Seoul Journal of Business
Volume 2 Number 1

Characterization and Analysis of a Nested Genetic Algorithm for Distributed Database Design

Salvatore T. March
University of Minnesota

Sangkyu Rho
Seoul National University

Abstract

Distributed database design is a difficult and complex task involving two interdependent problems: data allocation and operation allocation. First, data must be allocated to nodes in the network. Second, given such an allocation, data must be efficiently retrieved, processed, and possibly communicated to meet the retrieval and update requirements of the users. The problem is characterized by integer variables, a discontinuous and extremely complex cost function, and numerous constraints. A nested genetic algorithm naturally fits this problem formulation with the outer algorithm addressing data allocation and the inner algorithm addressing operation allocation. We present and characterize such an algorithm according to its gene structure and control parameters. We experimentally analyze the effects of poolsize and crossover operator on the performance of our algorithm.

Index Terms - Genetic algorithms, performance modeling and analysis, experimental analysis of algorithms, distributed database design

1. INTRODUCTION

With the emergence of relatively inexpensive, high-capacity communications capabilities, geographically distributed databases (DDB) have become an integral part of many computer applications [Thomas, et al., 1990]. Such distributed systems can yield significant management and organizational advantages over centralized systems [King, 1983]. Judicious placement of data and processing capabilities can significantly reduce operating costs and response time. Inappropriate placement of data or poor data access strategies, however, can result in high cost and poor system performance [Ozsu and Valduriez, 1991].

Distributed database design involves two interrelated problems, data allocation and operation allocation [Apers, 1988; Cornell and Yu, 1989; Blankinship, et. al., 1991]. Data allocation defines what data is allocated to each node in the network (see [Dowdy and Foster, 1982] for a survey of methods). To enhance retrieval efficiency, the same data can be redundantly allocated to multiple nodes. However, such redundancy increases update costs.

Operation allocation defines where retrieval and processing operations (e.g., join) are performed. Each retrieval operation must be allocated to a node containing the required data. Processing operations can be allocated to any node, however, if the data are not located at the processing node, they must be sent over the communication network. Update operations must be done at all nodes containing a copy of the effected data, although update strategies can vary (e.g., delayed or "lazy" update).

Since data and operation allocation are interdependent, they must be solved simultaneously [Apers, 1988]. The optimal data allocation depends on the processing schedules of all queries accessing that data (i.e., the operation allocation). However, the optimal processing schedules depend on where data are located (i.e., the data allocation). Hence, to be effective a distributed database design model must comprehensively treat both data and operation allocation as a unified whole (see e.g., Apers [1988], Blankinship et al., [1991], Cornell and Yu [1989], March and Rho [1995] for comprehensive distributed database design models).

One of the difficulties facing researchers in this area is tractability. Even

simplistic models treating only one of the above problems are NP-hard [Eswaran, 1974; Hevner, 1979]. Many algorithms have been developed for various formulations and subproblems of the distributed database design problem. These include integer mathematical programming [Cornell and Yu, 1989], exhaustive enumeration [Lohman, et. al., 1986], branch and bound [Martin, et. al, 1990], dynamic programming [Lafortune and Wong, 1986], simulated annealing [Martin, et. al., 1990], genetic algorithms [March and Rho, 1995], and heuristic approaches [Chen and Yu, 1994].

A genetic algorithm has several advantages over other approaches. First, genetic algorithms have been successfully applied to large, complex, combinatoric, real-world problems (see, e.g., Davis [1991], Goldberg [1989a, 1994], Grefenstette [1993]). The genetic algorithm developed by March and Rho [1995] embeds a genetic algorithm for operation allocation within a genetic algorithm for data allocation, thus addressing a comprehensive formulation of the distributed database design problem[1]. Second, genetic algorithms are robust in that they work well even in discontinuous, multimodal, noisy search spaces [Goldberg, 1989a] common in distributed database design. Third, genetic algorithms result not only in a "best" solution, but also in a pool of good solutions. Thus from a practical perspective, they result in a number of good solutions from which a design may be chosen for implementation.

As with all heuristic procedures, the performance of a genetic algorithm can be evaluated by (1) the goodness of its results (e.g., how close to optimal) and (2) by its run time (e.g., computer resources required). While these are problem dependent, they are also affected by the control parameters used to define the genetic algorithm. These control parameters include poolsize, number of iterations, crossover operator, and mutation rate [Grefenstette, 1986]. In a nested genetic algorithm these parameters may vary independently for the outer (data allocation) and inner (operation allocation) genetic algorithms.

In this paper we first briefly describe genetic algorithms and their control

---

1) The cost model and a brief description of the genetic algorithm are presented in [March and Rho, 1995], the algorithm is presented in detail, characterized, and experimentally evaluated in this paper.

parameters using data allocation as an example. We then describe the nested genetic algorithm developed by March and Rho [1995] for the combined data and operation allocation problem and characterize it according to its control parameters. Next we describe a set of experiments used to analyze the effects of various control parameters on the performance of this algorithm. Finally, we summarize our results and present directions for future research.

2. GENETIC ALGORITHMS AND THEIR CONTROL PARAMETERS

Genetic algorithms [Holland, 1975; Goldberg, 1989a] apply principles of natural population genetics to a pool of candidate solutions, each of which is likened to the genes of a living organism. Genetic algorithms produce new solutions (offspring or children) by selecting solutions (parents) from the pool and combining components (genes) from them using some crossover operator. Some combination of parents and offspring are retained to keep the poolsize constant for the next iteration (generation). Mutation introduces random changes into the solution pool. Given a non-zero mutation rate, genetic algorithms can, theoretically, guarantee optimality. That is, given an infinite number of generations, all possible solutions will be generated. However, to be practical, a genetic algorithm must have a stopping condition (typically some number of generations). Thus, genetic algorithms can be considered to be heuristic procedures.

In order to apply a genetic algorithm to an optimization problem, we must (1) develop a gene-like representation of solutions, (2) determine the size of the solution pool to maintain, (3) determine how and how many solutions will be selected from the pool to be parents, (4) determine how and how many children will be generated from parents (including the possibility of mutation), (5) determine how solutions will be selected for the next generation, and (6) determine stopping conditions for the algorithm.

Consider the problem of allocating five data files each of 1 million characters to four nodes in a fully connected computer network where all nodes require access to all files at a rate of one access per day. Further, suppose that the objective is to minimize the sum of storage and communication

costs (as discussed briefly below, the cost model used in [March and Rho, 1995] is considerably more complicated). If the retrieval activities were such that no two nodes request the same file, then the optimal solution is obvious -- store each file at its retrieval node. When multiple nodes require the same data, then tradeoffs must be evaluated. If files are replicated and stored at each requesting node then communication costs are minimized but storage costs are increased.

For illustrative purposes, an example genetic algorithm is described for this problem corresponding to the above components.

(1) Represent a solution by five sets of four bits each, one set for each file. The four bits in each set represent where a copy of the file is stored. For example, the set 1000 stores the file at node 1, the set 1100 stores the file at nodes 1 and 2, etc. The solution (0011 0101 1110 0111 0100) stores file 1 at nodes 3 and 4; file 2 at nodes 2 and 4; file 3 at nodes 1, 2, and 3; file 4 at nodes 2, 3, and 4; and file 5 at node 2. Thus, for this algorithm the gene structure is defined as five sets of four bits each.

(2) Randomly generate an initial pool of 6 different solutions (likely a poolsize of 6 is too small; poolsize is a control parameter as discussed below). Figure 1 illustrates a possible initial pool.

〈Figure 1〉 Example Solution Pool with Performance and Fitness

Solution	Cost($ / day)	Fitness	Selection Probability
1111 1101 1010 0100 0100	911	.91	.169
0001 0100 0010 1000 1100	1406	.86	0.159
1011 1101 0110 1101 0101	713	.93	.172
0100 0010 1001 0111 0100	1208	.88	.163
0010 0101 1000 0100 0101	1307	.87	.161
1010 0101 1111 1111 0111	515	.95	.176
	6060	5.40	1.000

(3) Select two different solutions as parents (i.e., parent selection without

replacement) probabilistically based on the "fitness" of the solution. Figure 1 shows the cost, fitness, and selection probability for each of the initial solutions. As an example we calculate these values for the first solution in Figure 1, (1111 1101 1010 0100 0100). This solution stores 11 physical files of 1 million characters each (i.e., file 1 is replicated at all four nodes; file 2 is replicated at nodes 1, 2, and 4; file 3 is replicated at nodes 1 and 3; files 4 and 5 are stored only at node 2). Communication is required for each file not stored at a node since we assumed that each node requests each file once per day. Hence, 9 million characters must be transferred (each 0 in the gene structure requires that file to be transferred to that node). Node 1 requires communication for files 4 and 5; node 2 for file 3; node 3 for files 2, 4, and 5; and node 4 for files 3, 4, and 5. The cost is 11 million characters $\times$ s for data storage (where s is the cost per character per day for storage) and 9 million characters / day $\times$ t for data transfer (where t is the cost per character for data transfer). If s is \$.000001 / day and t is \$.0001 / character, then the total cost of this solution is \$911 / day (11,000,000$\times$.000001 + 9,000,000$\times$.0001). Fitness is defined to be one minus the solution's cost divided by a constant (10,000). Hence, the fitness of this solution is (1 - 911 / 10000), or .91. The total fitness of the pool is 5.40. Thus, the selection probability for the first solution is .91 / 5.40 or .169. Cost, fitness, and selection probability are similarly calculated for each of the other five solutions.

