Parallel Genetic Algorithms with Local Search

Christopher L. HuntleyB3344
Donald E. Brown

IPC-TR-90-006
Revised: July 11, 1991

The Institute for Parallel Computation
and
The Department of Systems Engineering
Thornton Hall
University of Virginia
Charlottesville, VA 22903-2442

E-mail: deb@virginia.edu and clh1n@virginia.edu

This research was supported by the Jet Propulsion Laboratory and CSX transportation under grants 95772 and 5-38426, respectively.

Abstract

This paper presents methods of applying local search to global optimization problems. The most common approach, multistart, selects the best solution from restarts of local search from random starting points. Partitional methods augment local search with general principles concerning the location of global optima in real space, significantly improving the effectiveness of local search in function optimization problems. Standard partitional methods, however, are not directly applicable to combinatorial optimization problems. We describe a genetic algorithm, GALO, that is similar to the partitional methods, but can be applied to combinatorial problems. Empirical results are presented for a parallel implementation of GALO that show it to be effective for the quadratic assignment problem.

We seek a general method for the efficient application of local search algorithms to global optimization. In its most general sense, a local search algorithm is a method for finding a point, called a local optimizer, that is optimal with respect to its "nearest neighbors" within a well-defined feasible region. Since any global optimization method employing local search will likely explore a large number of search paths, we desire that the method be implementable on a parallel computer. In addition, we would like the method to be robust, making few assumptions about the problem beyond those made by the local search algorithm.

A common method for finding a global optimum is the multistart approach. Multistart algorithms apply local search to each of a large number of random initial feasible points and then select the best of the local optimizers. Part of the reason for multistart's popularity is its easy implementation. Multistart requires only a method for generating random starting points and a local search algorithm that iteratively improves on a feasible point. This "brute-force" method of optimization can be quite successful, particularly when the local search heuristic is good, but does not take advantage of information learned through local search.

This paper presents a genetic algorithm[1], called the Genetic Algorithm with Local Optimizers (GALO), that employs various kinds of "intelligence" to generate and select starting points for local search. Since the kinds of intelligence needed for different instances of a problem may vary, GALO has several options that can be configured for a given instance. In the last two sections, we show how GALO can be applied to the Quadratic Assignment Problem (QAP) and compare the results with multistart for several GALO configurations.

1. Local Optimizers and the Multistart Approach

Using the notation of Papadimitriou and Steiglitz[2], a minimization problem is defined as follows:

Given

F, a set of feasible points,

$c{:}\mathrm{F} \rightarrow \Re$, a real - valued cost function,

Minimize $c(f)$,

Subject to

$f \in \mathrm{F}$.

The pair (F, c) is an *instance* of the optimization problem. If the domain set F is continuous, then the problem is a *function* optimization problem. If F is discrete, then the problem is a *combinatorial* optimization problem.

Local optimizers are points that are optimal within a *neighborhood* $N \subseteq \mathrm{F}$. If $N(f)$ is a neighborhood about a point $f \in \mathrm{F}$ and $c(f) \leq c(y)$ for all $y \in N(f)$, then f is a local minimizer of the function c. Typically, neighborhoods are defined with respect to a distance function. In function optimization problems, $N(f)$ is usually defined to be all points within a fixed Euclidean distance from f (i.e., $N(f)$ is an open ball in Euclidean space). In combinatorial optimization problems, the structure of N is likely to be quite a bit more complicated because the variables may be categorical. Consider, for example, permutation problems, where F is defined to be

$$\mathrm{F} = \{\mathbf{p} \in \Im^n : \forall i, j \in \{1,\ldots,n\}, \quad p_i \in \{1,\ldots,n\} \text{ and } p_i \neq p_j\}.$$

For such problems, the actual numeric value of each variable p_i is irrelevant; what matters is the permutation the described by $\mathbf{p}$. A natural way to measure distance is the number of pairwise interchanges (i.e., swaps of a p_i with a p_j) needed to make one permutation equal another. In this case, a neighborhood about a point in F might be the points that are less than 2 pairwise interchanges distant. A pairwise interchange optimal permutation, then, is a permutation for which no swap exists that improves the value of c.

Local search algorithms are procedures that find a local optimizer given a starting point $f_0 \in F$. Usually, they proceed as shown in figure 1. Starting from the point f_0, they iteratively improve the value of f. The selection of g is a *perturbation* of f in the neighborhood N. If g is selected to maximize the improvement in each iteration, then the search is steepest descent. When g is selected at random from among the improving points in $N(f)$, the algorithm is randomized descent. If the selection of g is relaxed to allow cost increases with a probability $\Pr(g)<1$, then the algorithm is a variant of probabilistic descent. Other names for probabilistic descent include probabilistic hillclimbing[3], stochastic relaxation[4] and statistical cooling[5].

```
$f \leftarrow f_0$
while ($f$ is not a local optimizer) do
        $f \leftarrow g \in N(f)$ such that $c(g) < c(f)$
endwhile
```

Figure 1. A General Local Search Algorithm

For some problems, the variance between the local optima can be large. In such cases, finding just any local optimizer is not good enough. Rather, one often desires a near-global optimizer. Using a common analogy[6], imagine that the search space is a rocky beach. The goal is to find an object (the global optimizer) that lies under one of the rocks (local optimizers). A systematic way to ensure finding the object is to "leave no stone unturned, but turn over no stone more than once." Branch and bound and other enumerative methods employ this strategy. If the problem is very difficult, then enumerative algorithms may not be practical. A logical alternative is multistart, local search with random restarts. The multistart approach approximates the "leave no stone unturned" objective by selecting from a *set* of local optimizers. Stopping rules and other issues for multistart algorithms are discussed by Boender and Rinnooy Kan[7] and by Betro and Shoen[8].

