CENTER FOR
PARALLEL OPTIMIZATION

A GENETIC ALGORITHM FOR DATABASE QUERY OPTIMIZATION

by

Kristin Bennett
Michael C. Ferris
Yannis E. Ioannidis

Computer Sciences Technical Report #1004

February 1991

A Genetic Algorithm for Database Query Optimization

Kristin Bennett* Michael C. Ferris† Yannis E. Ioannidis‡

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706

February 6, 1991

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 [Hol75, Gol89] 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 [Cod70], 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 $\mathcal{S}$

*Supported by The Air Force Laboratory Graduate Fellowship Program
†Partially supported by the Air Force Office of Scientific Research under Grant AFOSR–89–0410
‡Partially supported by the National Science Foundation under Grant IRI-8703592

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 [JK84, KRB86]. Specifically, 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, so the size of the strategy space is relatively small. 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 [S+79], 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 tried on query optimization with some success [IW87, SG88, IK90], giving ample reason to believe that a GA will perform well. 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 [AF90], 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

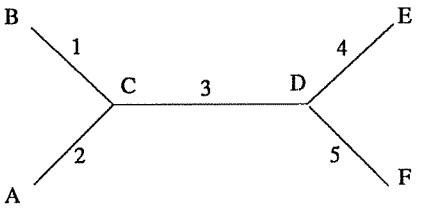

Figure 1: Query Graph

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:

$$(A \bowtie C) \textit{ and } (B \bowtie C) \textit{ and } (C \bowtie D) \textit{ and } (D \bowtie E) \textit{ and } (D \bowtie F) \qquad (1)$$

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* [Ull82], 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 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,

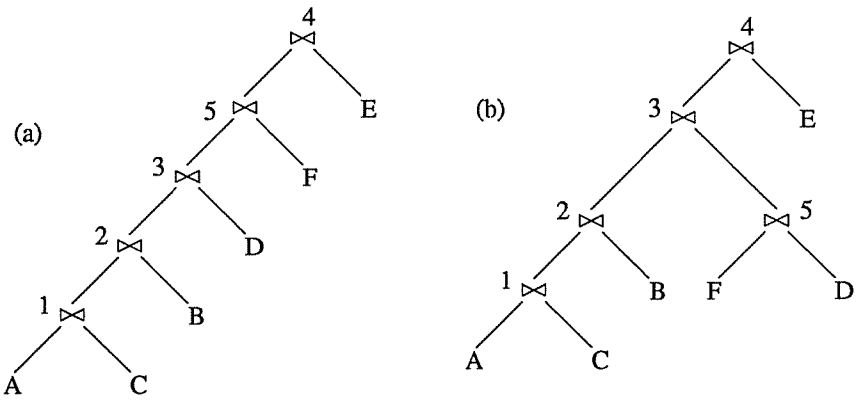

Figure 2: (a) Left-deep Tree; (b) Bushy Tree

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 $\mathcal{L}$.

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 its memory requirement, which grows exponentially with the number of joins in the query. This makes the algorithm inapplicable

to queries with more than 10 or 15 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 [S+79].

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 [AF90]. 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) after which reproduction (which for this implementation consists of crossover and mutation) occurs between pairs of chromosomes to produce offspring. The newly created population becomes the next generation and the process is repeated. The model we use has a fixed population size N.

Our GA [AF90] uses a neighborhood scheme in which the fitness information is only transmitted within a local neighborhood, see for example [Mö89]. A model algorithm for a scheme in which fitness information is only compared locally 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 is defined by stating that chromosomes i and j are considered 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. In the problem we are considering, the aim is to

minimize the cost of the strategy. To generate a fitness distribution in a neighborhood, we take the negative of this value (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 [AF]. 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 [Mö89]).

We now specialize to the particular problem of query optimization. We describe two ways of encoding this problem which attempt to incorporate as much problem specific information as possible (see [Gre87]), and show how mutation, initial population choice and crossover are carried out. We break our discussion 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

Each chromosome represents a left–deep strategy. A natural encoding of this search space is to let a gene consist of a relation and a join method. We associate the join method with the inner (right) relation of each join. A chromosome is then an ordered list of these genes, for example,

$$JA \;\; JC \;\; JB \;\; JD \;\; JF \;\; JE$$

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

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.