(4) Produce one child from the pair of parents selected using uniform crossover with no mutations (i.e., the mutation rate is 0.00). In uniform crossover the child inherits a value for each gene position from one parent or the other randomly, i.e., with probability .5 (variations of the crossover operator are discussed below). Thus, if the first two solutions in Figure 1 were selected as parents, the child would be: xxx1 x10x x010 xx00 x100 where the x positions have a .5 probability of being a 1 and a .5 probability of being 0. That is, positions 4, 6, 11, and 18 have a 1 in both parents -- the child will inherit a 1 for those positions no matter which parent is selected. Positions 7, 10, 12, 15, 16, 19, and 20 have a 0 in both parents -- the child will inherit a 0 for those positions. In the remaining positions one parent has a 1 and the other has a 0 -- the parent from which the child inherits a value for that position is randomly determined.

(5) Select the best 6 solutions for the next generation. Including the newly

generated child, there are 7 solutions in the pool. To keep the poolsize constant, remove the worst, leaving 6 solutions to survive into the next generation.

(6) Stop after 1000 generations. That is, repeat steps (2) through (5) nine hundred ninety nine times.

Although greatly simplified, the above example provides insight into why genetic algorithms are effective. As crossover combines solutions, a number of partial solutions, termed schemata, having good performance begin to emerge in multiple solutions. Solutions with good performance are expected to contain some number of good schemata. If the selection of parents is based on performance (i.e., fitness), such solutions are more likely to be selected as parents than those with poor performance (which are expected not to contain as many good schemata). Over successive iterations (generations), the number of good schemata represented in the pool tends to increase, and the number of bad schemata tends to decrease. Therefore, the average performance of the pool tends to improve.

All genetic algorithms share the basic structure described above. However, there are numerous variations and control parameters that impact the effectiveness and efficiency of a genetic algorithm.

In the rest of this section, we characterize the important variations and introduce control parameters for each of the 6 steps of a genetic algorithm described above. We present and characterize our distributed database design algorithm in the following section.

2.1 Gene Structure

The most common gene structure for genetic algorithms is a set of bit strings. Of course, we can arbitrarily group and order the bits. We could, for example, have 4 sets of 5 bits each where each set represents a node and each bit position in each set represents the allocation of a file to that node. Alternately, we can use a non-binary gene structure where the value in each position represents a solution component (e.g., as in genetic algorithms for job shop scheduling [Uckun, et.al., 1993]).

2.2 Poolsize and the Initial Pool

Poolsize significantly affects the effectiveness and efficiency of genetic algorithms [Grefenstette, 1986]. Genetic algorithms do poorly with very small poolsizes because a small pool is unlikely to contain sufficient genetic material to effectively represent the solution space. A poolsize of 6 for the data allocation problem described above is probably not large enough to adequately simulate natural genetics. With such a small poolsize, solutions would quickly become "inbred". The algorithm would converge on a local optima, failing to search significant portions of the solution space.

The pool should be large enough to insure a reasonable sample of the actual solution space, but not so large as to make the algorithm approach exhaustive enumeration. The five file, four node data allocation problem, for example, has 1,048,576 (i.e., 2^{20}) possible solutions (not all of which are feasible). A poolsize of 1,048,576 generated without duplicates would guarantee optimality. With such exhaustive enumeration, however, the genetic algorithm is superfluous.

Genetic algorithms are, in fact, based on random sampling. A larger pool is more likely to contain a better representation of the entire solution space than a smaller one. Thus with a larger poolsize a genetic algorithm is more likely to find good solutions since the genetic algorithm can perform a more informed search. However, with larger poolsizes a genetic algorithm tends to converge more slowly, thus requiring a larger number of iterations in order to insure that a good solution is found. Goldberg [1989a] suggests that a poolsize on the order of 100 is typically sufficient for a solution space in the billions.

Generation of the initial pool can require that all solutions be different, i.e., generation without duplicates, or can simply generate solutions randomly without regard to duplicates. Furthermore, the initial pool can be "seeded" with solutions to insure a wide set of genetic material. Interestingly, the initial pool often does not contain good solutions. In our experiments, for example, the cost of the best solution in the initial pool was typically at least an order of magnitude higher than that of the final solution selected.

2.3 Selecting Parents, Fitness, and Scaling

Genetic algorithms select some number of solutions to be parents in each generation. Selection of parents can be *random or probabilistic* and can be done *with or without replacement.* In a random selection method each solution in the pool has an equal probability of being selected. In a probabilistic (or *stochastic*) selection method the probability of a solution being selected is based on the solution's *fitness*, where fitness is a measure of the solution's conformance to the optimization criteria.

Selection with replacement means that a solution can be selected to be a parent any number of times in a generation. Selection without replacement means that a given solution can be selected to be a parent at most once in any generation. Hence, in selection with replacement all solutions retain the same selection probability for the selection of all parents in any generation. In selection without replacement, a solution is given a selection probability of zero after it is selected and the selection probabilities of all unselected solutions are adjusted accordingly.

In the data allocation example, two solutions are selected probabilistically without replacement. The optimization criteria is cost minimization. Hence, the fitness function is inversely related to cost. The probability of selecting a solution is proportional to its fitness (i.e., its fitness divided by the sum of the fitness over all solutions in the pool). The selection probabilities are adjusted when selecting the second parent to account for the removal of the solution selected to be the first parent.

How fitness is defined and scaled can have significant effects on the performance of a genetic algorithm. In the above example, fitness ranges from .86 to .95 (see Figure 1) yielding only a .090 difference between the probability of selecting most fit solution and the probability of selecting the least fit solution. As a genetic algorithm proceeds, the absolute range of fitness, and therefore the range of selection probabilities becomes even smaller. The fitness measure may not adequately distinguish "good" from "poor" solutions. This can reduce the pressure toward selecting the better solutions as parents, causing the algorithm to stagnate. One solution to this problem is to dynamically scale fitness and selection probability as the search progresses.

For example, a constant equal to the fitness of the worst solution could ar-

bitrarily be subtracted from the fitness of each solution to provide a wider variation in fitness and selection probabilities. Alternatively, fitness can be scaled based on the rank of each solution in the pool [Davis, 1991; Whitley, 1988].

Scaling can increase the likelihood of selecting relatively more fit solutions to be parents, thus avoiding the stagnation that can occur when less fit solutions are selected. However, there is the danger of prematurely losing genetic material from those less fit solutions and possibly settling on a locally optimal solution (i.e., converge too quickly).

2.4 Generating Children

Children are generated from selected parents using two types of operators, crossover and mutation. Crossover leads to a structured, yet randomized exchange of genetic material between parents, with the possibility of generating "better" offspring. Traditionally crossover determines the point(s) at which parent genes are split to form offspring.

Simple crossover has a single crossover point. Genes preceding the crossover point are taken from one parent, genes after the crossover point are taken from the other parent (i.e., a crossover point is between two genes). In general, crossover may have more than one crossover point. Genes are taken from alternating parents between crossover points. *Uniform* crossover [Syswerda, 1989; Ackley, 1987], as discussed above, randomly selects the parent for each gene position, i.e., the probability that its value is taken from one parent is .5; the probability that it is taken from the other parent is also .5. Crossover may generate one or two children. The second child (if generated) has the opposite parent from the first for all crossover points.

Depending on the method used to select solutions to survive into the next generation, it may be desirable to reproduce a single parent (or both parents) rather than to apply crossover to produce children. If, for example, as discussed below, all parents are replaced by children in each generation, it may be desirable to retain a portion of parents by completely reproducing some of them in their children. *Crossover rate* is the probability that crossover is used to produce children from parents. A crossover rate of 1

always uses crossover to produce children. A crossover rate of 0 always reproduces parents. Aside from the random variation introduced by mutation, a genetic algorithm with a crossover rate of 0 would never produce any new solutions, and hence, would not be an effective search method. Typically crossover rates are above 0.6 [Goldberg, 1989a].

In order to assure a wide search of the solution space, genetic algorithms typically include the concept of mutation. Mutation generates a new solution by independently modifying one or more gene values of an existing solution, selected at random. It serves to guarantee that the probability of searching a particular subspace of the solution space is never zero.

Mutation rate is the probability of applying mutation rather than crossover when a solution is selected for a genetic operation. *Mutation probability* is the probability of each gene of an individual being mutated when it is selected for mutation [Davis, 1991]. These two parameters combined reflect the probability of random variation in offspring allowing for a wide search of the solution space. Mutation probabilities are typically quite low. Goldberg [1989a] suggests mutation probabilities on the order of .001. When the mutation rates and probabilities are very high, generic algorithms approach random search.

2.5 Selecting Solutions for the Next Generation

A new generation is formed by adding all children generated by genetic operators to the pool and then removing solutions to keep the poolsize constant. The number of children generated in a generation can vary from 1 to poolsize. For example, SGA [Goldberg, 1989a] generate poolsize children, while $\Re_1$ [Holland, 1975] and GENITOR [Whitley 1988] generate one or two. Once children are generated, solutions to be removed from the pool can either include or exclude the children. Solutions can be removed either at random or based on their fitness (e.g., only the fittest survive). As discussed above, when the entire pool is replaced by children (i.e., no parents survive into the next generation), it may be desirable to have a crossover rate less than 1 so that some proportion of parents are reproduced in their children, thus preserving at least some parent gene structures.

