non-systematic search. For 15 or more query variables, even good solutions may have several violations, which do not permit the pruning of a large part of the search space.

In the last experiment we compare *SEA-ILA* and *SEA-IBB* for various query types. In particular we use the 15 datasets originally created for cliques (d=0.298) and apply increasingly tight queries[5]. In order to generate such queries we start with a random one involving 15 conditions (contains a cycle) and at each step we add another 15 conditions at random. Figure 25 illustrates the time required by the algorithms to find the first exact solution as a function of the join conditions. The results are again averaged over 10 executions and include the time for both systematic and non-systematic algorithms.

[5] For 15 variables, the least constrained (acyclic) graph has 14 conditions, while the most constrained one (clique) has 105. We do not consider non-connected query graphs, as these can be processed by solving connected sub-graphs and computing their Cartesian product.

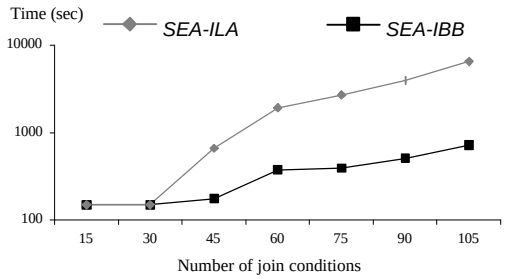


Figure 25: Processing time of *SEA-ILA* and *SEA-IBB* as a function of the query constrainedness (*n*=15)

For queries with 15 and 30 join conditions there is no difference because *SEA* retrieved exact solutions in all 10 executions. This not surprising because, as datasets are dense and queries are sparse, there exist numerous exact solutions. When the query constrainedness increases, the number of solutions drops since density remains constant. For 45 or more conditions, the cases where *SEA* finds an exact solution gradually decrease, and the relative performance of *ILA* and *IBB* becomes important. As shown, *SEA-IBB* clearly outperforms its *ILA* counterpart, and their performance gap widens with the number of conditions. Similar differences were also observed with several other query topologies.

Summarizing, the integration of systematic and non-systematic search, *SEA-IBB* in particular, is the best approach for dealing with retrieval of the solution with the highest similarity. The extension for retrieval of all such solutions is trivial. Notice that, if somehow we know that there exist exact solutions, previous methods for multiway spatial join processing (e.g, *PJM*) are preferable. This, however, is rather uncommon because even if the datasets are dense and the queries under-constrained, there may exist no solutions due to skewed data distributions. Present analytical formulae for the expected number of solutions only apply to uniform datasets.

6. Discussion

In this paper we propose several algorithms for multiway spatial join processing when the goal is (i) retrieval of the best solutions possible within limited time (ii) retrieval of the best existing solution without time limit. For the first goal we propose heuristics based on local and evolutionary search that take advantage of spatial indexes to significantly accelerate search. The best algorithm, *SEA*, can usually find optimal solutions even for difficult problems. For the second goal we propose systematic algorithms that combine indexes with backtracking. 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 approximate 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.

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.

Acknowledgements

This work was supported by the Research Grants Council of the Hong Kong SAR, grants HKUST 6090/99E, HKUST 6070/00E and HKUST 6081/01E.

References

ARGE, L., PROCOPIUC, O., RAMASWAMY, S., SUEL, T., and VITTER, J., 1998, Scalable Sweeping-Based Spatial Join. In Proceedings of *VLDB Conference*, 570-581.

BECKMANN, N., KRIEGEL, H.P. SCHNEIDER, R., and SEEGER, B., 1990, The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. In Proceedings of *ACM SIGMOD Conference*, 322-331.

BLICKLE, T. and THIELE, L., 1996, *A Comparison of Selection Schemes used in Genetic Algorithms* (2nd Edition), TIK-Report No. 11, Computer Engineering and Communication Networks Lab (TIK), ETH, Zurich.

BRINKHOFF, T., KRIEGEL, H.P., and SEEGER B., 1993, Efficient Processing of Spatial Joins Using R-trees. In Proceedings of *ACM SIGMOD Conference*, 237-246.