There are at least two deficiencies in the multistart approach. First, if the starting points are sufficiently similar, the algorithm may find the same local optimizer many times, violating the second half of the adage. Second, multistart has no way of capitalizing on finding near-global optimizers, points that are very similar to the global optimizer. In a recent study[9], it was found that optimizers for the travelling salesman problem share approximately two-thirds of their arcs, emphasizing the importance of learning from near-global optimizers.

For function optimization problems, there exist methods that account for both of multistart's deficiencies. These techniques, called the partitional methods, attempt to divide the search space into *regions of attraction*, with one region for each local minimizer. A region of attraction about a local minimizer is the largest set of points such that steepest descent within the set will converge on the local minimizer. The motivation for finding these regions of attraction is that such knowledge can reduce the number of reexplored local minimizers. Since most partitional techniques identify regions of attraction by clustering points about local minimizers, they are sometimes called clustering methods of optimization[10][11].

Application of partitional methods to combinatorial problems is complicated by the discrete search space. Often, partitional algorithms rely heavily on the properties of euclidean space. Since all function optimization problems have real-valued domains, the assumptions implicit in euclidean distance are not overly strong. For combinatorial problems, however, the appropriate distance function is bound to vary from problem to problem, so the assumptions have to somewhat weaker than those made by partitional methods. Additionally, many partitional methods assume implicitly that all previously-seen local minimizers can be stored as the search progresses. For the smooth euclidean spaces searched in function optimization, this is not a bad assumption. For many combinatorial problems, however, the memory required to store all local minimizers can easily exceed physical limits.

We seek a method that resembles the partitional algorithms without making assumptions about the neighborhood structure. In particular, we do not want to presume continuity or ordering in the search space. We also want to limit the memory requirements to fit in available memory. Hence, we require selectivity in the information retained about the local optimizers. The next section describes how GALO meets these requirements.

2. GALO: An Approach for Combinatorial Optimization

2.1 Genetic Algorithms for Combinatorial Optimization

Genetic algorithms (GAs) were chosen because they have been successful for a variety of other complex learning and optimization problems[12][13][14]. Like multistart, part of the reason for their popularity is their easy implementation over a wide range of problems. Given a representation that defines feasible points, all that is needed is a *crossover* operator and a *fitness* function. A crossover operator is any function that can randomly combine any two feasible points, f_1 and f_2, to form another (randomized) feasible point, f_3, that is similar to both of its *parents*. A fitness function is any positive, real score to be maximized. (Minimization problems must be translated to appropriate maximization problems. An example of such a translation is in section 2.2, which describes GALO applied to the quadratic assignment problem.)

Like the partitional algorithms, genetic algorithms are based on a set-to-set mapping rather than a point-to-point mapping. This allows both the partitional and genetic algorithms to learn from patterns in the search space that might not be evident to a point-to-point improvement algorithm. For a GA, the sample set is a *population* P of *structures*, where each structure represents a feasible point $f \in F$. Although a structure may include information not explicitly in the problem definition, we ignore this subtle distinction between a structure and a feasible point in the discussion that follows.

A simple genetic algorithm is described in figure 2. The algorithm starts with an initial population set $P=\{P_k, k=1,...,S\}\subseteq F$, usually sampled at random from F. Then the population passes through an evolutionary loop, where each iteration is called a *generation*. First, each point is assigned a fitness based on its objective function score. Next, the breeding step selects pairs of parents (f_1, f_2) in proportion to their fitness:

$$\forall P_i \neq P_j \in P, \quad \Pr[f_1 = P_i, f_2 = P_j] = \left(1/(S-1)\right)\cdot\left(\mathit{Fitness}(P_i)\Big/\sum_k \mathit{Fitness}(P_k)\right).$$

The algorithm "crosses over" each pair, which produces an ordered set of *offspring*, $O=\{O_k, k=1,...,K\}\subseteq F$. Occasionally, the crossover operator makes a "mistake," causing a mutation. These randomly occurring mutations help the population maintain diversity. In order to keep the population size constant, the algorithm then selects the members of the next population (with size S) at random from the structures in P and O. The GA stops when a criterion, usually a limit on the number of generations, is met.

```
**Initialize**(P)
while (*stopping condition not met*) do
    for all $P_j \in P$ do **CalculateFitness**($P_j$)
    for $i \leftarrow 1$ to $K$ do
        $(f_1, f_2) \leftarrow$ **SelectParents**(P)
        $O_i \leftarrow$ **CrossOver**$(f_1, f_2)$
    endfor
    $P \leftarrow$ **SelectSurvivors**$(P \cup O)$
endwhile
```

Figure 2. A Simple Genetic Algorithm.