2.6 Stopping Conditions

The most common stopping condition for genetic algorithms is a maximum number of iterations (generations). Other possible stopping conditions include: (1) limits on the number of generations with no improvement in the best solution or no improvement in the total fitness of the pool and (2) limits on the difference between the fitness of the worst and best solutions in the pool.

The next section describes and characterizes our nested genetic algorithm for distributed database design. The following section experimentally analyzes the effects of varying selected control parameters on the solutions obtained.

3. A NESTED GENETIC ALGORITHM FOR DISTRIBUTED DATABASE DESIGN

3.1 Distributed Database Design

In general, distributed database design involves three steps [Cornell and Yu,1989; March and Rho, 1995]. First, the data is partitioned into a set of file fragments for allocation. File fragments are typically defined based on the selection and projection criteria of the set of known queries [Apers, 1988]. Second, each query is decomposed into a set of query steps or operations, each of which references at most two file fragments [Cornell and Yu, 1989]. Query steps include communication steps (i.e., sending messages and result files) as well as data retrieval and processing steps (i.e., select, project, join, union, and final processing). Third, data (file fragments) and operations (query steps) are allocated to nodes to minimize a cost function, subject to resource and intrinsic constraints. The cost function should represent all appropriate system operating costs, including communication, disk I / O, CPU processing, and data storage. Resource constraints should include disk I / O, CPU processing, and storage capacities at each node and the communication capacity of each link. Intrinsic constraints require that all data be stored at least one node and that data is retrieved from a node only if it is stored at that node.

3.2 A Genetic Algorithm for Distributed Database Design

As discussed in Section 1, data and operation allocation are interrelated problems, each of which is NP-hard. To address the tractability problem, we developed a nested genetic algorithm [March and Rho, 1995]. A genetic algorithm was chosen for several reasons. First, genetic algorithms have been successfully applied to similar complex, combinatoric, real-world problems such as communication network design [Coombs and Davis, 1989; Davis and Coombs, 1989], job shop scheduling [Uckun et al., 1993], facility layout design [Tam, 1992], rule induction [Chung and Silver, 1992], and VLSI cell placement [Shahookar and Mazumder, 1991].

Second, genetic algorithms are robust in that they work well even in discontinuous, multimodal, noisy search spaces [Goldberg, 1989a]. Therefore, more realistic cost models can be used in distributed database design models. For example, very few models include the costs of queueing delays in the network or in the nodes. Genetic algorithm based solution methods can easily incorporate such costs.

Third, genetic algorithms result not only in a "best" solution, but also in a pool of good solutions. This last point is important since the set of solutions in the final pool provides significant intuition into the effects of design alternatives. That is, solutions represent "good" schemata (partial solutions) that the designer should be able to recognize from the final pool. For example, if all solutions in the final pool store a given file at a particular node, the designer would be reasonably confident that it is important to store that file at that node.

Our distributed database design algorithm contains a genetic algorithm within a genetic algorithm. The outer genetic algorithm addresses data allocation. The inner genetic algorithm addresses operation allocation. A nested approach is advantageous over a standard approach, because with a nested approach the dependency between data allocation and operation allocation can be handled relatively easily. As discussed before, the feasibility of an operation allocation is dependent on the data allocation. That is, each retrieval operation must be allocated to a node containing the required data. It is very difficult to enforce this type of constraint with a standard approach.

Furthermore, a nested approach allows us to easily incorporate different

operation allocation models. Such flexibility is desirable in distributed database design, since different distributed database management systems utilize different operation allocation models (i.e., query optimization models).

Our nested genetic algorithm operates as follows:

1. Generate initial pool of solutions:
   1.a. Randomly generate a feasible data allocation (to be feasible, each file (fragment) must be allocated to at least one node),
   1.b. Use the (nested) operation allocation genetic algorithm (see below) to allocate operations for this data allocation, thus producing a complete solution for this data allocation,
   1.c. Evaluate the cost of this solution,
   1.d. Repeat until the initial solution pool is generated.
2. Iterate through successive generations:
   2.a. Probabilistically select two parent solutions from the solution pool,
   2.b. Produce a new data allocation (child) by applying crossover or mutation,
   2.c. Use the (nested) operation allocation genetic algorithm (see below) to allocate operations for this data allocation (child), thus producing a complete solution for this data allocation,
   2.d. Evaluate the cost of this solution,
   2.e. If the new solution is better than the worst solution in the solution pool, add it to the pool and remove the worst solution,
   2.f. Repeat for N generations, where N is a maximum number of iterations.

The genetic algorithm to allocate operations for a given data allocation, used in steps 1. b. and 2. c. , is similar:

3. Generate initial pool of operation allocations:
   3.a. Randomly generate a feasible operation allocation for the given data allocation (to be feasible all retrieval operations must be assigned to nodes at which the required data is stored),
   3.b. Evaluate the cost of this solution,
   3.c. Repeat until the initial operation allocation pool is generated.
4. Iterate through successive generations:
   4.a. Probabilistically select two parent solutions from the operation all-

ocation pool,

4.b. Produce a new operation allocation (child) by applying crossover or mutation,

4.c. Evaluate the cost of this solution,

4.d. If the new solution is better than the worst in the operation allocation pool, add it and remove the worst,

4.e. Repeat for M generations, where M is a maximum number of iterations.

3.3 Characterization of the Genetic Algorithm

Table I summarizes the characteristics of our genetic algorithm and compares it with the simple genetic algorithm (SGA) in Goldberg [1989a]. The cost model and constraints used in the genetic algorithm are summarized in Appendices 1 and 2.

〈Table I〉 Characterization of a Nested Genetic Algorithm

Characteristics	Simple GA [Goldberg, 1989]	Data Allocation (Outer) GA	Operation Allocation (Inner) GA
Gene structure	binary	binary w / constraints	integer w / constraints
Poolsize	parameter	parameter	parameter
Initial pool generation	random	random*	random
Parent selection	stochastic w / o replacement	stochastic w / o replacement	stochastic w / o replacement
Fitness Function**	constant-cost	1-cost / constant	1-cost / constant
Scaling	none	none	none
Crossover	single point	uniform	uniform
Crossover rate	parameter	1	1
# of offspring / crossover	2	1	1
Mutation rate / probability	parameter	parameter	parameter
# of offspring / generation	poolsize	1	1
Next generation selection	pool replaced w / offspring	best among pool and offspring	best among pool and offspring
Stopping condition	max # of generation	max # of generation	max # of generation

* Two solutions are NOT generated randomly

** For minimization problems

The gene structure contains two sections, one for data allocation and one for operation allocation. In the data allocation stage, file fragments are allocated to nodes in the network. In the cost model, summarized in Appendix 1, the decision variable X_{ij} represents the data allocation. It has a value of 1 if file fragment i is allocated to node j. It has a value of 0 otherwise. In the gene structure for the genetic algorithm, as discussed above, each file fragment is represented by a set of n bits, where n is the number of nodes in the network. Hence, the gene structure for data allocation is simply a bit structure representing the decision variables X_{ij} in file fragment order.

In the operation allocation stage, operations are allocated to nodes. Retrieval operations must be allocated to nodes containing the required data. Join operations can be allocated to any node, however, if the data needed for the join operation is not stored at the node, it must be communicated to the join node. The decision variables Z_{kit} and Y_{kmt} are used to represent the operation allocation in the cost model (Appendix 1). These are represented in the gene structure by a set of s integers where s is the number of query steps for all queries. The integer value in each position is the node to which the operation is allocated. Hence, a complete solution for a five file (fragment), four node problem with 10 query steps would look as follows:

1111 1101 1010 0100 0100 3 2 2 3 3 1 1 4 3 4

The data allocation is represented by five sets of four bits (file 1 is stored at all four nodes; file 2 at nodes 1, 2, and 4; file 3 at nodes 1 and 3; file 4 at node 2; file 5 at node 2). The intrinsic constraint that all files must be allocated to some node is enforced by disallowing solutions in which all bits for a file are 0. The operation allocation is represented by 10 integers representing the node at which the operation is performed. Thus, operation 1 is performed at node 3, operation 2 at node 2, etc. The intrinsic constraints requiring a file to be accessed only from nodes at which it is stored need only to be enforced when the initial pool is produced and when mutation is performed.

Poolsize is a parameter for each of our genetic algorithms and is analyzed in our experiments. The initial pool is randomly generated and allows the possibility of generating duplicates in the initial pool for both data and oper-

ation allocation. However, it also seeds the initial pool for data allocation with a solution in which all files are stored at each node and with another in which each file is stored at only one node (to guarantee that these extreme solutions are considered).

Parents are selected probabilistically (i.e., stochastic selection without replacement) in each genetic algorithm. Fitness is calculated as:

Fitness = (1 - Solution Cost / Constant).

The solution cost is a complex function including communication, disk I/O, CPU processing, and data storage, as illustrated in Appendix 1. The selection probability is calculated for each solution as the solution's fitness divided by the total fitness of the pool. Fitness scaling techniques are not used.

