Cellular Genetic Algorithms as Function Optimizers: Locality Effects

V. Scott Gordon, Keith Mathias, and Darrell Whitley
Department of Computer Science, Colorado State University

Keywords: cellular genetic algorithms, massively parallel genetic algorithms, locality, optimization, traveling salesman problem.

Abstract

In this paper, we compare the performance of several cellular genetic algorithms (CGA) to investigate their effectiveness in function optimization. Sometimes called "massively-parallel" GA's, CGA's assign one individual to each processor. Individuals are arranged in a torus and their mating is restricted to within demes (neighborhoods). Three models are considered: *fixed topology*, *random walk*, and *island*. We do not try to obtain the fastest run times nor do we compare these results to other GA's. A test suite of numerical functions and Traveling Salesman Problems (TSP) is used. We investigate the interaction between problem difficulty and measurements of the spatial locality in the CGA's. Initial results suggest that difficult numerical optimization problems require small demes (high spatial locality), but this is not evident for TSP problems.

Introduction

Several schemes have been used to parallelize genetic algorithms. Experiments have shown that parallel versions which impose some form of locality on mating are better able to solve function optimization problems than serial versions with global mating [3, 10]. Cellular genetic algorithms (CGA), sometimes called "massively-parallel" GA's, assign one individual per processor and limit mating to within demes (neighborhoods). Each individual is processed in parallel at each generation. There are many types of CGA's, depending on the selection and replacement mechanisms employed, and the size/structure of the demes.

We have tested several CGA's, representing various selection, replacement, and deme characteristics, on a suite of optimization problems. Our goal is to determine the effectiveness of different CGA models, regardless of actual parallel execution. We do not try to obtain the fastest results, nor do we compare cellular models against other approaches. Rather, we compare only these cellular models against each other.

Cellular Genetic Algorithm Models

CGA's are designed for massively parallel machines. In the simplest case one can use a single large population with one string per processor. In order to avoid high communication overhead, CGA's generally impose limitations on mating based on distance. For example, strings may conceptually be arranged on a grid, and only be allowed to perform crossover with nearby strings. This leads to a form of locality known as "isolation-by-distance". An individual's set of potential mates is called a *deme*. It has been shown that groups of similar individuals (also called *niches*) can form, and that these groups function in ways that may be similar to discrete subpopulations (islands) [1]. However, there are no actual boundaries between the groups, and nearby niches can more easily be overtaken by competing niches than in an island model. At the same time, more distant niches may be affected more slowly. Davidor calls this model the ECO GA [1].

Current literature on parallel genetic algorithms includes several examples of CGA implementation [1, 3, 6, 9]. We consider cellular genetic algorithms wherein cells reside on a torus (the strings form a grid where the edges of the grid wrap-around). Each resident individual (i.e., string) selects a mate from nearby strings, and the offspring replaces the individual according to some probability. Each string is processed in parallel at each generation. We consider three approaches: *fixed topology*, *random walk*, and *island*.

Fixed Topology CGA. In a fixed topology CGA, a deme consists of the strings residing in particular grid locations near the cell in question. The same locations comprise the deme for every generation. A mate is selected from the set and crossover is performed. Whitley has shown that this type of CGA is equivalent to a cellular automata with probabilistic rewrite rules, where the alphabet of the cellular automata is equal to the set of strings in the search space [12]. We consider three fixed topology CGA's:

1. *Deme-4.* The best of the four strings above, below, left, and right of an individual is chosen for mating (see Figure 1). If either offspring has a higher fitness, the individual is replaced by the most highly fit offspring.

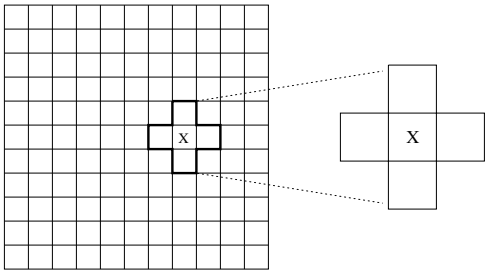

Figure 1: The deme for node X in a Deme-4 CGA

2. *Deme-4+migration.* This algorithm is identical, but *migration* is employed to copy one string to a random location at predefined intervals. The use of migration in a CGA was proposed by Gordon *et al.* [2] and Gorges-Schleuter [5], as a way of allowing distant niches to interact.

3. *Deme-12.* Demes consist of the twelve nearest strings. To reduce selective pressure, four strings are chosen randomly from the twelve, and the best of the four is selected for mating (i.e., tournament selection is used). The individual is replaced by the offspring probabilistically according to an adjustable replacement pressure.