A genetic algorithm has a number of parameters. They include, but are not limited to, the population size (S), the number of offspring generated each generation (K), the mutation rate, and the stopping criterion used. Determining the optimal settings for such parameters is still an open research question. However, there do exist some general rules of thumb based on theoretical arguments. For example, it is desired that the population is large enough for each of the possible species to be represented in the population in proportion their average fitness. Goldberg[15] uses such an argument to develop a method for estimating an "adequate" population size for a given problem instance. Grefenstette[16] presents an empirical study of GA parameters, including population size.

2.2 GALO

Without modification, there is nothing explicit in the genetic algorithm to encourage exploring new regions. Holland's "schema theorem"[1] suggests that each species (or, more formally, each schema) contributes to an infinitely-large population in proportion to its fitness. Hence, with a large population size, a GA will likely explore each of the good search regions. However, the property also implies that each bad search region will also be explored, though with lesser probability.

Figure 3 shows GALO, the modified genetic algorithm. GALO attempts to make the search for good regions more explicit by incorporating three special features. The first feature, *prioritized local search*, is the basis for the algorithm's name. The last two features, *crowding*[17] and *sharing*[18], are optional and control the development of species in the population. Each of these features is described in the following paragraphs.

Prioritized local search refers to the selection of a a subset, L, of the offspring for local optimization. There are several options for the selection priorities. They include Best-First (BF), Worst-First (WF), and Random-First (RF) selection, but there may be many more. If BF is used, then the K' children with highest fitness are selected for improvement. Likewise, WF selects the K' children with lowest fitness. When RF is used, the algorithm selects the offspring in birth-order. After GALO selects the K' offspring, local search replaces each offspring with a local optimizer.

```
**Initialize**(P)
while (*stopping condition not met*) do
        for all $P_j \in P$, **CalculateFitness**($P_j$)
        if (*sharing*) then
                for all $P_j \in P$, **CalculateSharedFitness**($P_j$)
        for $i \leftarrow 1$ to $K$ do
                $(f_1, f_2) \leftarrow$ **SelectParents**(P)
                $O_i \leftarrow$ **CrossOver**($f_1, f_2$)
        endfor
        L $\leftarrow$ **SelectStartPoints**(O)
        for all $O_j \in L$, $O_j \leftarrow$ **LocalSearch**($O_j$)
        if (*crowding*) then
                P $\leftarrow$ **SelectWithCrowding**(P, O)
        else
                P $\leftarrow$ **SelectSurvivors**($P \cup O$)
endwhile
```

Figure 3. GALO, the Modified Genetic Algorithm

Crowding simulates the establishment of niches in the environment. In nature, each species (i.e., collection of similar organisms) fills a niche, or role, in the environment. Disturbing such niches can be catastrophic, even when the species is not very well-suited to survival in its current habitat. Elimination of certain species of plankton, for example, by a natural disaster could cause extinction for many higher life forms, including humans. The disturbance that killed the plankton might be temporary, but the result would be the same none-the-less. Crowding encourages the establishment and maintenance of niches by biasing the survival selection to structures that fill an established but vacated niche. With crowding, if a member of the population "dies," then it is replaced by the offspring to which it is most similar. More specifically, once it is determined that a member of the population will not survive into the next generation, its replacement is the least distant of the offspring that have not yet been selected. In this way, crowding maintains promising species by finding likely candidates to fill any vacated niches. This is particularly useful if a species has low fitness, but is similar to a very good species, and thus can still make a positive contribution to the search for better species.

Sharing is provided because a single good species can drive out other good species. Consider, for example, a small population of similarly-fit structures consisting of a pair of identical twins and a set of other, very distinct offspring. Recall that in a genetic algorithm each member of the population is selected for breeding in proportion to its fitness. Since an average structure is present twice in the population, it has an unfair breeding advantage that is totally unrelated to its fitness, which violates Holland's schema theorem. If the population were infinite, then the effects of such redundancies would be negligible, but the population is finite. Sharing prevents a species from dominating the gene pool by "sharing the wealth" among the members of each species. If there are two copies of a structure in the population, then sharing divides each of the two fitnesses in half. Goldberg and Richardson[18] quantified this idea by introducing a shared fitness function. First, a similarity measure, *sh*, is calculated for each pair of members in the population:

$$sh(d_{ij}) = \begin{cases} 1 - \left(d_{ij} / \sigma_{share}\right)^{\alpha}, & d_{ij} < \sigma_{share} \\ 0, & \text{otherwise} \end{cases}$$

where d_{ij} is the distance between individuals P_i and P_j in P. Then, a total measure of similarity is calculated for each structure:

$$m_i = \sum_{j \in P} sh(d_{ij}), \qquad \forall i \in P.$$

Finally, each structure's fitness is divided by its total similarity, producing a shared fitness used to calculate reproduction probabilities:

$$SharedFitness(P_i) = Fitness(P_i) / m_i.$$

This formula rewards unique members of the population by increasing their reproduction probability. The constants α and σ_{share} control how much unique structures are rewarded. For the remainder of this paper, it is assumed that $\alpha=1$ and that σ_{share} is calculated using the formula described by Deb and Golberg[19].