Each algorithm produces one child per generation using uniform crossover with a crossover rate of 1 (two-point crossover is also analyzed in our experiments). The mutation rate and mutation probability are parameters (they were set at 0.01 throughout the experiments). As only one child is produced in each generation (a so called, *steady-state approach*) only one solution needs to be removed to keep the poolsize constant. Each algorithm removes the least fit solution including the newly produced child (if multiple solutions tie for the worst fitness, one of them is selected arbitrarily).

The stopping condition employed is a maximum number of iterations (generations). Unless otherwise stated, the maximum number of iterations for the outer algorithm was set at 3,000 while that for the inner algorithm at 5,000.

The genetic algorithm is written in C++ and runs in a MS-DOS or UNIX environment. To demonstrate the feasibility of this approach and its effectiveness we solved a series of 13 small problems (3 nodes, 3 file fragments and 12 to 18 query steps) with a pool size of 40. We then compared the results to the optimal solution obtained by exhaustive enumeration. The genetic algorithm found the optimal solution for all 13 problems. The run time for the genetic algorithm was four to eight minutes on an IBM-compatible PC with a 33Mhz 80386 processor.

4. EXPERIMENTAL ANALYSIS

It is important to find good control parameters for a genetic algorithm as a genetic algorithm is applied to a new domain [Davis, 1991]. In this section, we experimentally analyze the effects of poolsize and crossover operator on our nested genetic algorithm. Although such analyses can not provide the "best" parameters for a genetic algorithm in a domain, we can gain some insights into what constitutes "good" control parameters.

This study extends prior studies in several ways. First, prior studies that analyzed the effects of control parameters in genetic algorithms were limited in domain. Most studies are based on the standard De Jong test suite [De Jong, 1975]. Most functions in the prior studies involve continuous variables (i.e., real numbers) and these variables are encoded as bits. Therefore, the empirical results from the prior studies may not hold for combinatoric, integer optimization problems, specifically distributed database design problems. Furthermore, none of the prior studies analyzed the effects of control parameters on nested genetic algorithms.

Second, prior studies were limited in methodology [De Jong, 1975; Grefenstette, 1986; Uckun, et al., 1993]. First of all, very few prior studies used statistical methods to analyze their results. They relied on graphical presentations. Although graphs are useful and intuitive, they may lead to unwarranted conclusions, especially in the analyses of randomized algorithms such as genetic algorithms. Furthermore, prior studies analyzed control parameters independently. They fail to include interaction effects among control parameters. This study employs a factorial design to detect the interaction effects between some of the control parameters and uses Analysis of Variance (ANOVA), a standard statistical method, to analyze the experimental results.

The distributed database design problem used in the experiments is a variation of that described in [March and Rho, 1995]. It has 4 nodes, 9 file fragments, 93 retrieval query steps, and 24 update query steps. This results in excess of 10^{39} possible solutions (not all of which are feasible). The experiments were conducted on a UNIX workstation.

We conducted three experiments. The first investigated the relationship between poolsize, number of iterations, and crossover operator (two-point

vs. uniform) of the inner algorithm on its performance. The second experiment analyzed how poolsize of the outer algorithm and that of the inner algorithm affect the performance of the algorithm. The last experiment investigated how crossover operator of the outer algorithm affect its performance.

4.1 Experiment 1: Effects of Poolsize, Number of Iterations, and Crossover Operator

Previous studies indicate that a larger poolsize leads to better solutions (closer to optimal) [De Jong, 1975], that uniform crossover outperforms two-point crossover in some cases while the opposite is true in other cases [Syswerda, 1989; Davis, 1991], and that uniform crossover is more suitable for small poolsize [Srinivas and Patnaik, 1994].

Few studies have analyzed the interaction effects among poolsize, number of iterations, and crossover operator. We do so by employing a 5x2x2 factorial design with repeated measures on number of iterations (see Appendix 3). We vary poolsize from 50 to 500 (5 levels) and employ two types of crossover: two-point and uniform (2 levels). For each level of poolsize and crossover, we performed 3 runs. We kept the number of iterations constant at 15,000 and recorded the minimum cost for each run at 5,000 and 15,000 iterations (2 levels).

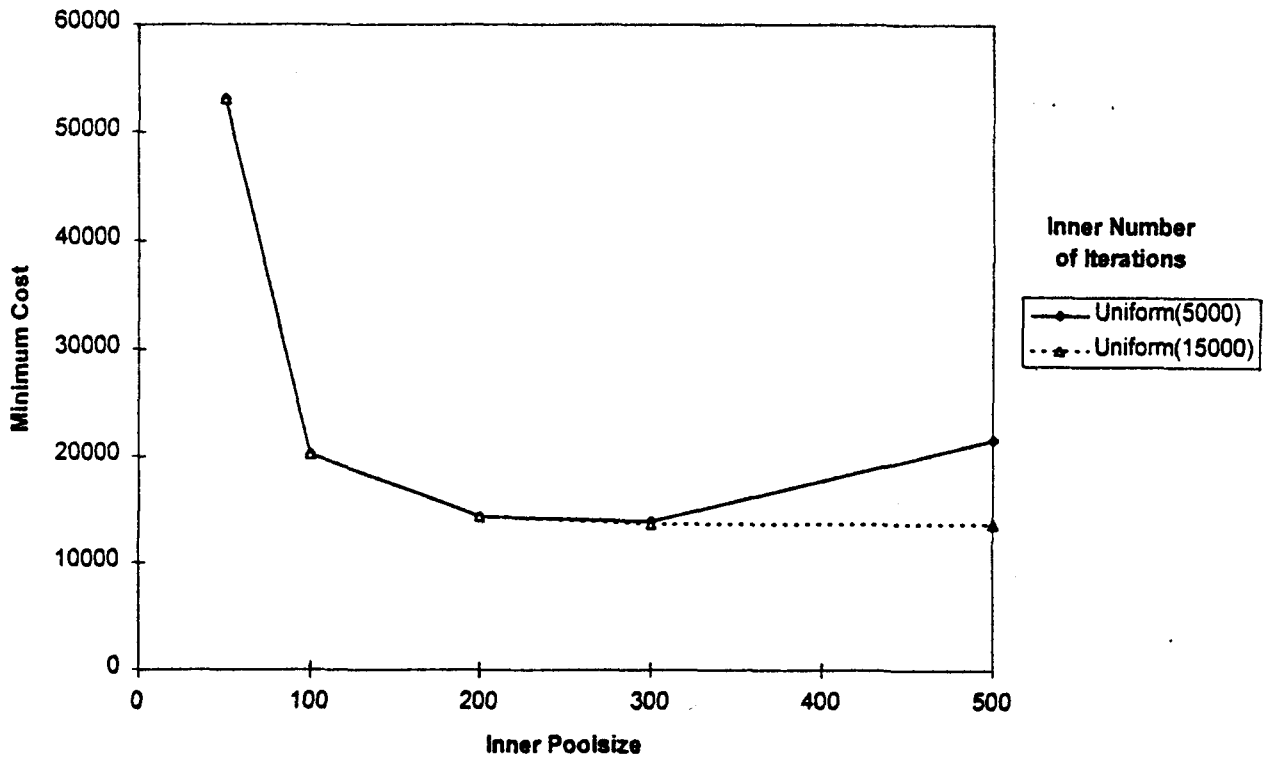
As Figure 2 and Table II indicate, poolsize has significant effects on the goodness (i.e., cost) of the solution found ($p = 0.000$). For uniform crossover, as shown in Figure 2. a, the cost of the best solution decreases dramatically as the poolsize increases. The cost levels off when the poolsize becomes sufficiently large (around 200). This is reasonable since with a small poolsize there is insufficient genetic material to adequately represent the solution space and the algorithm converges to locally optimal solutions. As more genetic material is added, more of the solution space is made accessible and better solutions are found. Eventually, sufficient genetic material is available to enable the algorithm to locate very good solutions (optimal or very close to it). At this point, the pool is "saturated" with genetic material and additional increments to the poolsize do not result in improved solutions.

As illustrated in the solid curve in Figure 2. a, however, the cost of the best solution increases when the poolsize becomes very large (i.e., 500) and the number of iterations is held constant at 5,000. Although this may seem counter-intuitive, it is not unexpected. A larger poolsize leads to slower convergence, i.e., requires more iterations to converge. Hence, a very large poolsize is likely to lead to poorer performance unless the number of iterations is increased (since it likely has not yet converged). Analysis of the costs at 15,000 iterations (the dotted curve in Figure 2.a) supports this contention. The minimum cost solution of a very large poolsize (i.e., 500) levels off instead of going up when the number of iterations is increased to 15,000. Similar results were obtained for two-point crossover (Figure 2.b).