Random Walk CGA. In a random walk CGA, a cell's deme consists of the strings which reside along a random walk starting at that cell. Each string along the walk is compared and the most fit string is chosen for mating (see Figure 2). In the implementations presented here, steps can be up, down, left, or right. It is possible for the walk to traverse the same cell more than once, and may traverse the resident individual (Figure 2, string Y). We consider walk lengths of 3, 5, and 7. Replacement is the same as for fixed topology CGA's, but is done probabilistically when the walk length is 7.

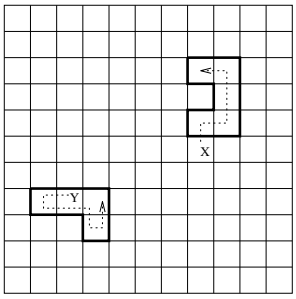

Figure 2: Possible demes for two nodes in a random walk CGA.

Island CGA. In an island CGA, the grid is divided into smaller subgrids, each of which is a torus that is processed separately using one of the above methods. Migration between subgrids occurs at predefined intervals. In the experiments presented here, each island uses the *Deme-4* algorithm described earlier.

Locality in Cellular Genetic Algorithms

The cellular genetic algorithms described in the previous section differ from one another with regard to their inherent locality. That is, the average distance between mates is smaller in some algorithms than in others. In order to determine the impact that locality has on performance, it is useful to have some means of measuring locality in the algorithms, since, for example, it is not immediately obvious whether a *random walk* algorithm has more or less locality than a *fixed topology* algorithm. At this time there exists no simple measure

of locality for genetic algorithms, partly because the various GA models each target machine characteristics which are not necessarily comparable. In this section, we will introduce locality measures for the *deme-4*, and *random walk [length=3]* algorithms.

Spatial locality refers to the likelihood that if a data element is referenced, a nearby element will also be referenced. The effect of locality on performance relates to the bandwidth and latency of the communication mechanism for the host machine, as well as the nature of the algorithm [11]. Spatial locality implies a concept of "distance" between data elements. In genetic algorithms, this can be expressed in various ways depending on how the algorithm is mapped onto a parallel machine.

There is no intrinsic requirement that each string in a CGA reside on a separate processor. It is possible to divide the entire population into logically adjacent subgrids, and then distribute the subgrids across processors. It is interesting to consider both the case where there exists one string per processor and where there exists one subgrid per processor.

One String per Processor

When each string resides on a separate processor, our approach is to examine the data transmission requirements for each node during recombination. A simplified method is to count the total number of string (or fitness value) transmissions per link, T, that a given individual (i.e., process – since there is one string per processor) requests in selecting a mate. This information will give us an indication of the total communication load in the system, since every string undergoes recombination at each generation.

In a two-dimensional grid such as was described above, there are $2n$ links, where n is the number of nodes in the grid. If each node spawns T transmissions per link, it follows that the total demand of network links is nT, and the strings transmitted per link is:

$$strings/link = \frac{n \cdot T}{2 \cdot n} = \frac{T}{2} \tag{1}$$

In the *deme-4* algorithm, each string needs to access the four surrounding elements, and the simplest manner is for them to merely transmit themselves. Thus $T = 4$, and from equation (1) the number of strings per link is $T/2 = 2$.

In a *random walk* algorithm, a short random traversal of part of the grid is made, and a superior string is ultimately returned to the original node. Thus there are data transmissions associated with the walk, and additional data transmissions associated with returning the mate. Clearly the number of data transmissions associated with the walk is equal to the walk length. The number of data transmissions associated with returning the mate equals the average distance from the original node to the last node in the walk.

For a walk length of three, it is simple to count the number of ways that a walk can end up on various surrounding nodes. In this case, there are 24 paths which end up on nodes which are three links away from the original node, and

36 paths which end up on nodes which are one link away from the original node. Thus the average distance is $[3(24) + 1(36)]/60] = 1.8$. Thus $T = 3 + 1.8 = 4.8$ and from equation (1) the number of strings per link is 2.4.

One Subgrid per Processor

A logical approach to examining the locality of a CGA in which strings are contained in subgrids, each of which reside on separate processors, is to compare the percentage of remote references. Here the percentage of remote references increases as the percentage of strings at the edges of the subgrids increases. This leads to the apparent anomaly that locality (and thus performance) degrades as the number of processors increase (assuming that total population size is fixed). Of course, this actually represents a trade-off since more processors also means more parallelism.

