A Genetic Algorithm for Database Query Optimization

Kristin Bennett **Michael C. Ferris** **Yannis E. Ioannidis**
Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706

Abstract

Current query optimization techniques are inadequate to support some of the emerging database applications. In this paper, we outline a database query optimization problem and describe the adaptation of a genetic algorithm to the problem. We present a method for encoding arbitrary binary trees as chromosomes and describe several crossover operators for such chromosomes. Preliminary computational comparisons with the current best–known method for query optimization indicate this to be a promising approach. In particular, the output quality and the time needed to produce such solutions is comparable to and in general better than the current method.

1 INTRODUCTION

Genetic algorithms [4, 6] are becoming a widely used and accepted method for very difficult optimization problems. In this paper, we describe the implementation of a genetic algorithm (GA) for a problem in database query optimization. In order to give a careful formulation of our GA, we first give a broad outline of this particular application.

The key to the success of a Database Management System (DBMS), especially of one based on the relational model [3], is the effectiveness of the query optimization module of the system. The input to this module is some internal representation of a query q given to the DBMS by the user. Its purpose is to select the most efficient *strategy* (algorithm) to access the relevant data and answer the query. Let $\mathcal{S}$ be the set of all strategies appropriate to answer a query q. Each member s of $\mathcal{S}$ has an associated cost $c(s)$ (measured in terms of CPU and/or I/O time). The goal of any optimization algorithm is to find a member s_0 of $\mathcal{S}$ that satisfies

$$c(s_0) = \min_{s \in \mathcal{S}} c(s).$$

Query optimization has been an active area of research ever since the beginning of the development of relational DBMSs. Good surveys on query optimization and other related issues can be found elsewhere [9, 10].

In the relational model, data is organized in *relations*, i.e., collections of similar pieces of information called *tuples*. Relations are the data units that are referenced by queries and processed internally. A *strategy* to answer a query q is a sequence of relational algebra operators applied to the relations in the database that eventually produces the answer to q. The cost of a strategy is the sum of the costs of processing each individual operator. Among these operators, the most difficult one to process and optimize is the *join*, denoted by $\bowtie$. It essentially takes as input two relations, combines their tuples one-by-one based on certain criteria, and produces a new relation as output. Join is associative and commutative, so the number of alternative strategies to answer a query grows exponentially with the number of joins in it. Moreover, a DBMS usually supports a variety of *join methods* (algorithms) for processing individual joins and a variety of *indices* (data structures) for accessing individual relations, which increase the options even further. Thus, all query optimization algorithms primarily deal with join queries. These are the focus of this paper as well.

In current applications, each query usually involves a small number of relations, e.g., less than 10. Hence, although exponential in the number of joins, the size of the strategy space is manageable. Most commercial database systems use variations of the same query optimization algorithm, which performs an exhaustive search over the space of alternative strategies, and whenever possible, uses heuristics to reduce the size of that space. This algorithm was first proposed for the System–R prototype DBMS [14], so we refer to it as the System–R algorithm.

Current query optimization techniques are inadequate

to support the needs of some of the newest database application domains, such as artificial intelligence (e.g., expert and deductive DBMSs), CAD/CAM (e.g., engineering DBMSs), and other disciplines (e.g., scientific DBMSs). Simply put, queries are much more complex both in the number of operands and in the diversity and complexity of operators in the query. This greatly exacerbates the difficulty of exploring the space of strategies and demands that new techniques be developed.

One of the proposed solutions is to use randomized algorithms. Simulated Annealing, Iterative Improvement, and Two-Phase Optimization (a combination of the first two) have already been successfully tried on query optimization [8, 15, 7], giving ample reason to believe that a GA will perform well also. Many of the operators used in these studies can be adapted for use in a GA and incorporated into a standard GA code. An advantage of our version of the GA [1], is that it is designed for a parallel architecture and significant computational savings over the other randomized methods can be obtained by a parallel implementation.

This paper is organized as follows. Section 2 defines two strategy spaces that are of interest to query optimization, which were used in our experiments. It also contains a description of the System–R algorithm, which is used as a basis for comparison of our results. Section 3 describes the specific genetic algorithm that we developed, including the representation that we used for the chromosomes for the two strategy spaces and the adopted crossover operators. Section 4 contains the results of our experiments. Finally, Section 5 gives a summary and provides some direction for future work.