2.3 Parallel Implementations

Since genetic algorithms are based on population biology models, they are inherently parallel. For example, consider the crossover and fitness evaluation steps. These steps perform a majority of the work if the points have high dimension or the fitness function is complex. Since the offspring are produced independently each generation, these steps are easily made parallel. If the parallel implementation branches to perform these steps in lockstep, then the algorithm is called a synchronous master-slave GA. There are many other parallel versions of the algorithm. An early study by Grefenstette[20] proposed four commonly-used models: synchronous master-slave, asynchronous master-slave, asynchronous concurrent, and network. A later paper reported on working implementations for three of the four models on a hypercube multicomputer[21]. (The asynchronous concurrent model requires shared memory, and was not implemented.) The three parallel versions described here closely resemble those in the latter paper.

The first parallel version of GALO is a variant of Grefenstette's synchronous master-slave model. It is the simplest of the parallel implementations, for it merely performs the local search step from figure 3 in parallel. Figure 4a shows a simplified schematic of the master-slave arrangement. A master GA provides starting points for a set of slaves, each running a local optimization algorithm on its own processor. After breeding, the master GA sends each selected offspring to a slave for improvement. It then waits for each processor to finish before continuing. The method is synchronous because the master must wait for each of the slaves to finish their work. If one considers the local improvement a part of the breeding process then this is clearly similar to Grefenstette's model. There is a subtle difference, however, because not every offspring is improved.

The asynchronous master-slave model is similar to the first, with the same basic architecture (figure 4a). As before, there is a master GA and a set of *K´* local search slaves, but in the asynchronous model the master GA does not wait for the slaves to finish. Rather, it improves a variable number of offspring each generation, depending on how many processors are idle. After a slave finishes a local search, it sends a message to the master that it is ready to exchange its improved structure for a new one and waits for a reply. Once the master has generated a set of new offspring, it reads the messages sent to it by the slaves. Since execution times for the local searches may vary, it is possible that less than *K´* messages are received. The master then exchanges an offspring for each of the structures held by the slaves that sent a message. The new local optimizers returned by the slaves replace the exchanged offspring. Once all of the exchanges are complete, the master continues as usual. Hence, the asynchronous master GA behaves like a simple, sequential GA, except every so often it "trades-in" some of its offspring for new local optimizers, as they become available.

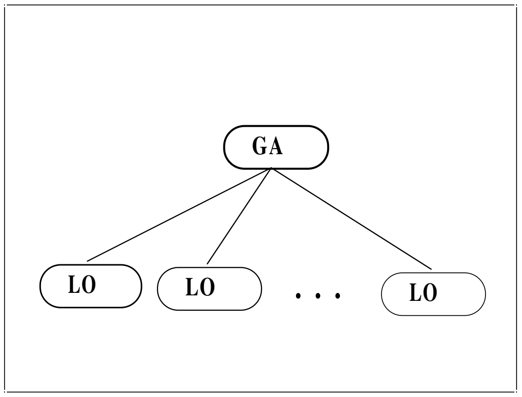

Figure 4a. The Master-Slave Model

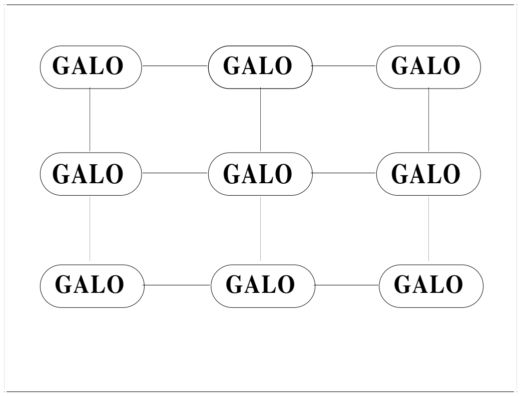

Figure 4b. The Network Model

The network model consists of a set of cooperating sequential GAs distributed over a multicomputer network. Figure 4b shows an architectural schematic for GALO on a grid-connected network. Each processor executes its own copy of sequential GALO on its own subpopulation. Occasionally, each processor exchanges information (i.e., a few of its structures) with its neighbors. If the time between exchanges is large enough, then it is likely that the subpopulations will converge on different species. The exchange step might be seen as species migration caused by a catastrophic environmental shift. Such an interpretation of Grefenstette's network model, called punctuated equilibria, is described by Cohoon et al.[22]. A fine-grained variant of GALO's network implementation is Muhlenbein's PGA[23], which maintains a single structure on each processor and uses breeding for information exchange.

Each of the models has its advantages, depending on the circumstances. If the execution times for the local searches are large and nearly-uniform, then the synchronous model is adequately efficient on relatively small (i.e., 128 nodes or smaller) hypercube multicomputers. The asynchronous master-slave model is applicable when the times are large but non-uniform. Although the network model's behavior is harder to predict, it may be the most appropriate when the number of processors is large because there is no master to act as a bottleneck.

3. GALO Applied to the Quadratic Assignment Problem

In order to demonstrate the application of GALO to a complex combinatorial optimization problem, we present an implementation for the quadratic assignment problem (QAP). First, we formally define the quadratic assignment problem, a computationally complex permutation problem. Then, we present details about a GALO implementation appropriate for the QAP.

3.1 The Quadratic Assignment Problem

The quadratic assignment problem is a classic combinatorial optimization problem. It is very simple to define an instance of the QAP — any two square, real matrices of identical dimension will do. Yet the QAP is very difficult to solve to a global optimum. In fact, the infamous travelling salesman problem (TSP) is a relatively easy special case of the QAP.