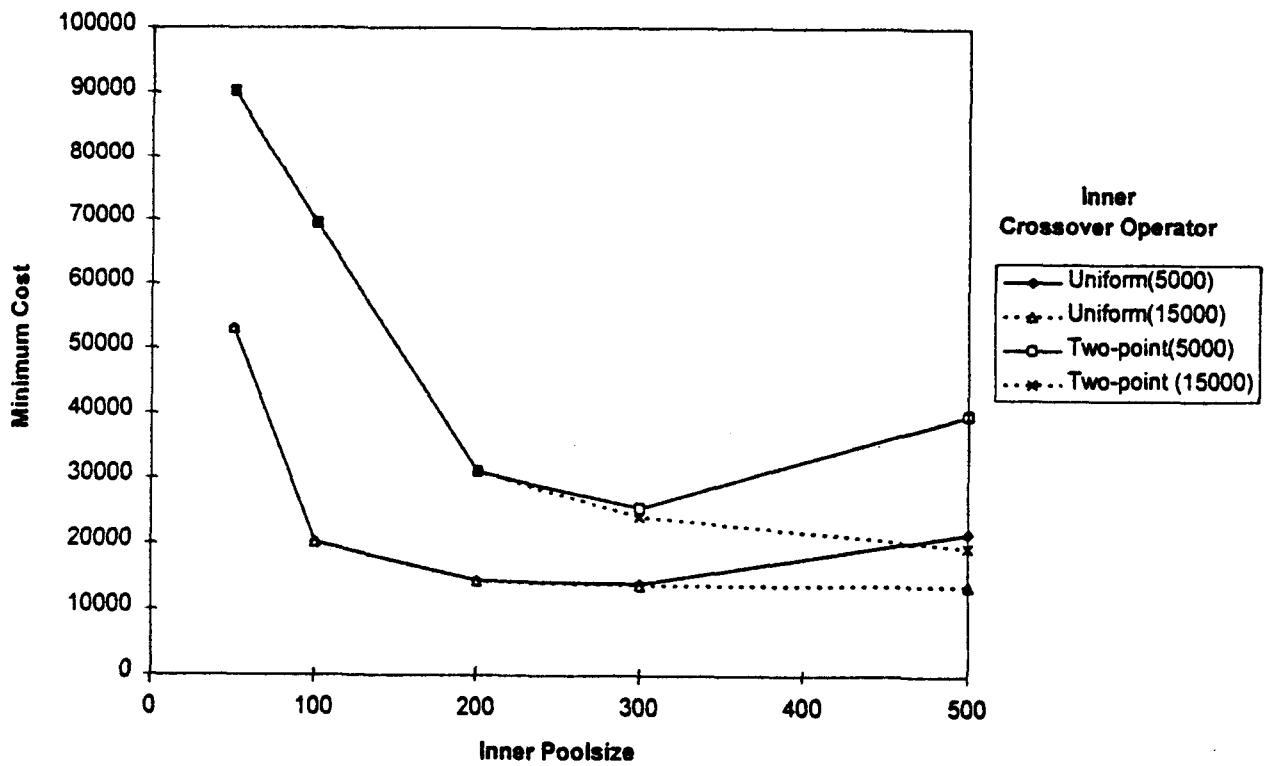
〈Table Ⅱ〉 ANOVA: Effects of Poolsize, Crossover Operator, and Number of Iterations of the Inner Algorithm on its Minimum Cost

Source of Variation	Sum of Squares(SS)	Degrees of Freedom(DF)	Mean of Squares(MS)	F*	p
Between Subjects:					
Poolsize(P)	13.810	4	3.453	19.13	0.000
Crossover(C)	7.570	1	7.570	41.94	0.000
PXC	1.260	4	0.315	1.75	0.179
Error Between	3.610	20	0.181		
Within Subject:					
Iterations(I)	0.240	1	0.240	53.33	0.000
PXI	0.840	4	0.210	46.67	0.000
CXI	0.020	1	0.020	4.44	0.076
PXCXI	0.050	4	0.013	2.78	0.056
Error Within	0.090	20	0.005		

〈Figure 2.a〉 Effects of Poolsize and Number of Iterations of the Inner Algorithm on its Minimum Cost(Uniform Crossover)



〈Figure 2.b〉 Effects of Crossover Operator of the Inner Algorithm on its Minimum Cost



The inner algorithm performed significantly better with uniform crossover than with two-point crossover across all levels of poolsize ($p = .000$), as illustrated in Figure 2.b and Table Ⅱ. That is, uniform crossover outperformed two-point crossover even when poolsize is large, contrary to the results described by Srinivas and Patnaik [1994]. There are no interaction effects between crossover operators and poolsize ($p = 0.179$).

Poorer performance by two-point crossover can be attributed to its less explorative nature. While two-point crossover can preserve more schemata than uniform crossover, it cannot obtain certain combinations of schemata in parents [Davis, 1991; Syswerda, 1989]. In our algorithm, or generally in steady state approach algorithms, parents survive into the next generation unless they are the worst solutions. Therefore, virtually all the schemata in the parents are preserved. In such an approach, two-point crossover may not be explorative enough to search significant portions of the solution space.

Another factor that might have contributed to the poorer performance of two-point crossover is its tendency to degenerate into reproduction of a parent when the pool becomes more homogeneous. When two parents are similar and the crossover points fall so as to exchange an identical segments, offspring will be identical to one of the parents (i.e., reproduction) [Booker, 1987]. This is a lost opportunity to sample new schemata and may lead to premature loss of diversity in the pool. Suppose, for example that the following operation allocations were selected as parents:

11212 22113 22331

22131 22113 32313

If, in two-point crossover, the two crossover points are after the fifth and before the eleventh genes, then the offspring would be identical to one of its parents, i.e., the sixth through the tenth genes of each parent are the same (22113). Uniform crossover is less likely to degenerate into reproduction of a parent.

4.2 Experiment 2: Effects of Poolsize of Inner and Outer Algorithms

Experiment 2 analyzed the effects of independently varying inner and outer poolsizes. Five levels were used for the outer poolsize (10, 25, 50, 100,

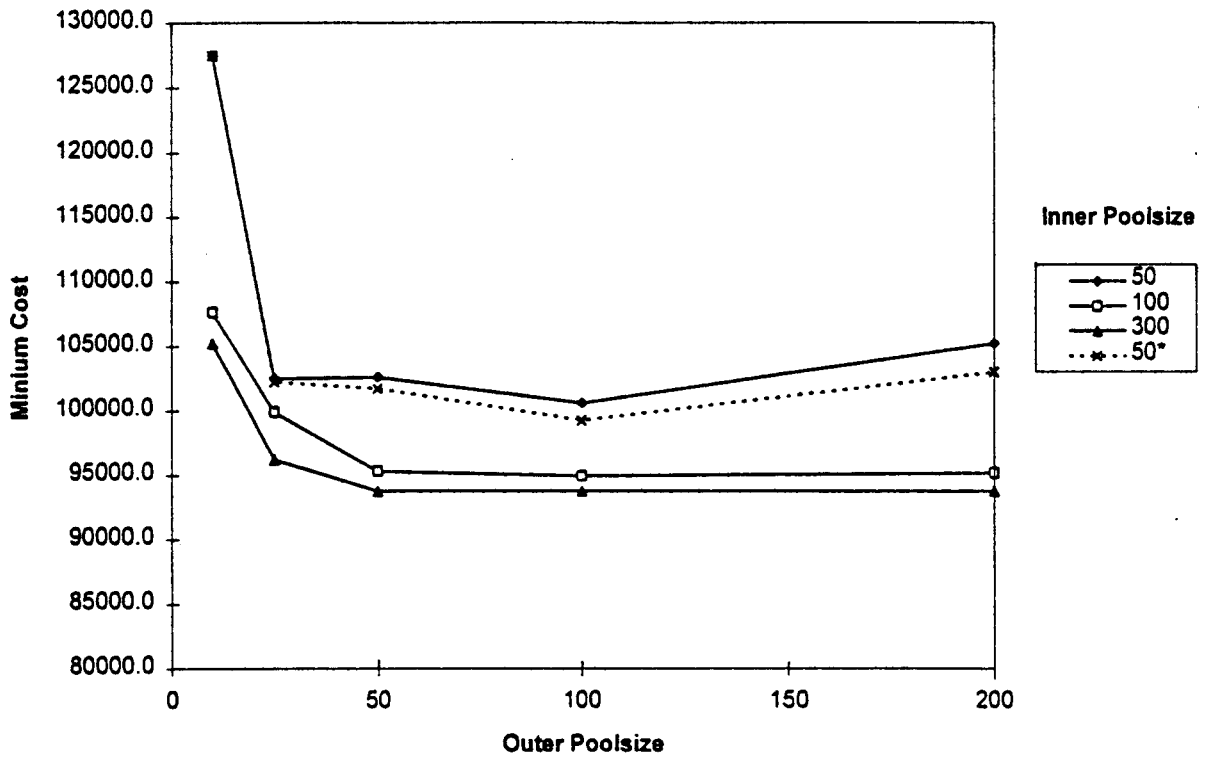
and 200); three levels were used for the inner poolsize (50, 100, and 300), which resulted in a 5×3 factorial design (See Appendix 3.b). As shown in Figure 3 and Table Ⅲ, the outer poolsize has significant effects (p = .000). The inner poolsize also has significant effects (p = .000). There are no significant interaction effects between outer and inner poolsizes (p = .393).

When the inner poolsize is small (i.e., 50), increasing the outer poolsize from 10 to 200 has inconsistent effects on the minimum cost solution (the solid curve in Figure 3., labeled 50). When the inner poolsize is larger (100 and 300), increasing the outer poolsize from 10 to 200 consistently improves (or at least maintains) the minimum cost solution. As discussed above, with a constant number of iterations, the expected results for a given inner poolsize are a decrease in the minimum cost solution until the poolsize becomes too large for the number of iterations, after which the minimum cost solution should increase.