One way to minimize the percentage of edge strings is to divide the total population, which is assumed to be a square, into equal subgrids which are themselves squares. Next, we define n to be the total number of nodes in the population and n_s to be the number of nodes in a subgrid. The length of a side of the population grid is then $\sqrt{n}$, and $\sqrt{n_s}$ is the length of a side of a subgrid. Finally, we assume that all remote accesses have the same overhead.

Deme-4:

Since we are assuming that the grid is a torus, with wrap-around edge elements, each subgrid is structurally equivalent. Therefore the percentage of remote accesses is the same for a subgrid as it is for the entire population. Thus we need only calculate the percentage of remote accesses for a single subgrid.

In a subgrid, all interior nodes mate only with nodes inside the subgrid. Edge nodes may mate with nodes within the grid, or with nodes in one or more other subgrids. In particular, for a deme size of 4, the process of reproduction for one interior node results in 6 local data accesses (4 for selecting a mate, one for retrieving the contents of the node itself, and one for replacing the node with the offspring). Nodes at the corners of the subgrid require 4 local data accesses and 2 remote accesses (2 remote accesses and two local accesses for selecting a mate, one local access for retrieving the contents of the node itself, and one local access for replacement). Nodes at the sides of the subgrid similarly require 5 local accesses and 1 remote access. The number of corner nodes is always 4, the number of side nodes is $4(\sqrt{n_s} - 2)$, and the number of interior nodes is $(\sqrt{n_s} - 2)^2$. Assuming $n_s \geq 9$, the total number of remote accesses A_r and local accesses A_l is:

$$A_r = (2 \cdot 4) + [1 \cdot (4\sqrt{n_s} - 8)] = 4\sqrt{n_s} \tag{2}$$

$$A_l = (4 \cdot 4) + [5 \cdot (4\sqrt{n_s} - 8)] + 6 \cdot (\sqrt{n_s} - 2)^2 = 6n_s - 4\sqrt{n_s} \tag{3}$$

It follows that the percentage of remote accesses is:

$$\%remote_{grid-4} = \frac{4\sqrt{n_s}}{6n_s} = \frac{2}{3\sqrt{n_s}} \tag{4}$$

Random walk

Determining the percentage of remote references in a random walk with one subgrid per processor requires considering the various cases where a walk could leave the local subgrid. Nodes which are closest to the edges are more likely to generate remote references than nodes which are not as near to an edge. Nodes near corners may generate remote references to more than one external processor. There are many machine-dependent considerations, such as the overhead associated with obtaining one element from each of two processors versus obtaining two elements from the same processor.

In order to simplify the analysis, we make the following assumptions. First, we assume that all remote accesses engender the same overhead. Thus the overhead associated with two remote references is twice that of a single reference, regardless of whether they are from the same processor or different processors. Second, we assume that the walk is generated ahead of time, and then all relevant data accesses are made.

Figure 3 contains two charts: the *template grid* (left) contains the number of ways that neighboring cells can be traversed (i.e., how many paths traverse each neighboring cell). The *edge grid* (right) labels the cells which are close to the edges of a subgrid.

			1			
		3	4	3		
	3	8	20	8	3	
1	4	20	16	20	4	1
	3	8	20	8	3	
		3	4	3		
			1			

template grid

x	x	x	i	h	g
x	x	x	i	h	g
x	x	x	i	h	g
i	i	i	f	e	d
h	h	h	e	c	b
g	g	g	d	b	a

edge grid

Figure 3: The *template grid* shows the number of ways, starting from the center cell, that various cells can be traversed during a random walk of length 3. The *edge grid* shows the various categories of critical cells (near edges and corners) which may mate with remote cells during a random walk of length 3. Overlaying the template grid on top of each cell in the edge grid allows one to determine the expected number of remotely referenced cells.