The QAP is defined as follows:
Given

$$\mathbf{C}\colon \Im^2 \to \Re, \text{ an } n\text{x}n \text{ real matrix,}$$

$$\mathbf{D}\colon \Im^2 \to \Re, \text{ an } n\text{x}n \text{ real matrix,}$$

Minimize $$\sum_{i=1}^{n}\sum_{j=1}^{n} C_{p_i p_j} D_{ij}$$

Subject To

$$p_i \in \{1,\ldots,n\}, \quad \forall i \in \{1,\ldots,n\},$$

$$p_j \neq p_i, \quad \forall i \neq j \in \{1,\ldots,n\}.$$

The pair of matrices (**C**, **D**) define an instance of a QAP. The objective is to minimize the point-serial correlation between **C** and **D** by permuting the indices into **C**. The two constraints guarantee that **p**=$\{p_i\}$ is a valid permutation of the indices $\{1,...,n\}$.

Many problems can be cast as QAPs. For example, the plant layout problem is often formulated as a QAP, where **C** is a matrix of machine-to-machine material flows and **D** is a matrix of location-to-location distances within the plant[24]. The optimal permutation **p** is an assignment of machines to locations that minimizes materials handling cost. In data analysis, **C** describes a desired structure with a set of proximities and **D** is the observed distances between a set of data points[25]. The goal is to find an optimal mapping of “structure” (**C**) onto “data” (**D**). Using this last interpretation of the QAP, the travelling salesman problem is a QAP where the desired structure is a cyclic permutation:

$$\mathbf{C} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 1 \\ 1 & 0 & \cdots & 0 & 0 \end{bmatrix}.$$

Hence, the QAP is at least as hard as the TSP. Since the TSP is NP-hard, the QAP is also NP-hard, requiring exponential solution times in the worst case.

A popular local search for the QAP is steepest-descent-pairwise-interchange[24] (SDPI), which returns a pairwise interchange optimal solution. SDPI is a direct implementation of the local search algorithm in figure 1, where f is a permutation of the integers $\{1,...,n\}$, the neighborhood $N(f)$ is all $n(n-1)/2$ interchanges of a label p_i with a label p_j, and g is the maximally improving interchange. Although SDPI is one of the simplest of the many QAP approximation algorithms, it is also one of the most effective[26]. All of the experiments described in the remaining sections use SDPI for local search.

3.2 GALO Applied to the QAP

This section describes GALO for the QAP by filling in four problem dependent details: the structures used to encode feasible points, the fitness function, the crossover and mutation operators used for breeding, and the distance measure needed for crowding and sharing.

GALO represents a permutation with an array of integers labels (e.g., "`3 1 2`"). Such a simple representation is possible because all the variables in a permutation problem are of the same type. Otherwise, additional information would be needed. If, for example, the problem were a mixed-integer linear program then the coding might not be as direct. For such problems, one might need to encode which variables are integral in the problem instance.

The fitness of an individual P_i in the population is its difference from the worst cost encountered so far (inclusive):

$$fitness(P_i) = c_{worst} - c(P_i).$$

where c_{worst} is the worst of the observed costs and $c(P_i)$ is the cost of structure P_i. Alternately, one could determine c_{worst} with multistart using steepest *ascent*, but this leaves open the possibility that an individual would have negative fitness (i.e., c_{worst} is not a global maximum), which violates the definition of fitness.

The crossover operator is a version of the partially-matched crossover (PMX) operator by Goldberg and Lingle[27]. Given two parents structures, the operator copies a substring of the labels from the first parent directly into corresponding positions in the offspring and fills in the remaining positions with the unused labels in the order they appear in the second parent. Figure 5 shows an application of the operator. First, a pair of *crossover positions* (shown as the two lines, "|") are selected at random. Next, the labels in the positions between the two lines are copied from the the first parent, (P1). Then the placement of the remaining labels is determined by their order in the second parent (P2), producing a feasible structure (Off). Finally, each label in the new string is subject to interchange with another label with probability *m*<<1. The last step of figure 5 shows such a mutated string with the two interchanged labels in **boldface**.

P1: 1 2 3 4 5 6 7 8   P2: 6 8 4 8 3 7 2 1 5

Step i.   Off: _ _ | _ _ _ | _ _ _
Step ii.   Off: _ _ | 3 4 5 | _ _ _
Step iii.   Off: 6 8 | 3 4 5 | 7 2 1
Step iv.   Off: 6 **7** 3 4 5 **8** 2 1

Figure 5. An Application of the PMX Crossover Operator

The function used by GALO to measure distance between strings is Hamming distance. A more natural distance function might be the number of pairwise interchanges needed to make one string identical to another, for such a measure is a direct consequence of the neighborhood structure in SDPI. But such a function would be impractical because of the need to find the sequence of interchanges. Hamming distance is a compromise between computational simplicity and accuracy. If two structures have “small” Hamming distance, they have even smaller interchange distance. Hence, Hamming distance is an upper bound on the interchange distance that is easy to calculate, requiring only O(n) comparisons and additions.

4.0 Empirical Results

GALO and multistart were applied to three QAP instances from the literature. Table 2 describes the test problems. For each problem, the table shows the source, the size, the best feasible solution in the literature, and a statistical lower bound from Derigs[28, pg. 1037]. Derig’s bound, based on the method by Golden[29], is lower than the true global minimum with nearly 100% confidence. More specifically, Pr[$c<c_{\text{lower}}$] = exp(-50).