2 QUERY OPTIMIZATION SPECIFICS

2.1 STRATEGY SPACES

Most query optimizers do not search the complete strategy space $\mathcal{S}$, but a subset of it, which is expected to contain the optimum strategy or at least one with similar cost. To understand the various options, we need some definitions related to databases. In a slight abuse of notation, consider the following query:

$$(A \bowtie C) \; and \; (B \bowtie C) \; and \; (C \bowtie D) \; and \; (D \bowtie E) \; and \; (D \bowtie F) \qquad (1)$$

Each join is associated with a constraint (omitted for clarity of presentation) that specifies precisely which tuples of the joined relations are to appear in the result. Query (1) can be represented by a *query graph* [16], which has the query relations as nodes and the joins between relations as undirected edges, as shown in Figure 1. Throughout, we use capital letters to denote relations and numbers to represent joins. In this

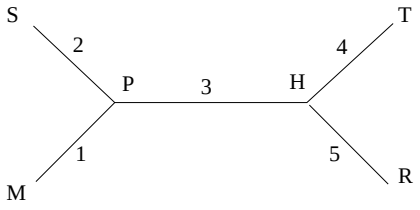

Figure 1: Query Graph

paper, we study tree queries, i.e., queries whose query graph is a tree. The answer to a given query is constructed by combining the tuples of all the relations in a query based on the constraints imposed by the specified joins. This is done in a step-wise fashion, each step involving a join between a pair of relations whose tuples are combined. These can be relations originally stored in the database or results of operations from previous steps (called *intermediate relations*). As a very strong and effective heuristic, database systems never combine relations that are not connected with a join in the original query. This is because such an operation produces the cartesian product of the tuples in the two relations. Not only is this an expensive operation, but its result is also very large, thus increasing the cost of subsequent operations. Most query optimizers confine themselves into searching the subspace of $\mathcal{S}$ of strategies with no cartesian products. This heuristic is adopted in the work presented in this paper as well.

Given the above, each strategy to answer a query can be represented as a *join processing tree*. This is a tree whose leaves are database relations, internal nodes are join operators, and edges indicate the flow of data from bottom-up. In addition, the chosen index for each database relation and the chosen join method for each join is specified. If all internal nodes of such a tree have at least one leaf as a child, then the tree is called *linear*. Otherwise, it is called *bushy*. Most join methods distinguish the two join operands, one being the *outer* (left) relation and the other being the *inner* (right) relation. An *outer linear join processing tree (left–deep tree)* is a linear join processing tree whose inner relations of all joins are base relations. In this study, we deal with two strategy spaces: one that includes only left-deep trees, which is denoted by $\mathcal{L}$, and one that includes both linear and bushy ones, which is denoted by $\mathcal{A}$. Examples of a left-deep tree and a bushy tree for query (1) are shown in Figure 2 (avoiding the details of the join constraints and the join methods). The interest in $\mathcal{L}$ stems from the fact that many DBMSs are using it as their strategy space, and is the one on which the System–R algorithm can be applied. We experiment with $\mathcal{A}$ as well, because quite often the optimum strategy is not in $\mathcal{L}$. We present results for applying a genetic algorithm on both spaces and compare them with the results of applying the System–R algorithm on $\mathcal{L}$.