〈Table Ⅲ〉 ANOVA: Effects of Poolsize on the Minimum Cost

Source of Variation	Sum of Squares(SS)	Degrees of Freedom(DF)	Mean of Squares(MS)	F*	p
Outer Poolsize(O)	4124.990	4	1031.248	13.96	0.000
Inner Poolsize(I)	2524.160	2	1262.080	17.09	0.000
OXI	647.910	8	80.989	1.10	0.393
Error	2215.610	30	73.854		
Total	9512.670	44	216.197		

〈Figure 3〉 Effects of Poolsize of the Outer and Inner Algorithms on the Minimum Cost



Results for the larger inner poolsizes (100 and 300) are consistent with the expected results, assuming that 3000 iterations is sufficient to assure convergence for an outer poolsize of 200. We would expect the minimum cost to increase for these inner poolsizes if the number of iterations is kept constant and the outer poolsize is increased significantly. In addition, as expected, larger inner poolsizes consistently find better solutions than smaller inner poolsizes for a given outer poolsize. As discussed earlier, further increasing the inner poolsize is likely to have detrimental effects (assuming the number of iterations is kept at 5000).

The results for the small inner poolsize are explained as follows. With a small inner poolsize, the inner algorithm will quickly converge, likely on suboptimal operation allocations. The outer algorithm may generate a very good (or even the optimal) data allocation, but the inner algorithm may not find a good operation allocation for it. As a result, this data allocation may not survive and its good schema may be lost. Hence, a small inner poolsize can mislead the outer algorithm in its search for good data allocations. When the outer poolsize is also small, these effects are exacerbated as the outer algorithm is not likely to generate many good data allocations. As the outer

poolsize grows, more of the data allocation space is searched, making it more likely that the outer algorithm will generate good data allocations. Even with inconsistent operation allocations, the effects of a wider search of the data allocation space are significant when the outer poolsize is small (i. e., 10 to 25). However, when the outer pool grows (50 to 100) the effects of a wider search of the data allocation space cannot compensate for inconsistent operation allocations.

The increase in the minimum cost when the outer poolsize is 200 is largely due to slow convergence as discussed above. When the number of iterations of the outer algorithm was increased from 3000 to 5000 the minimum cost was equal to that obtained with an outer poolsize of 100 (within 1.4%).

Interestingly, with a small inner poolsize, the outer poolsize also affected the ability of the inner algorithm to find good operation allocations. To illustrate this point, minimal cost operation allocations were determined for the data allocations selected with an inner poolsize of 50 (i.e., the selected data allocation for each of the trials with an inner poolsize of 50) for each of the outer poolsizes. These are shown in the dotted curve in Figure 3 (labeled 50*). When the outer poolsize is small (10 and 25), the minimal cost operation allocation for the given data allocation was found. However, as the outer poolsize grows, the selected operation allocation becomes worse (the difference between the solid curve labeled 50 and the dotted curve labeled 50* increases). This is explained as follows. When the outer poolsize is small, the algorithm quickly converges. Hence the outer pool contains a large proportion of identical or at least very similar data allocations. Thus it is likely that the same data allocation is given to the inner algorithm many times. Repeated execution of the operation allocation algorithm for the same data allocation increases the likelihood of finding a very good, if not optimal, operation allocation for that data allocation. As the outer poolsize grows it is less likely that operation allocation algorithm is executed for the same data allocation, thus lowering the likelihood of finding a good or near optimal operation allocation for it.

In conclusion, an inadequate inner poolsize can cause difficulties for the outer algorithm as well as for the inner algorithm. This suggests that it is crucial to optimize control parameters of the inner algorithm before optimizing those of the outer algorithm.

4.3 Experiment 3: Effects of Crossover Operator

Experiment 3 compared uniform crossover to two-point crossover for the outer algorithm. Although we compared uniform and two-point crossover for the inner algorithm in Experiment 1, there are several differences between the outer and inner algorithms that warrant additional experimentation. First of all, the gene structures for the two algorithms are different. The outer algorithm uses a set of bits while the inner algorithm uses a set of integers. Second, for the outer algorithm, the degeneration of crossover into reproduction is not necessarily bad as reproduction may improve the operation allocation for small inner poolsizes.

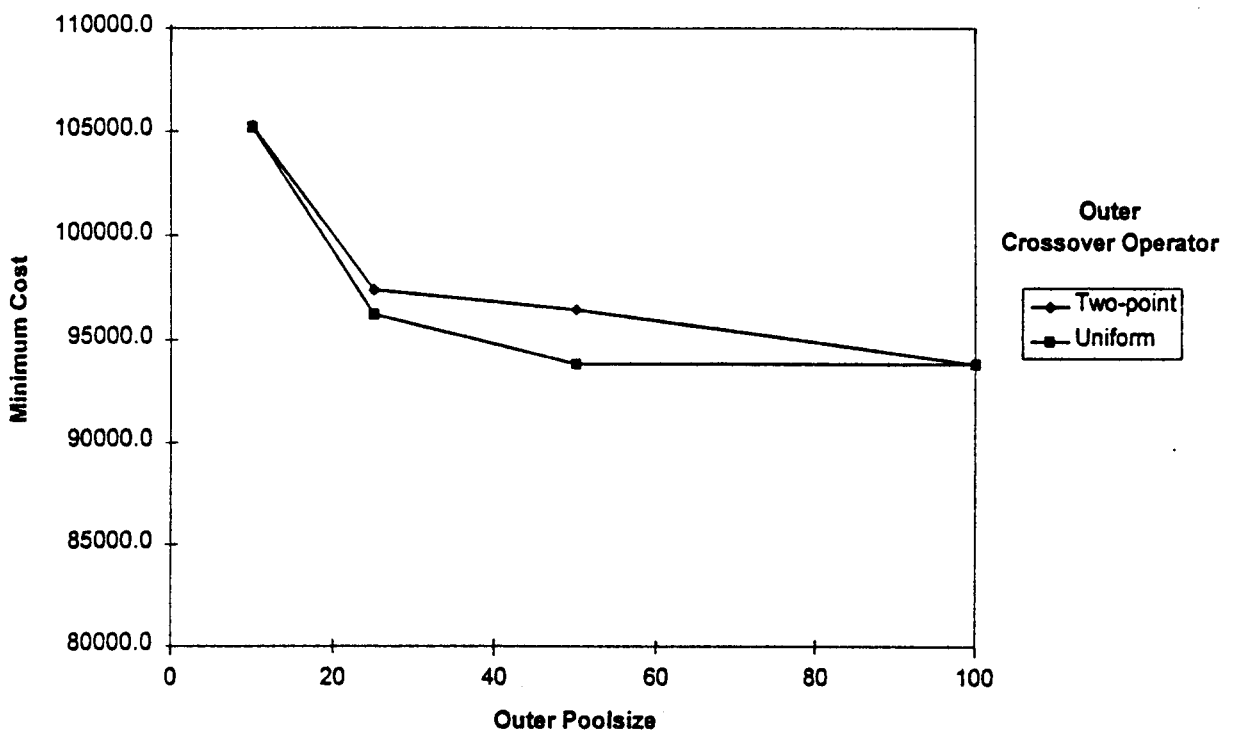
We varied the poolsize of the outer algorithm for each crossover operator from 10 to 100 (4 levels) to analyze the interaction effects, thereby resulted in 2x4 factorial design (See Appendix 3.c). For the inner algorithm, uniform crossover was used and the poolsize and number of iterations were set at 300 and 5,000, respectively.

As illustrated in Figure 4 and Table Ⅳ, although the uniform crossover operator consistently resulted in lower costs than the two-point crossover operator, the differences were not statistically significant ($p = .715$). As expected, the poolsize affects the performance significantly for both types of crossover operators ($p = .036$).

〈Table Ⅳ〉 ANOVA: Effects of Crossover Operator and Poolsize of the Outer Algorithm on the Minimum Cost

Source of Variation	Sum of Squares(SS)	Degrees of Freedom(DF)	Mean of Squares(MS)	F*	p
Crossover(C)	14.610	1	14.610	0.14	0.715
Poolsize(P)	1148.500	3	382.833	3.63	0.036
CXP	17.620	3	5.873	0.06	0.982
Error	1688.540	16	105.534		
Total		23	0.000		

〈Figure 4〉 Effects of Crossover Operator and Poolsize of the Outer Algorithm on the Mnimum Cost



5. CONCLUSION AND FUTURE RESEARCH

In this paper, we have described and analyzed a nested genetic algorithm to solve a comprehensive formulation of the distributed database design problem. The outer genetic algorithm addresses data allocation while the inner genetic algorithm addresses operation allocation.

We have described control parameters for genetic algorithms and characterized our distributed database design algorithm according to them. We have analyzed the effects of poolsize and crossover operator on the goodness of the solution found (i.e., its cost). The minimum cost of the best solution decreases dramatically as the poolsize increases until the pool becomes "saturated" with genetic material. At this point, the solution space is adequately represented and the algorithm finds a globally optimal solution (or one very close to it). Beyond this point, increases in the poolsize may have an adverse effect if the number of iterations is not large enough for the algorithm to converge. Our experiments further indicate that when the inner poolsize is inadequate or the inner algorithm is not performing well, increas-

ing the outer poolsize is not an effective way to improve the overall performance of the algorithm. Finally, uniform crossover outperformed two-point crossover in both inner and outer algorithms at all poolsizes.