Problem #	Source	Size (n)	c_{upper}	c_{lower}
1	Elshafei[30]	19	17212548	12143024
2	Nugent et al.[26]	30	6124	6057
3	Steinberg[31]	36	9526	8920

Table 2. The Test Problems.

In an effort to improve the c_{upper} bounds, we applied the simulated annealing algorithm from previous work[32] to each problem 10 times. Simulated annealing matched the bounds in each case, but failed to improve on them.

All of the results presented in the next two subsections were obtained on the 128-node Intel iPSC/860 hypercube multicomputer at Oak Ridge National Laboratories. The computer code was written in C with message passing primitives. All random numbers were generated using **ran2()**, the machine independent random number generator detailed by Press et al.[33, pg. 210].

4.1 Multistart Experiments

The intent of the multistart experiments was to explore the test problems and to provide a comparison point for GALO. SDPI was applied to each of 100,000 random starting points for each problem. Table 3 shows a sampling of the local optima, where each $c_{(i)}$ is the i-th best of the local optima found (i.e., $c_{(25,000)}$ and $c_{(75,000)}$ are the quartiles and $c_{(50,000)}$ approximates the median). Note that multistart matched c_{upper} for problem 1. It did so 214 times, or about once in every 467 trials. Multistart failed to match c_{upper} for either problem 2 or problem 3.

Problem	$c_{(1)}$	$c_{(25,000)}$	$c_{(50,000)}$	$c_{(75,000)}$	$c_{(100,000)}$
1	17212548	19788612	22198042	24197342	62087210
2	6140	6416	6508	6622	8436
3	9676	10856	11234	11710	22440

Table 3. Multistart Results.

4.2 GALO Experiments

After exploring the problems with multistart and simulated annealing, we implemented the synchronous master-slave version of GALO. The reasons for selecting the synchronous model were two-fold. First, the available hypercube was not large enough for the master GA to be a bottleneck. Second, SDPI exhibited nearly uniform run time in the multistart experiments, lessening the need for an asynchronous implementation.

We standardized all GALO test runs with a common set of parameters. Of particular concern were the sizes of P and O, the population and offspring sets. As suggested by the schema theorem, we desired a large population size, yet not so large as to allow the superlinear time complexity of crowding and sharing to dominate computation. In informal testing, the algorithm performed adequately with 300 members in the population and 150 offspring per generation. We determined the mutation rate (5% of all interchanges each generation) and the maximum number of generations (1000) similarly. In addition to the usual parameters, we added a stopping condition: the algorithm stopped if it ever matched or improved on c_{upper} in table 2. This condition was added because in these tests we were primarily concerned with establishing GALO's place among QAP algorithms. If GALO could consistently match the best answer ever found for a problem, then it must be considered competitive.

In the tests that follow, we use each of the problems in a different way. The first experiment measures GALO's improvement over multistart on the first problem, for which multi-start seems well-suited. Problem 1 is also used to observe the impact of the various GALO options on performance. The second experiment uses problem 2 to assess the impact of the number of local search slaves on performance with each of the various feature options. The last experiment measures GALO's performance on problem 3. The problem is well-known in the literature, yet largely unexplored because of its large size. (Recall that solution times for the QAP increase exponentially with problem size, so the size 36 problem is much harder than either the size 19 or the size 30 problem.)

Experiment 1:

The first test compared GALO to multistart on problem 1. Table 5 summarizes GALO's performance for 20 trials with each combination of configuration options. The table describes the GALO options used, the average CPU time per generation, and the distribution of the number of generations (G) needed to match c_{upper} from table 2. In the same notation as before, $G_{(i)}$ is the *i*-th best in the sample, so $G_{(10)}$ approximates the median of the distribution of G. From the table, it appears that it is best to use worst-first (WF) local search priority without sharing on problem 1; the corresponding lines in table 4 are shown in **boldface**. Using WF, GALO reduced the average number of local searches needed to match c_{upper} from 467 for multistart to $(K')(G_{(10)})=(15)(1)=15$. Also note that even the poorest performing GALO configuration, (BF, On, On), was better on average than multistart (75 versus 467).

LO Priority	Crowding	Sharing	K'	Time/ Gen (sec)	$G_{(1)}$	$G_{(5)}$	$G_{(10)}$	$G_{(15)}$	$G_{(20)}$
BF	Off	Off	15	0.6	1	2	3	7	19
BF	Off	On	15	1.2	1	2	5	9	20
BF	On	Off	15	0.8	1	3	5	12	28
BF	On	On	15	1.3	1	2	5	9	12
WF	**Off**	**Off**	**15**	**0.6**	**1**	**1**	**1**	**2**	**3**
WF	Off	On	15	1.2	1	1	3	4	8
WF	**On**	**Off**	**15**	**0.8**	**1**	**1**	**1**	**2**	**3**
WF	On	On	15	1.3	1	1	3	5	10
RF	Off	Off	15	0.6	1	1	3	5	19
RF	Off	On	15	1.2	1	1	2	8	12
RF	On	Off	15	0.8	1	1	3	4	8
RF	On	On	15	1.3	1	1	2	2	12

Table 4. GALO Results for Problem 1.