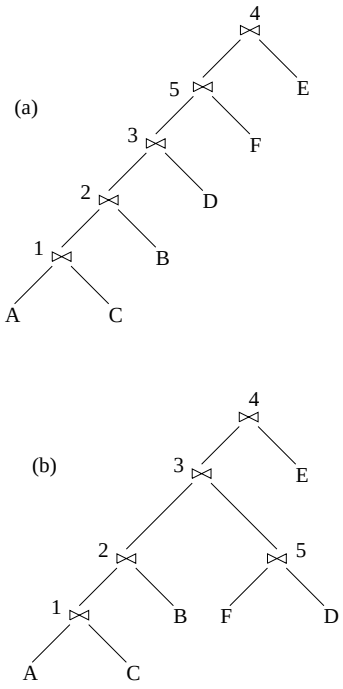

Figure 2: (a) Left-deep Tree; (b) Bushy Tree

We should emphasize at this point that, even after ignoring all strategies that contain cartesian products, the number of the remaining strategies still grows exponentially with the number of joins in the query. The above is true even for the $\mathcal{L}$ space, let alone for the $\mathcal{A}$ space, which is a superset of the former. The size of these spaces depends not only on the specific query which is being optimized (being dependent on the shape of its query graph), but also on the number of join methods supported by the system and the index structures that exist on the query relations. Hence, although cartesian products are excluded from strategies, even for the small queries with which we deal in this paper (up to 16 joins), the corresponding $\mathcal{L}$ and $\mathcal{A}$ spaces are very large and the associated optimization problem is computationally very difficult.

2.2 THE SYSTEM–R ALGORITHM

The System–R algorithm is based on dynamic programming. Specifically, the complete space $\mathcal{L}$ is constructed, occasionally pruning parts of it that are identified as suboptimal. The space is constructed by iteration on the number of relations joined so far. That is, at the k-th iteration, the best strategy to join k relations from the query is found, for all such sets of k relations. In the next iteration, strategies of $k+1$ relations are constructed, by combining each strategy from the previous collection with the appropriate remaining relations. For each set of $k+1$ relations, multiple strategies are usually constructed, of which only the one with the least cost is kept, since it can be shown that all the rest cannot be part of the optimum final strategy. This process needs as many iterations as there are relations in the query to complete. The main disadvantage of the algorithm is that it needs to maintain and process a very large number of strategies during its execution, a number that grows exponentially with the number of joins in the query. Especially towards the later iterations, this implies both a significant cpu cost for processing and a significant I/O cost due to increased page faults. This makes the algorithm inapplicable to queries with more than about 16 relations.

The above informal description of the algorithm is slightly simplified. In the interest of space, we have not discussed various complications that arise from side-effects that the use of specific types of indices can have on the desirability of a strategy. However, the version of the algorithm that was used in our experiments did address all these complications as well. The interested reader can find further details in the original paper on the System–R algorithm [14].

3 GENETIC ALGORITHM

In this section, we describe the implementation of a genetic algorithm to solve the problem outlined above. For completeness, we briefly review our terminology, details of which can be found in [1]. Our GA works with a population of chromosomes, each of which can be decoded into a solution of the problem. For each chromosome i in the population, a measure of its quality is calculated, called its fitness, $f(i)$. Chromosomes are selected from the population to become parents based on fitness. Then, reproduction occurs between pairs of chromosomes to produce offspring. The newly created population becomes the next generation and the process is repeated. The model that we use has a fixed population size N.

Our GA [1] uses a neighborhood scheme in which the fitness information is only transmitted within a local neighborhood, see for example [12]. A model algorithm for such a scheme is as follows.

Local Neighborhood Algorithm:

```
repeat
    for each chromosome i do
        evaluate f(i)
        broadcast f(i) in the neighborhood of i
        receive f(j) for all chromosomes j
            in the neighborhood
        select chromosome k to mate from the
            neighborhood of i based on fitness
        reproduce using chromosomes i and k
        replace chromosome i with one
            of the offspring
until population variance is small
```

In the experiments that we report below, we have used a neighborhood structure we call *ring6*, which considers chromosomes i and j to be neighbors if

$$\min(|i-j|, |i+N-j|, |i-N-j|) \leq 3$$

This can be viewed as each chromosome residing on a ring with neighbors that are chromosomes no further than three links away. This neighborhood structure was chosen since it has proven very effective in other problem instances [1]. For query optimization, the aim is to minimize the cost of the strategy, so to generate a fitness distribution in a neighborhood, we take the negative of the cost (to convert to maximization) and use a linear scaling of these values in each neighborhood so that the maximum fitness is some proportion (user supplied) of the average fitness. Details of this scaling routine are found in [1]. Since each chromosome i only selects one mating partner, selection is carried out by choosing chromosome k from the neighborhood with probability

$$f_k / \sum_{j \in \text{nhd(i)}} f_j .$$

Reproduction produces two offspring. The current chromosome is replaced with its best offspring provided this offspring is better than the worst chromosome in the neighborhood (see [12]).