Future research will progress in several direction. First, we will further analyze the effects of different parameters and fine-tune these parameters for distributed database design problems. Our goal is to understand the relationship between problem parameters (e.g., number of nodes, number of query steps) and genetic algorithm parameters (e.g., poolsize, mutation rate). We plan to analyze a number of different distributed database design problems and to vary different genetic algorithm parameters (e.g., mutation rate, fitness scaling technique). Furthermore, theoretical analysis will also be performed (e.g., Goldberg [1989b]).

Secondly, future research will compare the performance of our genetic algorithm with alternate algorithms. As mentioned above, the genetic algorithm can treat more realistic cost functions than standard optimization approaches. Furthermore, it can easily enforce constraints, a difficulty with strictly numerical approaches. However, there are several other approaches that could compete with genetic algorithms. Among them are: branch and bound algorithms, simulated annealing, switching heuristics, and randomized hill-climbing approaches.

Finally, we are considering parallelization of our genetic algorithm (See, e.g., Petty and Leuze [1989], Petty et al. [1987], Tanese [1987]). In general, genetic algorithms are conducive to parallelization when multiple offspring are produced in a given generation. Furthermore, since our algorithm is nested, additional parallelism can be achieved. We anticipate that the use of parallelism will result in orders of magnitude speedup in execution time.

REFERENCES

Ackley, D.H., "An Empirical Study of Bit Vector Function Optimization," Davis, L.(ed.), *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann, 1987, pp. 170-204.

Apers, P.M. G., "Data Allocation in Distributed Database Systems," *ACM Transactions on Database Systems*, Vol. 13, No. 3, September 1988, pp. 263-304.

Blankinship, R., Hevner, A. R., and Yao, S. B., "An Iterative Method for Distributed Database Design," *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991, pp. 389-400.

Booker, L., "Improving Search in Genetic Algorithms," in Davis, L.(ed.), *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann, 1987, pp. 61-73.

Chen, M. S. and Yu, P. S., "A Graph Theoretical Approach to Determine a Join Reducer Sequence in Distributed Query Processing," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 1, February 1994, pp. 152-165.

Chung, H. M. and Silver, M. S., "Rule-Based Expert Systems and Linear Models: An Empirical Comparison of Learning-By-Example Methods," *Decision Sciences*, Vol. 23, No. 3, May / June 1992, pp. 687-707.

Cornell, D. W. and Yu, P. S., "On Optimal Site Assignment for Relations in the Distributed Database Environment," *IEEE Transactions on Software Engineering*, Vol. 15, No. 8, August 1989, pp. 1004-1009.

Coombs, S. and Davis, L., "Genetic Algorithms and Communication Link Design: Constraints and Operators," *Proceedings of the 2nd International Conference on Genetic Algorithms*, 1987, pp.

Davis, L., ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

Davis, L. and Coombs, S., "Genetic Algorithms and Communication Link Design: Theoretical Considerations," *Proceedings of the 2nd International Conference on Genetic Algorithms*, 1987.

De Jong, K. A., *Analysis of the behavior of a class of genetic adaptive systems*, Ph. D. thesis, University of Michigan, 1975.

Dowdy, L. W. and Foster,D. V., "Comparative Models of the File Assignment Problem," *ACM Computing Surveys*, Vol. 14, No. 2, June 1982, pp. 287-314.

Eswaran, K. P., "Placement of Records in a File and File Allocation in a Computer Network," in *Information Processing '74*, Stockholm, 1974, pp. 304-307.

Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989a.

Goldberg, D. E., "Sizing Populations for Serial and Parallel Genetic Algorithms," *Proceedings of the 3rd International Conference on Genetic Algorithms*, June 4-7, 1989b, pp. 70-79.

Goldberg, D. E., "Genetic and Evolutionary Algorithms Come of Age," *Communications of the ACM*, Vol. 37, No. 3, March 1994, pp. 113-119.

Grefenstette, J. J., "Optimization of Control Parameters for Genetic Algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-16, No. 1, January / February 1986, pp. 122-128.

Grefenstette, J. J., Guest Editor, "Genetic Algorithms," Special issue of *IEEE Expert*, October, 1993.

Hevner, A., *The Optimization of Query Processing on Distributed Database Systems*, PhD Thesis, Purdue University, 1979.

Holland, J. H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI, 1975.

King, J. L., "Centralized versus Decentralized Computing: Organizational Considerations and Management Options," *ACM Computing Surveys*, Vol. 15, No. 4, December 1983, pp. 319-349.

Lafortune, S. and Wong, E., "A State Transition Model for Distributed Query Processing," *ACM Transactions on Database Systems*, Vol. 11, No. 3, September 1986, pp. 294-322.

Lohman, G. M., Mohan, C., Haas, L. M., Daniels, D., Lindsay, B. G., Selinger, P. G., and Wilms, P. F., "Query Processing in R*," in Kim, W. et al.(eds.) *Query Processing in Database Systems*, Spring-Verlag, Berlin, 1985, pp. 31-47.

March, S. T. and Rho, S., "Allocating Data and Operations to Nodes in Distributed Database Design," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 2, April 1995, pp. 305-317.

Martin, T. P., Lam, K. H., and Russell, J. I., "Evaluation of Site Selection Algorithms for Distributed Query Processing," *Computer Journal*, Vol. 33, No. 1, February 1990, pp. 61-70.

Ozsu, M. and Valduriez, P., *Principles of Distributed Database Systems*, Prentice-Hall, Inc., 1991.

Petty, C. B. and Leuze, M. R., "A Theoretical Investigation of a Parallel Genetic Algorithm," *Proceedings of the 3rd International Conference on Genetic Algorithms*, 1989, pp. 398-405.

Petty, C. B., Leuze, M. R., and Grefenstette, J. J., "A Parallel Genetic Algorithm," *Proceedings of the 2nd International Conference on Genetic Algorithms*, 1987, pp. 155-161.

Shahookar, K. and Mazumder, P., "VLSI Cell Placement Techniques," *ACM Computing Surveys*, Vol. 23, No. 2, June 1991, pp. 143-220.

Srinivas, M. and Patnaik, L. M., "Genetic Algorithms: A Survey," *Computer*, June 1994, pp. 17-26.

Syswerda, G., "Uniform Crossover in Genetic Algorithm," *Proceedings of the 3rd International Conference on Genetic Algorithms*, 1989, pp. 2-9.

Tam, K. Y., "Genetic Algorithms, Function Optimization, and Facility Layout Design," *European Journal of Operational Research*, Vol. 63, No. 2, December 10, 1992, pp. 322-346.

Tanese, R., "Parallel Genetic Algorithms for a Hypercube," *Proceedings of the 2nd International Conference on Genetic Algorithms*, 1987, pp. 177-183.

Thomas, G, Thompson, G. R., Chung, C. W., Barkmeyer, E., Carter, F., Templeton, M., Fox, S., and Hartmen, B., "Heterogeneous Distributed Database Systems for Production Use," *Computing Surveys*, Vol. 22, No. 3, September 1990, pp. 237-266.

Uckun, S. Bagci, S., Kawamura, K., and Miyabe, Y., "Managing Genetic Search in Job Shop Scheduling," *IEEE Expert*, October, 1993, pp. 15-25.

Whitley, D., "GENITOR: A Different Genetic Algorithm," *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, 1988.

Appendix 1.
Decision Variables and Cost Model Used in the Distributed Database Design Algorithm [March and Rho, 1995]

Decision Variables:

X_{it} = 1 if file fragment i is stored at node t
0 otherwise

Z_{kit} = 1 if query k uses file fragment i from node t
0 otherwise

Y_{kmt} = 1 if step m of query k is done at node t
0 otherwise

Cost Function:

$$\text{Min Cost} = \sum_k \sum_j f(k,j) \sum_m (COM(k,j,m)+IO(k,j,m)+CPU(k,j,m)) + \sum_t STO(t)$$

where f(k,j) is the frequency of execution of query k originating at node j per unit time, COM(k,j,m), IO(k,j,m), and CPU(k,j,m) are the respective costs of communication, disk I/O, and CPU processing time for step m of query k originating at node j, and STO(t) is the cost of storage at node t per unit time.

Communication Costs: COM(k,j,m) =

$\sum_t Z_{k,a(k,m),t} L^M c_{jt}$ for message steps of retrieval

$\sum_t \sum_p Z_{k,a(k,m),t} Y_{kmp} L_{a(k,m)} c_{tp}$ for selection and projection steps

$\sum_t \sum_p Y_{kmp} c_{tp} (Y_{k(m-3)t} L_{a(k,m)} + Y_{k(m-1)t} L_{b(k,m)})$ for combine fragment steps

$\sum_t Y_{k(m-1)t} L_{a(k,m)} c_{jt}$ for final steps

$\sum_t X_{a(k,m)t} L^M c_{jt}$ for message steps of update

where a(k,m) and b(k,m) are the file fragments referenced by step m of query k; L_i and L^M are the size of file fragment i and the size of a message, respectively; and c_{tp} is the communication cost per character from node t to p.