Using this encoding, the problem is similar to a constrained traveling salesman problem (TSP) with a choice of methods of transport between the cities. 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 the sizes of the relations. 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 those relations.

Mutation 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 $\mathcal{L}$. In order to look at $\mathcal{A}$, we encode a bushy tree into a chromosome by considering 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) ordered by o (for instance (a)lphabetically or (r)everse–alphabetically). The chromosome is then an ordered list of these genes

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

Note that this list represents the strategy given in Figure 2(b).

The decoding of this 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 from left to right, it finds the constituent relations in each join(gene) by examining the query graph. The order information in the gene indicates which relation is the outer (left) and inner (right) relation. If the right relation has been used in the formation of intermediate relation in the list, the intermediate relation is substituted as the right relation 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. The corresponding query strategy is guaranteed not to contain a cartesian product. Thus the decoding of the chromosomes guarantees that the constraints of the problem can never be violated. This coding has the additional 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.

Mutation is carried out in two ways. The first way is to randomly change the join method or the order information. The second way 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. 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 [LK73] to incorporate information from both parents and 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 the order from $\mathcal{Y}$, to create one offspring. For example, in the bushy case, suppose the parent chromosomes $\mathcal{X}$ and $\mathcal{Y}$ are given by

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

where m and n represent particular join methods and and we randomly choose genes labeled 1 and 3. The resulting chromosome is

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

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 used for database query optimization in the context of other randomized algorithms [SG88, IK90]. 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.

The second method, which we refer to as CHUNK, is adapted from [CS89, M̈89]. 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. In the example above, 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 $\mathcal{Y}$.

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 [S+79]. Tree queries were generated randomly whose size ranged from 5 to 15

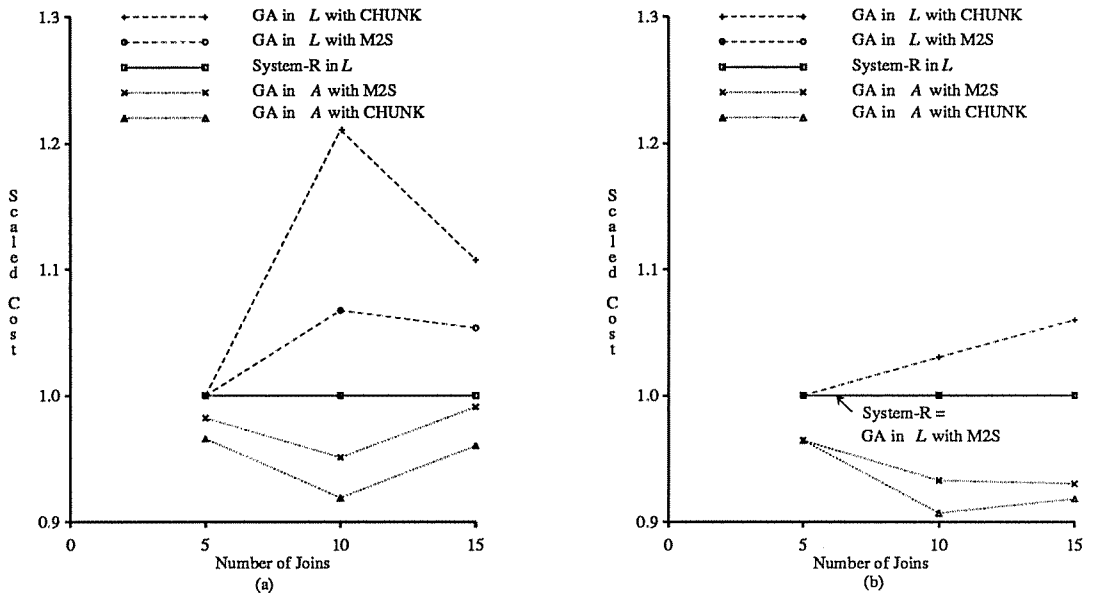

Figure 3: Scaled Cost of Strategy at Convergence: (a) average and (b) best of 5 runs

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 15-join queries were runnable by the System–R algorithm, so a large number of them were generated until we could find among them a large subset of runnable ones for our results to be meaningful. 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 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 [IK90].