For a GA to be an efficient optimization technique, we believe that reasonable parameter and algorithm choices (such as those detailed above) can be made without reference to a particular problem. However, the effectiveness of the overall algorithm depends crucially on the representation of the problem and the operators we use to exchange genetic information. Thus, we now specialize to the particular problem of query optimization. We describe two ways of encoding this problem which correspond to the two strategy spaces $\mathcal{L}$ and $\mathcal{A}$. We attempt to incorporate as much problem specific information as possible (see [5]), and show how mutation, initial population choice and crossover are carried out. Our discussion is broken into three parts, the first dealing with left–deep strategies, the second with bushy strategies, and the third dealing with crossover operators.

3.1 LEFT–DEEP STRATEGIES

Left–deep strategies ($\mathcal{L}$) are a relatively small subset of the strategy space $\mathcal{A}$. However, it has been observed that frequently a strategy exists in $\mathcal{L}$ whose cost is very close to the optimal cost over $\mathcal{A}$. System–R uses $\mathcal{L}$ as its strategy space, so it is appropriate to apply a GA in $\mathcal{L}$ and compare the results of these two methods. The advantage of choosing $\mathcal{L}$ over $\mathcal{A}$ is that the search space is much smaller (although still large in absolute terms); the disadvantage is that we cannot beat System–R in terms of output quality using a GA in $\mathcal{L}$.

Each chromosome represents a left–deep strategy. The chromosome is an ordered list of genes, where each gene consists of a relation and a join method. For example,

$$^{J}A \ ^{J}C \ ^{J}B \ ^{J}D \ ^{J}F \ ^{J}E$$

represents the left–deep strategy in Figure 2(a), with J representing some join method.

In our code, we associate the join method with the inner (right) relation of each join. Thus, to recreate the join processing tree, join the first and second relations using the method associated with the second relation. Then join the resulting intermediate relation with the next relation according to the specified method. Repeat until no relations remain. At each step verify that an edge exists in the query graph (Figure 1) between the current relation and one of the relations that occurred previously in the chromosome. If no such edge exists, then the query strategy contains a cartesian product, and the chromosome is penalized with an infinite cost. Note that the join method associated with the first relation is ignored and that if the crossover method produces many cartesian products, then the GA will not perform well.

Using this encoding, the problem is similar to a constrained traveling salesman problem (TSP) with a choice of methods of transport between the cities. In fact we use this analogy to motivate our choices of crossover operator. However, the query optimization function is much more expensive to evaluate than typical TSP functions, since each join, or equivalently the cost of traveling *directly* between cities, can be dependent on the route previously taken and/or the future cities to be visited. As an indication of the above complexity, we want to emphasize that the cost of a join between two relations is a function of their sizes. That size depends directly on the precise set of joins that have occurred previously. It also depends on the joins that remain to be processed later, because much of the data that is necessary for their execution is contained in the two relations.

Mutation is a secondary operator used to guarantee connectedness of the search pace. In our implementation it is of two types. The first type changes the join method randomly, and the second swaps the order of two adjacent genes. A left–deep strategy generator ensures that the initial population contains no cartesian products. This is achieved by cycling through a randomly generated permutation of the relations, only adding a relation to our chromosome if this can be done without introducing a cartesian product. The join methods are generated randomly.

3.2 BUSHY STRATEGIES

Quite often the best strategy for a query is in $\mathcal{L}$. However, in order to produce better solutions that System–R, we must consider a larger strategy space, ($\mathcal{A}$), which contains both linear and bushy strategies. We encode such strategies into chromosomes by consider-

ing each join as a gene, so that k_o^J represents join k with some join method J and its constituent relations (found on the query graph) in o orientation (for instance (a)lphabetically or (r)everse–alphabetically). A chromosome is then an ordered list of these genes. An example is

$$1_a^J \; 5_r^J \; 2_r^J \; 3_a^J \; 4_a^J$$

which represents the strategy given in Figure 2(b).

The decoding of the list into a solution is more costly than the left–deep decoding but has the ability to represent many more strategies. The decoding process grows the bushy tree from the bottom up. It maintains a list of intermediate relations waiting to be joined. Scanning the chromosome from left to right, it finds the constituent relations in each join (gene) by examining the query graph. The orientation of the gene indicates which relation is the outer (left) and inner (right) relation. If the right relation has been used in the formation of some intermediate relation in the list, the latter is substituted as the right relation of the join and the intermediate relation is removed from the list. The same process is done for the left relation. The left and right relations are joined according to the method in the gene, and the resulting relation is added to the list. After all the genes are processed, one intermediate relation remains in the list. This corresponds to the root of the bushy tree.