Experiment 2:

The second experiment investigated the interaction of the proportion of offspring improved each generation with the local search priority and speciation options. GALO was applied to the second problem 20 times for each configuration of options. The results are in tables 6a and 6b. Note that when $G_{(i)}$=1000, the algorithm "timed out" without ever matching the c_{upper} bound.

LO Priority	Crowding	Sharing	K'	Time/ Gen (sec)	$G_{(1)}$	$G_{(5)}$	$G_{(10)}$	$G_{(15)}$	$G_{(20)}$
BF	Off	Off	15	1.9	33	205	804	1000	1000
BF	Off	On	15	2.3	30	227	422	702	1000
BF	**On**	**Off**	**15**	**2.6**	**79**	**135**	**269**	**525**	**1000**
BF	**On**	**On**	**15**	**2.9**	**44**	**83**	**239**	**579**	**1000**
WF	Off	Off	15	2.2	88	578	1000	1000	1000
WF	Off	On	15	2.5	51	211	1000	1000	1000
WF	On	Off	15	2.9	19	182	389	1000	1000
WF	On	On	15	3.2	84	329	703	1000	1000
RF	**Off**	**Off**	**15**	**2.1**	**24**	**137**	**230**	**626**	**1000**
RF	Off	On	15	2.4	92	490	696	1000	1000
RF	**On**	**Off**	**15**	**2.9**	**18**	**168**	**304**	**630**	**1000**
RF	On	On	15	3.8	4	197	591	1000	1000

Table 5a. GALO Results on a 16-node Hypercube for Problem 2.

LO Priority	Crowding	Sharing	K'	Time/ Gen (sec)	$G_{(1)}$	$G_{(5)}$	$G_{(10)}$	$G_{(15)}$	$G_{(20)}$
BF	**Off**	**Off**	**63**	**2.1**	**4**	**42**	**64**	**106**	**169**
BF	Off	On	63	2.4	14	112	189	340	522
BF	**On**	**Off**	**63**	**2.4**	**10**	**45**	**78**	**142**	**659**
BF	On	On	63	3.1	20	73	121	204	447
WF	Off	Off	63	2.3	13	87	146	467	819
WF	Off	On	63	2.7	4	39	193	368	798
WF	On	Off	63	2.4	7	108	216	435	1000
WF	On	On	63	3.2	6	78	347	675	968
RF	Off	Off	63	2.2	40	104	155	302	431
RF	Off	On	63	2.9	56	125	166	325	454
RF	On	Off	63	2.5	8	66	151	277	349
RF	On	On	63	3.2	12	53	122	251	606

Table 5b. GALO Results on a 64-node Hypercube for Problem 2.

From table 5a, it appears that with a 16-node hypercube the best configuration for problem 2 is either BF with crowding or RF without sharing. As before, the corresponding lines are shown in **boldface**. These configurations have relatively small $G_{(10)}$ and at least average spread ($G_{(15)}$ - $G_{(5)}$). For the 64-node hypercube (table 5b), the best configuration is BF without sharing. The two corresponding lines show small $G_{(10)}$ and small spread. When the two tables are considered together, the most consistently good configuration is BF with crowding enabled and sharing disabled. The 64-node version of this configuration allocated its local searches almost as efficiently as the 16-node version, increasing the total number of searches by only 12%.

Experiment 3:

In the third experiment, we applied the 64-node implementation (using BF with crowding) to problem 3. As before, the stopping criterion was improvement on c_{upper}. Table 6 shows the results for 20 runs. The algorithm matched the best known cost in 75% of its trials. Multistart, by comparison, never matched c_{upper} in its 100,000 searches.

Trial	Generations	Time (sec)	Cost
1	696	2538	9526
2	578	2122	9526
3	105	402	9536
4	600	2217	9526
5	209	773	9536
6	866	3184	9536
7	101	382	9526
8	661	2451	9526
9	178	663	9526
10	322	1191	9536
11	529	1937	9526
12	124	473	9526
13	125	468	9526
14	113	436	9526
15	781	2871	9526
16	56	227	9526
17	468	1749	9536
18	696	2568	9526
19	249	958	9526
20	270	1008	9526

Table 6. GALO Results for Problem 3 on a 64-node Hypercube.

5. Conclusions

Methods that incorporate local optimization are an important class of global optimization techniques. A simple approach to the use of local optimization is the multistart strategy. Partitional algorithms use local optimization in a more sophisticated way for function optimization. Unfortunately, partitional approaches are not directly applicable to a number of hard combinatorial optimization problems. To address these problems, we developed GALO, a parallel search using a genetic algorithm to sample local optimizers. GALO represents a particularly effective approach that incorporates ideas from partitional methods for function optimization. Application to three quadratic assignment problems from the literature leads to two conclusions:

1) The most robust of the GALO settings is Best-First local optimization priority with crowding, though for small problems Worst-First with crowding may be better.

2) GALO performs well on some of the most difficult quadratic assignment problems in the literature, finding the best solutions ever obtained.

To summarize, we found that an intelligent implementation of local search can solve global optimization problems, but the nature of the intelligence may be problem dependent. So any method that employs local search should, like GALO, allow the user some flexibility in its application.