The total number of possible data references is the sum of the values in the template grid, which equals 172. By overlaying the template grid on top of each cell in the edge grid, it is possible to determine how many of these 172 possible references would be remote. Enumerating these then becomes relatively simple for each category of edge elements *a*-*i* and *x* in the edge grid:

(a nodes) $Remote = 92/172$ (f nodes) $Remote = 2/172$
(b nodes) $Remote = 61/172$ (g nodes) $Remote = 53/172$
(c nodes) $Remote = 22/172$ (h nodes) $Remote = 11/172$
(d nodes) $Remote = 54/172$ (i nodes) $Remote = 1/172$
(e nodes) $Remote = 12/172$ (x nodes) $Remote = 0/172$

Note that for all interior nodes (i.e., those that are at least three cells away from an edge, marked "x" in Figure 3), the percentage of remote references is 0. Assuming that $n_s \geq 36$, the number c_t of each type of node t in any subgrid is given by:

$c_a = 4$ $c_f = 4$
$c_b = 8$ $c_g = 4\sqrt{n_s} - 24$
$c_c = 4$ $c_h = 4\sqrt{n_s} - 24$
$c_d = 8$ $c_i = 4\sqrt{n_s} - 24$
$c_e = 8$ $c_x = (\sqrt{n_s} - 6)^2$

It follows that the fraction of remote accesses is:

$$\frac{4(\frac{116}{172}) + 8(\frac{127}{172}) + (4\sqrt{n_s} - 24)(\frac{65}{172})}{n_s} \tag{5}$$

which simplifies to:

$$remote_{rw} = \frac{65\sqrt{n_s} - 20}{43n_s} \tag{6}$$

Equation 6 does not include the 2 local accesses necessary to fetch and to replace the particular node in question (for mating). Including this reduces the fraction of remote accesses to:

$$remote_{rw} = 1 - [\frac{3}{5}(1 - \frac{65\sqrt{n_s} - 20}{43n_s}) + \frac{2}{5}] = \frac{39\sqrt{n_s} - 12}{43n_s} \tag{7}$$

Table 1 contains examples of locality calculations using the expressions described earlier, for a 4-processor machine. Figure 4 shows a plot of equations 4 and 7.

Various GAs. 4 processors, popsize of 400, 2500			
	Refs	#Rem	%Rem
Grid4, p = 4x(10x10)	2400	160	6.67
Grid4, p = 4x(25x25)	15000	400	2.67
RWalk3, p = 4x(10x10)	2000	176	8.79
RWalk3, p = 4x(25x25)	12500	448	3.58

Table 1: Summary of locality for two CGA's.

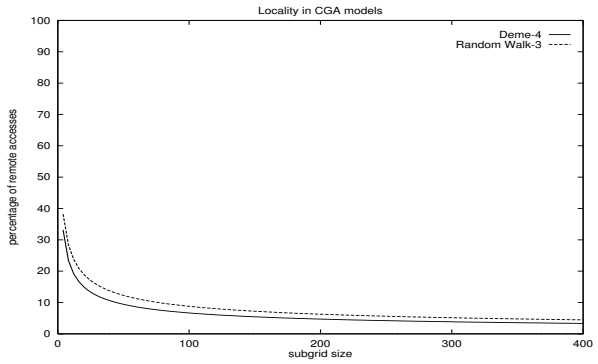

Figure 4: Locality in CGAs with small demes / short walk lengths

Test Suite

Our test suite consists of three numerical functions and two Traveling Salesman Problems (TSP). The numerical problems, all minimization functions, are hereafter referred to as the Rastrigin, Schwefel, and Griewank functions (described by Mühlenbein *et al.* [10]). We consider TSP problems of size 30 and 105. These functions were chosen because: (1) they require long encodings, (2) they contain many local minima, and (3) they appear frequently in other genetic algorithm literature.

The Rastrigin function is a fairly difficult problem for genetic algorithms due to the large search space and large number of local minima. The Schwefel function is characterized by a second-best minimum which is far away from the global optimum. In the Schwefel function, V is the negative of the global minimum, which is added to the function so as to move the global minimum to zero, for convenience. The exact value of V depends on system precision; for our experiments $V = 4189.829101$. Griewank's function is difficult for genetic algorithms because the product term causes the 10 substrings to be strongly interdependent. We used Gray coding on the Rastrigin and Griewank functions, but not on the Schwefel function. The Schwefel function can be solved faster using Gray coding, but our results are determined without Gray coding. The three numeric functions are as follows:

$$R\text{:}\ f(x_i|_{i=1,n}) = 200 + \sum_{i=1}^{20} x_i^2 - 10cos(2\pi x_i) \quad x_i \in [-5.12, 5.11]$$

$$S\text{:}\ f(x_i|_{i=1,10}) = V + \sum_{i=1}^{10} -x_i sin(\sqrt{|x_i|}) \quad x_i \in [-512, 511]$$

$$G\text{:}\ f(x_i|_{i=1,10}) = \sum_{i=1}^{10} \frac{x_i^2}{4000} - \prod_{i=1}^{10} cos(\frac{x_i}{\sqrt{i}}) + 1 \quad x_i \in [-512, 511]$$

Performance Measurements

We keep the following CGA parameters constant for the numeric functions: popsize = 2500, mutation rate = .005, probability of crossover = 100%, and replacement pressure = 90% (for Deme-12 and RW-7). For the Island-CGA we used 25 subpopulations, each of which were of size 100 (10x10), and a swap interval of 5 generations. The migration interval for Deme-4+migration is set to 50 generations. For the TSP problems, we tried population sizes of 1600 and 4225 and no mutation was used.