Note that although the representation is similar to the one described for left–deep trees (it is an ordered list of genes), there are three important differences. First, the decoding scheme guarantees that the corresponding query strategy has no cartesian products. Thus, we only consider "feasible" strategies of our problem. Second, our representation is based on labeling joins, not relations. This is somewhat natural since there may be several intermediate relations in use at any given step of the decoding scheme and these relations can be easily associated with a join. This association is critical for simple computation. In the left–deep case, only one intermediate relation is constructed at each stage, so the problem of handling these relations does not occur. Third, the representation is not uniquely defined, i.e., several chromosomes can decode into the same tree.

The coding described above has the advantage that it may now be possible to beat the System–R solution. However, the search space has been greatly increased giving the GA a more difficult task. We believe that the decoding scheme is very important for this GA to perform well: the extra work that we carry out in decoding guarantees that the algorithm only considers feasible solutions of the problem, and furthermore, that we can use standard crossover schemes motivated by GA's for TSP and other database work to generate good chromosomes.

Mutation is carried out in two ways. The first is to randomly change the join method or the orientation. The second is to perform reordering of genes on the chromosome by transposing a gene with its neighbor. Together, these guarantee that the search space is connected. The initial population is generated randomly.

3.3 CROSSOVER

In order to complete the discussion of our method, we describe the two crossover operators that we investigated. In each of the encodings above, the chromosome is an ordered list of genes. As outlined above, in the left–deep case the genes can be identified by their relation letter and in the bushy case by their join number. We describe the crossover operators solely in terms of these genes.

The first method, modified two swap (M2S), modifies the local improvement algorithm given in [11] to incorporate information from both parents and was designed primarily for the left–deep case. It can be described as follows. Given two parent chromosomes, $\mathcal{X}$ and $\mathcal{Y}$, randomly choose two genes in $\mathcal{X}$ and replace them by the corresponding genes from $\mathcal{Y}$, retaining their order from $\mathcal{Y}$, to create one offspring. For example, in the left–deep case, suppose the parent chromosomes $\mathcal{X}$ and $\mathcal{Y}$ are given by

$$\begin{aligned} \mathcal{X} &= {}^mA \;\; {}^nC \;\; {}^mB \;\; {}^mD \;\; {}^nF \;\; {}^nE \;, \\ \mathcal{Y} &= {}^mB \;\; {}^nC \;\; {}^mD \;\; {}^mF \;\; {}^mE \;\; {}^nA \end{aligned}$$

where m and n represent particular join methods and we randomly choose genes labeled A and D. The resulting chromosome is

$${}^mD \;\; {}^nC \;\; {}^mB \;\; {}^nA \;\; {}^nF \;\; {}^nE$$

We interchange the roles of $\mathcal{X}$ and $\mathcal{Y}$ to create another offspring. The use of M2S was partly motivated by the *Swap* transformation that has been successfully used for database query optimization in the context of other randomized algorithms [7, 15]. The two transformations are quite similar, except that, as a crossover, the transformation takes into account two strategies, whereas in its previous use it simply operates on one. Note that most of the ordering information from one of the chromosomes is retained, and it is this information that is of primary importance in both decoding schemes outlined above. Therefore, the crossover operator retains most of the ordering information from one chromosome, and uses order information from the other chromosome to exchange two genes. In the bushy case, the modified chromosome can be very different from both of its parents (depending on the locus of the genes being exchanged) due to the decoding scheme; in the left–deep case the solution will look very similar to one parent, although there is a small possibility of introducing a cartesian product.