CHANG, S., SHI, Q., and YAN C., 1987, Iconic Indexing by 2-D String. *IEEE PAMI* 9(3), 413-428.

CLARK, D., FRANK, J., GENT, I., MACINTYRE, E., TOMOV, N., and WALSH, T., 1998, Local Search and the Number of Solutions. In Proceedings of *Constraint Programming Conference*, 325-339.

CRAWFORD, J. and AUTON, L., 1993, Experimental Results on the Crossover Point in Satisfiability Problems. In Proceedings of *AAAI Conference*, 21-27.

DAVENPORT, A., TSANG, E., WANG, C., and ZHU, K., *1994*, GENET: A Connectionist Architecture for Solving Constraint Satisfaction Problems by Iterative Improvement. In Proceedings of *AAAI Conference*, 325-330.

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

GLOVER F., and LAGUNA, M., 1997, *Tabu Search*. (London: Kluwer).

GOLDBERG, D., 1989 *Genetic Algorithms in Search, Optimization and Machine Learning* (Reading: Addison-Wesley).

GREFENSTETTE, J., 1986 Optimization of Control Parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 16 (1), 122-128.

HOLLAND, J., 1975, *Adaptation in Natural and Artificial Systems* (Ann Arbor: University of Michigan Press).

HUANG, Y.W., JING, N., and RUNDENSTEINER, E., 1997, Spatial Joins using R-trees: Breadth First Traversal with Global Optimizations. In Proceedings of *VLDB Conference*, 396-405.

KOUDAS, N., and SEVCIK, K., 1997, Size Separation Spatial Join. In Proceedings of *ACM SIGMOD Conference*, 324-335.

LEE, S, YANG, M, and CHEN, J., 1992, Signature File as a Spatial Filter for Iconic Image Database. *Journal of Visual Languages and Computing*, 3, 373-397.

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

LO, M-L., and RAVISHANKAR, C.V., 1994, Spatial Joins Using Seeded Trees. In Proceedings of *ACM SIGMOD Conference*, 209-220.

LO, M-L., and RAVISHANKAR, C.V., 1996, Spatial Hash-Joins. In Proceedings of *ACM SIGMOD Conference*, 247-258.

MAMOULIS, N., and PAPADIAS, D., 1999, Integration of Spatial Join Algorithms for Processing Multiple Inputs. In Proceedings of *ACM SIGMOD Conference*, 1-12.

MINTON, S. JOHNSTON, M., PHILIPS, A., and LAIRD P., 1992, Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence* 58(1-3), 161-205.

PAPADIAS, D., MAMOULIS, N., and DELIS, V., 1998, Algorithms for Querying by Spatial Structure. In Proceedings of *VLDB Conference*, 546-557.

PAPADIAS, D., MAMOULIS, N., and THEODORIDIS, Y., 2001, Constraint-Based Processing of Multiway Spatial Joins. *Algorithmica* 30(2): 188-215.

PAPADIAS, D., MANTZOUROGIANNIS, M., KALNIS, P., MAMOULIS, N., and AHMAD, I, 1999, Content-Based Retrieval Using Heuristic Search. In Proceedings of *ACM SIGIR Conference*, 168-175.

PAPADOPOULOS, A., RIGAUX P., and SCHOLL, M., 1999, A Performance Evaluation of Spatial Join Processing Strategies. In Proceedings of *SSD Conference*, 286-307.

PATEL J.M., and DEWITT D.J., 1996, Partition Based Spatial-Merge Join. In Proceedings of *ACM SIGMOD Conference*, 259-270.

PETRAKIS, E., and FALOUTSOS, C., 1997, Similarity Searching in Medical Image Databases. *IEEE TKDE*, 9 (3) 435-447.

SMITH, B., 1994, Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In Proceedings of *ECAI Conference*, 100-104.

THEODORIDIS, Y., STEFANAKIS, E., and SELLIS, T., 2000, Efficient Cost Models for Spatial Queries Using R-Trees. *IEEE TKDE* 12(1): 19-32.