Disk I/O Costs: $IO(k,j,m) = \sum_t O(k,j,m,t)\, d_t$

where O(k,j,m,t) is the disk I/O load at node t due to step m of query k origination at node j and d_t is the cost per disk I/O at node t. O(k,j,m,t) for each step is defined as follows:

$$O(k,j,m,t) = \begin{cases} Y_{kmt}\, D_{kmt} + (1-Y_{kmt})\, Z_{k,a(k,m),t}\, F_{a(k,m)t} \\ \quad + Y_{kmt}(1-Z_{k,a(k,m),t})\, E_{a(k,m)t} & \text{for selection and projection steps} \\ Y_{kmt}\, D_{kmt} + (1-Y_{kmt})\, (Y_{k(m-3)t}\, F_{a(k,m)t} + Y_{k(m-1)t}\, F_{b(k,m)t}) + Y_{kmt}\, ((1-Y_{k(m-3)t}) \\ E_{a(k,m)t} + (1-Y_{k(m-1)t})\, E_{b(k,m)t}) & \text{for combine fragment steps} \\ Y_{k(m-1)t}\, F_{a(k,m)t} & \text{if } j \neq t, \text{ and} \\ (1-Y_{k(m-1)t})\, E_{a(k,m)t} & \text{if } j = t \quad \text{for final steps} \\ X_{a(k,m)t}\, D_{kmt} & \text{for update steps} \end{cases}$$

where D_{kmt} is the number of disk I/Os required to process step m of query k at node t, $F_{a(k,m)t}$ is the number of disk I/Os needed at node t to send a(k, m) from node t to another node, and $E_{a(k,m)t}$ is the number of disk I/Os required to receive and store a(k,m) at node t.

CPU Costs: $CPU(k,j,m) = \sum_t U(k,j,m,t)\, p_t$

where U(k,j,m,t) is the number of CPU processing units expended at node t for local processing and communication for step m of query k originating at node j and pt is the CPU processing cost per unit. U(k,j,m,t) for each step is defined as follows:

$$U(k,j,m,t) = \begin{cases} (1-Z_{ka(k,m)t})\, S_t & \text{if } j = t, \text{ and} \\ Z_{ka(k,m)t}\, R_t & \text{if } j \neq t \quad \text{for message steps of retrieval} \\ Y_{kmt}\, W_{kmt} + (1-Y_{kmt})\, Z_{k,a(k,m),t}\, F'_{a(k,m)t} \\ \quad + Y_{kmt}\, (1-Z_{k,a(k,m),t})\, E'_{a(k,m)t} & \text{for selection and projection steps} \\ Y_{kmt}\, W_{kmt} + (1-Y_{kmt})\, (Y_{k(m-3)t}\, F'_{a(k,m)t} + Y_{k(m-1)t}\, F'_{b(k,m)t}) + Y_{kmt}\, ((1-Y_{k(m-3)t}) \\ E'_{a(k,m)t} + (1-Y_{k(m-1)t})\, E'_{b(k,m)t}) & \text{for combine fragment steps} \\ (1-Y_{k(m-1)t})\, E'_{a(k,m)t} & \text{if } j = t, \text{ and} \\ Y_{k(m-1)t}\, F'_{a(k,m)t} & \text{if } j \neq t \quad \text{for final steps} \\ \sum_{p \neq t} X_{a(k,m)p}\, S_t & \text{if } j = t, \text{ and} \\ X_{a(k,m)t}\, R_t & \text{if } j \neq t \quad \text{for send-message steps of update} \\ \sum_{p \neq t} X_{a(k,m)p}\, R_t & \text{if } j = t, \text{ and} \\ X_{a(k,m)t}\, S_t & \text{if } j \neq t \quad \text{for receive-message steps of update} \\ X_{a(k,m)t}\, W_{kmt} & \text{for update steps} \end{cases}$$

where W_{kmt} is the number of CPU units required to process step m of query k at node t: S_t and R_t are the expected CPU units required to send and receive a message: and $F'_{a(k,m)t}$ and $E'_{a(k,m)t}$ are the number of CPU operations required to send and receive a(k,m) from and to node t, respectively.

Data Storage Costs: $STO(t) = G(t)\ s_t$

where $G(t) = \sum_i X_{it}\ L_i$ and st is the unit storage cost per unit time at node t.

Appendix 2.
Intrinsic and Resource Constraints Used in the Distributed Database Design Algorithm [March and Rho, 1995]

Intrinsic Constraints:

$\sum_i X_{it} \geq 1$ for all file fragments, i=1, 2, ... , no. of fragments (all file fragments must be stored at one or more nodes)

$Z_{kit} \leq X_{it}$ for all queries, k = 1, 2, ... , no. of queries for all file fragments, i = 1, 2, ... , no. of fragments for all nodes, t = 1, 2, ... , no. of nodes (a file fragment cannot be accessed from a node unless it is stored at that node)

$Y_{kmt}=1$ for all queries, k = 1, 2, ... , no. of queries for all steps m, m = 1, 2, ... , no. of steps for query k (all query steps must be processed at some node).

Resource Constraints:

Disk I/O Capacity Constraints

$\sum_k \sum_j f(k,j)\ O(k,j,m,t) \leq UIO(t)$ for each node, t = 1, 2, ... , number of nodes

where UIO(t) is the disk I / O capacity at node t.

CPU Capacity Constraints

$\sum_k \sum_j f(k,j) \sum_m U(k,j,m,t) \leq UCPU(t)$ for each node, t = 1, 2, ... , number of nodes

where UCPU(t) is the CPU processing capacity at node t.

Storage Capacity Constraints

$G(t) \leq US(t)$ for each node, t = 1, 2, ... , number of nodes

where US(t) is the storage capacity at node t.

Communication Link Capacity Constraints

$\sum_k \sum_j f(k,j) \sum_m H(k,j,m,t,p) \leq UL(t,p)$ for each link (t,p), t = 1, 2, ... , no of nodes: p = 1, 2, ... , no of nodes.

where H(k,j,m,t. p) is the amount of communication on the link connecting nodes t and p due to step m of query k originating at node j and UL(t,p) is the communication capacity of link from node t to p. H(k,j,m,t,p) for each step is defined as follows:

$$
H(k,j,m,t,p) = \begin{cases}
Z_{ka(k,m)p} L^M & \text{if } j = t \\
Z_{ka(k,m)t} L^M & \text{if } j = p \\
0 & \text{otherwise} \quad \text{for message steps of retrieval} \\
L_{a(k,m)}(Z_{ka(k,m)t} Y_{kmp} + Z_{ka(k,m)p} Y_{kmt}) & \text{for selection and projection steps} \\
L_{a(k,m)}(Y_{k(m-3)t} Y_{kmp} + Y_{k(m-3)p} Y_{kmt}) & \\
\quad + L_{b(k,m)}(Y_{k(m-1)t} Y_{kmp} + Y_{k(m-1)p} Y_{kmt}) & \text{for combine fragment step} \\
Y_{k(m-1)p} L_{a(k,m)} & \text{if } j = t \\
Y_{k(m-1)t} L_{a(k,m)} & \text{if } j = p \\
0 & \text{otherwise} \quad \text{for final steps} \\
X_{a(k,m)p} L^M & \text{if } j = t \\
X_{a(k,m)t} L^M & \text{if } j = p \quad \text{for both send-message steps and} \\
0 & \text{therwise} \quad \text{receive-message steps}
\end{cases}
$$

Appendix 3.
Average Minimum Cost

a. Average Minimum Cost by Poolsize, Crossover Operator, and Number of Iterations of the Inner Algorithm (n = 3)

Crossover		Two-point		Uniform		Average	
# of iterations		5000	15000	5000	15000	5000	15000
Inner Poolsize	50	90214.2	90214.2	53097.5	53097.5	71655.9	71655.9
	100	69523.1	69523.1	20235.6	20235.6	44879.4	44879.4
	200	31012.2	31012.2	14366.1	14362.6	22689.2	22687.4
	300	25421.1	24111.0	13933.7	13697.3	19677.4	18904.2
	500	39770.4	19451.2	21571.6	13689.6	30671.0	16570.4
	Average	51188.2	46862.3	24640.9	23016.5	37914.6	34939.4

b. Average Minimum Cost by Outer and Inner Poolsize (n = 3)

		Inner Poolsize			
		50	100	300	Average
Outer Poolsize	10	127512.7	107636.0	105215.8	113454.8
	25	102478.7	99872.2	96219.2	99523.3
	50	102607.9	95311.5	93801.2	97240.2
	100	100603.9	94951.5	93801.2	96452.2
	200	105274.7	95197.6	93801.2	98091.2
	Average	102741.3	96333.2	94405.7	97826.7

c. Average Minimum Cost by Crossover Operator and Poolsize of the Outer Algorithm (n = 3)

		Outer Crossover Operator		
		Two-point	Uniform	Average
Outer Poolsize	10	105284.6	105215.8	105250.2
	25	97390.1	96219.2	96804.7
	50	96453.2	93801.2	95127.2
	100	93801.2	93801.2	93801.2
	Average	98232.3	97259.4	97745.8