The second method, which we refer to as CHUNK, is adapted from [2, 12] and was designed primarily for the bushy case. Here, we generate a random chunk

of the chromosome as follows. Suppose the number of genes in the chromosome is l. The start of the chunk (of genes) is a uniformly generated random integer in $[0, l/2]$ and the length of the chunk is uniformly generated from $[l/4, l/2]$. Suppose we randomly generate the chunk $[3, 4]$, then one resulting chromosome copies the third and fourth genes of $\mathcal{X}$ into the same position in the offspring, then deletes the corresponding genes of $\mathcal{Y}$, using the remainder of $\mathcal{Y}$'s genes to fill up the remaining positions of the offspring. For example, if $\mathcal{X}$ and $\mathcal{Y}$ are given by

$$\mathcal{X} = 1_a^n\ 5_r^n\ 2_r^m\ 3_a^m\ 4_a^n\ ,$$
$$\mathcal{Y} = 3_r^n\ 5_a^n\ 1_r^m\ 4_a^m\ 2_a^m$$

where m and n represent particular join methods, the resulting chromosome is

$$5_a^n\ 1_r^m\ 2_r^m\ 3_a^m\ 4_a^m$$

Again, another chromosome is created by interchanging the roles of $\mathcal{X}$ and $\mathcal{Y}$. We arrived at this method after some experimentation. The essential motivation behind the above scheme is to force a reasonably sized subtree (between a quarter and a half of the size of the original tree) from one parent to be incorporated into the other parent with minimal disruption to the latter. Of course, the chunk may or may not correspond to a subtree, but to avoid excessive computation in determining a chunk, the above scheme was used. The ordering information on the chromosome crucially determines the strategy in our decoding scheme, so the crossover operator attempts to minimize ordering changes. Since our representation is many to one, forcing the chunk to have a large size generally results in some genetic information being exchanged, enabling the algorithm to look at a larger variety of solutions and hence generate better solutions in reasonable times.

Although the crossover operators were designed with particular strategy spaces in mind, both operators can be applied in the two strategy spaces since they essentially consider the chromosomes as an ordered list of genes. Thus, we experimented with all combinations of crossover operators and spaces.

4 PERFORMANCE RESULTS

In this section, we report on an experimental evaluation of the performance and behavior of the above genetic algorithm on query optimization compared to the System–R algorithm. First, we describe the testbed that we used for our experiments, and then we discuss the obtained results.

4.1 TESTBED

For our experiments we assumed a DBMS that supports the *nested–loops* and *merge–scan* join methods [14]. Tree queries were generated randomly whose size ranged from 5 to 16 joins. The limit on the query size was due to the inability of the System–R algorithm to run with larger queries, primarily because of its huge memory requirements. Moreover, not all generated 16-join queries were runnable by the System–R algorithm. Thus, for large queries, the genetic algorithm is clearly superior to the traditional algorithm.

In the interest of space, we do not present the precise cost formulas that were used in this study. They capture the I/O cost of the various join methods and indices used and can be found in any textbook on databases. We also avoid presenting any details on the assumed physical design of the database. The specifics are exactly as in previous studies [7].

We implemented all algorithms in C, and tested them on a dedicated DecStation 3100 workstation. All experiments were conducted with a population size of $N = 64$. Ten different queries were tested for each size up to 16 joins. However, System–R managed to terminate on only seven of the 16-join queries, so the results for that case represent a smaller number of queries. For each query, each algorithm was run five times.

4.2 OUTPUT QUALITY

We compare output quality based on the lowest cost chromosome in the final generation. The cost of the average output strategy produced by the algorithms as a function of the query size is shown in Figure 3(a). The x–axis is the number of joins in the query. The y–axis represents scaled cost, i.e., the ratio of the output strategy cost over the cost found by the System-R algorithm. For each size, the average over all queries of that size, of the average scaled output cost over all five runs of each query is shown.

The results are rather interesting. We observe that on the average, when GA is applied to $\mathcal{L}$, it fails to find the optimum strategy, but except for the largest queries (16 joins), it is clearly within a small range (10%) of optimality. For small queries (4-6 joins), both crossovers always find the optimum. As the query size grows, however, the algorithm becomes less stable and the quality of its output deteriorates, primarily due to the dramatic increase in the size of the strategy space.

When GA is applied to $\mathcal{A}$ the results improve. On the average, for all three sizes, the algorithm found a better strategy than the best left-deep tree, with the gains ranging up to 13%. Note that as query size grows, GA becomes relatively better than System–R. This is because, with increased query size, the relative difference between the best bushy strategies and the best left-deep strategies increases as well, so by searching a richer space, GA is able to improve on the output quality. A small set of experiments with an increased population size has given very promising results for further improving the output quality in large queries.