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 15 joins. For each query, each algorithm was run five times.

4.2 Output quality

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 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 is clearly within a small range (20%) of

optimality. For small queries (5 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. A noteworthy observation is that when GA is applied to $\mathcal{L}$ for 15-join queries the results obtained are better than for 10-join queries. This is due to the specific 15-join queries used in our experiments. As we mentioned above, we worked with queries on which the System-R algorithm was runnable, i.e., queries whose corresponding $\mathcal{L}$ space was relatively small. This bias was helpful to GA on $\mathcal{L}$ as well, which thus had a relatively better performance with 15-join queries than with 10-join queries.

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 9%. Interestingly, the relative performance of the GA algorithm between 10-join and 15-join queries is exactly the opposite of what was observed in $\mathcal{L}$. In this case, the degradation in performance is primarily due to the significant increase in the size of the $\mathcal{A}$ strategy space for large queries. The bias towards queries that are easy for System-R has also an effect but not as dominant as in $\mathcal{L}$. A small set of experiments with an increased population size has given very promising results for improving the output quality in large queries as well.

Another interesting comparison is that between the two crossovers. When GA is applied in $\mathcal{L}$, 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.

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, 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. Moreover, there is no evidence for any significant differences between 10-join and 15-join queries as was for the average case. 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. Briefly, System–R performs faster for queries of size 5 and 10, but the GA in $\mathcal{L}$ is much faster for queries of

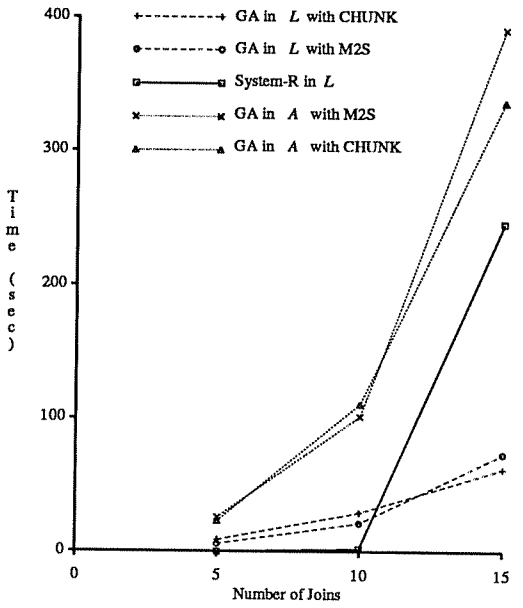

Figure 4: Average Processing Time

size 15. The increase in times for GA in $\mathcal{L}$ is almost linear. There is a large increase in the time for GA in $\mathcal{A}$ from problems of size 10 to 15. However, we believe that this increase is not exponential in the number of joins, whereas the increase in time for System–R is exponential. Also recall that large time differences can be expected since $\mathcal{L}$ is a much smaller space than $\mathcal{A}$.

Furthermore, this version of the 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 [AF] 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 introduced 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 for the problem. 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.

References

[AF] E.J. Anderson and M.C. Ferris. Genetic algorithms for combinatorial optimization: The assembly line balancing problem. In preparation.

[AF90] 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.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[CS89] G.A. Cleveland and S.F. Smith. Using genetic algorithms to schedule flow shop releases. In Schaeffer [Sch89], pages 160–169.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison–Wesley, Reading MA, 1989.

[Gre87] J.J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In L.D. Davis, editor, *Genetic Algorithms and Simulated Annealing.* Pitman, London, 1987.

[Hol75] J. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan, 1975.

[IK90] 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.

[IW87] 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.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[KRB86] W. Kim, D. Reiner, and D. Batory. *Query Processing in Database Systems.* Springer Verlag, New York, N.Y., 1986.

[LK73] S. Lin and B.W. Kernighan. An efficient heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[MÖ89] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In Schaeffer [Sch89], pages 416–421.

[S+79] 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.

[Sch89] J.D. Schaeffer, editor. *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, California, 1989. Morgan Kaufmann Publishers, Inc.

[SG88] 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.

[Ull82] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, 1982.