The research described in this report can be extended in several ways. The first is investigating more powerful local optimization procedures. A second, related extension is to implement the asynchronous parallel model of GALO. Toward both of these ends, we are developing an asynchronous implementation of GALO that uses simulated annealing for the local search procedure. Provided that we can devise a method for simulated annealing to take advantage of "good" starting points, the method may significantly improve on the SDPI results. An early version of the GALO/simulated annealing implementation is reported in a previous paper[32]. The third extension is to examine other problems. We are currently working on the application of GALO to the routing and scheduling of trains, a real problem that promises to be even more complex than the QAP.

7. References

1. J. Holland, 1975. *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI.
2. C. Papadimitriou and K. Steiglitz, 1982. *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ.
3. F. Romeo and A. Sangiovanni-Vincentelli, 1985. Probabilistic Hill Climbing Algorithms: Properties and Applications, In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, 393-417.
4. S. Geman and D. Geman, 1984. Stochastic Relaxation, Gibbs Distributions, and the Restoration of Images, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-6**, 721-741.
5. J. Storer, A. Nicas, and J. Becker, 1985. Uniform Circuit Placement, In P. Bertolazzi and F. Luccio, (ed.), *VLSI: Algorithms and Architectures,* Elsevier's Science Publishers, Amsterdam.
6. J. Pearl, 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA.
7. G. Boender and A. Rinnooy Kan, 1987. Bayesian Stopping Rules for Multistart Global Optimization Methods, *Mathematical Programming* **37**, 59-80.
8. B. Betro and F. Shoen, 1987. Sequential Stopping Rules for the Multistart Algorithm in Global Optimization, *Mathematical Programming* **38**, 271-286.
9. H. Muhlenbein, 1990. Parallel Genetic Algorithms and Combinatorial Optimization, Presented at *The Symposium on Parallel Optimization*, Madison, WI.
10. A. Rinnooy Kan and G. Timmer, 1987. Stochastic Global Optimization Methods. Part I: Clustering Methods, *Mathematical Programming* **39**, 27-56.

11. A. Torn and A. Zilinskas, 1989. *Global Optimization*, Springer-Verlag, New York, NY.

12. J. Grefenstette, (ed.), 1987. *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, Lawrence Erlbaum Associates, Hillsdale, NJ.

13. L. Davis, (ed.), 1987. *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann, Los Altos, CA.

14. J. Schaffer, (ed.), 1989. *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA.

15. D. Goldberg, 1985. Optimal Population Size for Binary-coded Genetic Algorithms, Research Report TCGA-85001, University of Alabama, Tuscaloosa, AL.

16. J. Grefenstette, 1986. Optimization of Control Parameters for Genetic Algorithms, *IEEE Transactions on Systems, Man, and Cybernetics* **SMC-16**, 122-128.

17. K. De Jong, 1975. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Ph.D. Thesis, University of Michigan, Ann Arbor, MI.

18. D. Goldberg and J. Richardson, 1987. Genetic Algorithms with Sharing for Multimodal Function Optimization, In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, 1-8.

19. K. Deb and D. Goldberg, 1989. An Investigation of Niche and Species Formation in Genetic Function Optimization, In *Proceedings of the Third International Conference on Genetic Algorithms*, 42-50, San Mateo, CA.

20. J. Grefenstette, 1981. Parallel Adaptive Algorithms for Function Optimization, Research Report CS-81-19, Vanderbilt University, Nashville, TN.

21. C. Pettey, M. Leuze, and J. Grefenstette, 1987. Genetic Algorithms on a Hypercube Multiprocessor, In M. Heath, (ed.), *Hypercube Multiprocessors 1987,* Siam, Philadelphia, PA.

22. J. Cohoon, S. Hegde, W. Martin, and D. Richards, 1987. Punctuated Equilibria: a Parallel Genetic Algorithm, In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, 177-183.

23. H. Muhlenbein, 1989. Parallel Genetic Algorithms, Population Genetics, and Combinatorial Optimization, In *Proceedings of the Third International Conference on Genetic Algorithms*, 416-422.

24. G. Armour and E. Buffa, 1963. A Heuristic Algorithm and Simulation Approach to the Relative Location of Facilities, *Management Science* **9**, 294-309.

25. L. Hubert and J. Schultz, 1976. Quadratic Assignment as a General Data Analysis Strategy, *British Journal of Mathematical and Statistical Psychology* **29**, 190-241.

26. C. Nugent, R. Vollmann, and J. Ruml, 1968. An Experimental Comparison of Techniques for the Assignment of Facilities to Locations, *Operations Research* **16**, 150-173.

27. D. Goldberg and R. Lingle, 1985. Alleles, Loci, and the Travelling Salesman Problem, In *Proceedings of an International Conference on Genetic Algorithms and their Applications*, 154-159.

28. U. Derigs, 1985. Using Confidence Limits for the Global Optimum in Combinatorial Optimization, *Operations Research* **33**, 1024-1049.

29. B. Golden and F. Alt, 1979. Interval Estimation of a Global Optimum for Large Combinatorial Problems, *Naval Research Logistics Quarterly* **26**, 69-77.

30. A. Elshafei, 1977. Hospital Layout as a Quadratic Assignment Problem, *Operations Research Quarterly* **28**, 167-179.

31. L. Steinberg, 1961. The Backboard Wiring Problem: a Placement Algorithm, *SIAM Review* **3**, 37-50.

32. C. Huntley and D. Brown, 1991. A Parallel Heuristic for Quadratic Assignment Problems, *Computers and Operations Research*, to appear.

33. W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, 1988. *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, NY.