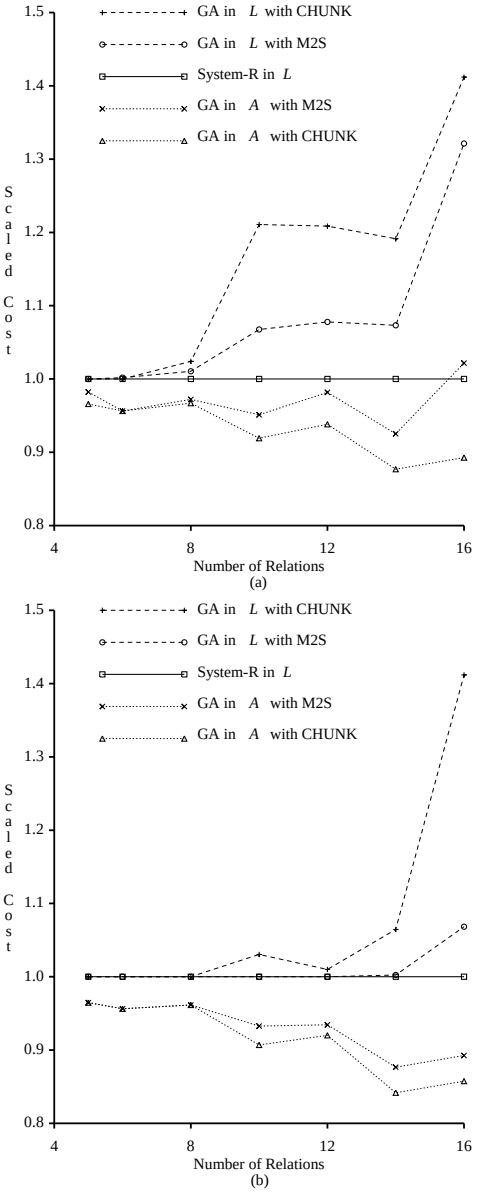

Figure 3: Scaled Cost of Strategy at Convergence: (a) average of 5 runs and (b) best of 5 runs

Another interesting comparison is that between the two crossovers. When GA is applied to $\mathcal{L}$, M2S is the preferred crossover, with CHUNK having much worse performance. This is due to the fact that, when the relations are the genes of the chromosome, applying CHUNK produces many offspring with cartesian products. Therefore, in that case, the algorithm spends much time in useless matings, thus failing to converge to a good strategy. On the other hand, M2S produces much fewer strategies with cartesian products and is the overall winner. Exactly the opposite happens when GA is applied in $\mathcal{A}$. CHUNK is the best performer, since it generates a large variety of strategies, thus looking at a larger area of the space and using its time much more effectively.

To overcome some of the inherent problems of randomized algorithms, it is occasionally proposed that such algorithms are run multiple times on a given instance problem, and the best solution among those found be chosen. With that in mind, we also compare the best output found among the five runs of each version of the GA algorithm for each query. We show the average of that over all queries of a given size in Figure 3(b). We now see that in $\mathcal{L}$, M2S is perfect, almost always finding the optimum strategy. CHUNK is considerably improved as well, but is still has inferior performance for the reasons explained above. Similar improvements are seen in the $\mathcal{A}$ space as well. Especially in the large joins, both crossovers find very good strategies. All these results indicate that multiple runs of the GA algorithm may be a plausible way to avoid some of its potential instabilities and produce high quality results.

4.3 TIME

The average time results are presented in Figure 4, where the x–axis represents the number of joins in the query, and the y–axis represents the processing time in seconds. For the 16-join queries on which System–R failed to finish, we use the time-to-failure in this figure. The results are as follows. System–R performs faster for queries of size up to 14, but the GA in $\mathcal{L}$ is much faster for queries of size 16. The increase in times for GA in $\mathcal{L}$ is almost linear. There is a larger increase in the time for GA in $\mathcal{A}$ than in $\mathcal{L}$. This is to be expected since $\mathcal{L}$ is a much smaller space than $\mathcal{A}$. As it is obvious from Figure 4, this increase is much less steep than the corresponding increase in time for System–R. We believe that even if System–R was runnable for queries beyond 16 joins, it would require much more time than GA.

A final comment that we want to make is that this version of GA is designed to be ported to a parallel machine. In a parallel implementation, each chromosome in the population resides on a processor and communication is carried out by message passing. The total communication overhead is thus minimal. Based on results on other optimization problems [1] where the evaluation of the fitness function dominates the processing time, as is the case with query optimization, we expect linear speedups in execution time. Since only limited parallelism can be incorporated into System–R, the time to execute the parallel GA should become much smaller than that of System–R.

5 CONCLUSIONS

We have presented a genetic algorithm for database query optimization. In doing so we have intro-

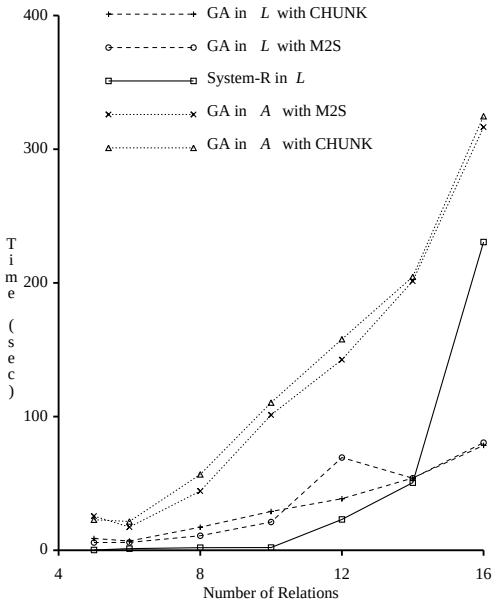

Figure 4: Average Processing Time

duced a novel encoding/decoding of chromosomes that represent binary trees together with associated new crossover operators. Although we did not exploit it in this paper, an important characteristic of the algorithm is its efficient parallelization. Our computational experiments with sequential implementations of the algorithm have shown the method to be a viable alternative to the commercially established algorithm. In fact, for large queries, one implementation of the GA found comparable solutions in much better time, whereas a different implementation found better quality solutions at the expense of additional time. Moreover, the GA was capable of optimizing large size problems on which the established algorithm fails.

In the future, we plan to adapt our parallel implementation of the GA to query optimization and verify our claims on its superiority over the System-R algorithm. In addition, we plan to investigate its applicability to query optimization in more complex database environments, e.g., parallel database machines.

Acknowledgements

The work described in this paper has been partially supported by The Air Force Laboratory Graduate Fellowship Program, the Air Force Office of Scientific Research under Grant AFOSR–89–0410 and the National Science Foundation under Grant IRI-8703592.

References

[1] E.J. Anderson and M.C. Ferris. A genetic algorithm for the assembly line balancing problem. In *Proceedings of the Integer Programming / Combinatorial Optimization Conference, Waterloo, September 1990*, Ontario, Canada, 1990. University of Waterloo Press.

[2] G.A. Cleveland and S.F. Smith. Using genetic algorithms to schedule flow shop releases. In Schaeffer [13], pages 160–169.

[3] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[4] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison–Wesley, Reading MA, 1989.

[5] J.J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In L.D. Davis, editor, *Genetic Algorithms and Simulated Annealing.* Pitman, London, 1987.

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

[7] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proc. of the 1990 ACM-SIGMOD Conference on the Management of Data*, pages 312–321, Atlantic City, NJ, May 1990.

[8] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. of the 1987 ACM-SIGMOD Conference on the Management of Data*, pages 9–22, San Francisco, CA, May 1987.

[9] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[10] W. Kim, D. Reiner, and D. Batory. *Query Processing in Database Systems.* Springer Verlag, New York, N.Y., 1986.

[11] S. Lin and B.W. Kernighan. An efficient heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[12] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In Schaeffer [13], pages 416–421.

[13] J.D. Schaeffer, editor. *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, California, 1989. Morgan Kaufmann Publishers, Inc.

[14] P. Selinger et al. Access path selection in a relational data base system. In *Proc. of the 1979 ACM-SIGMOD Conference on the Management of Data*, pages 23–34, Boston, MA, June 1979.

[15] A. Swami and A. Gupta. Optimization of large join queries. In *Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data*, pages 8–17, Chicago, IL, June 1988.

[16] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, 1982.