For the numeric functions, we employ two-point reduced surrogate crossover and next-point mutation (determining the next bit to be mutated rather than calculating mutation for every bit [4]). For the TSP problems, we use Whitley's edge-3 recombination [8].

Computation times are expressed in terms of *generations*. For the numeric functions, each CGA performs two function evaluations for each recombination done at each location in the grid, every generation. Thus twice as many evaluations are performed in a numeric function as in a TSP problem (edge recombination produces a single offspring [8]). We normalize the numeric function results by doubling the number of reported generations. Therefore, the total number of function evaluations can be determined by multiplying the numbers in the tables by the population size.

Avg generations to solve (Avg) and standard dev. (Std)				
	Rastrigin		Schwefel	
CGA	Avg	Std	Avg	Std
D-4	485	95	137	26
D-4+mig	485	93	136	26
D-12	557	109	126	25
RW-3	606	115	160	31
RW-5	535	102	148	28
RW-7	685	141	158	30
I-CGA	521	103	152	29

number solved (ns) and avg best (avg)		
	Griewank	
CGA	ns	avg
D-4	20	.009
D-4+mig	24	.007
D-12	21	.016
RW-3	29	.005
RW-5	25	.006
RW-7	23	.016
I-CGA	19	.012

Table 2: CGA performance on numeric functions. Population size = 2500, mutation = .005, crossover probability = 100%, replacement pressure = 90% (where applicable). For Island-CGA, number of subpopulations = 25, subpopulation size = 100 (10x10), migration interval = 5 (generations). For Deme-4+mig, migration interval = 50. All use two-point reduced surrogate crossover.

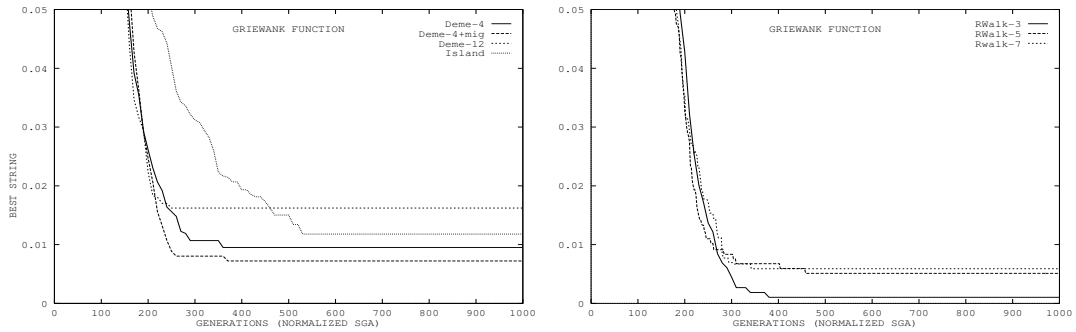

Figure 5: Performance of CGA's on the Griewank function.

Performance on numeric functions

The performance of each of the seven CGA's on the Rastrigin, Schwefel, and Griewank functions (with no local optimization) is shown in Table 2. For the Rastrigin and Schwefel functions, we report the average number of generations it takes for each CGA to find the optimum (averaged over 30 runs), and the standard deviation. The Griewank function is harder, and the CGA's are not always able to solve the problem within 500 CGA generations. Therefore we report the average number of runs in which the global optimum is found, along with the average fitness of the best strings found at the end of the runs. Figure 5 contains convergence plots for each CGA on the Griewank function.

Performance on TSP problems

The performance of each of the CGA's on the TSP problems is shown in Table 3. On the 105-city problem, we tested the use of local optimization (2-opt [7]) on the initial population, as well as different population sizes, to see if they would affect the relative order of performance of the CGA's. For the 30-city and 105-city+2-opt problems, we report the average number of generations it takes for each CGA to find the optimum (averaged over 30 runs – except for 2-opt results which are averaged over 10 runs), and the standard deviation. The blind 105-city problems are harder, and the CGA's are not always able to solve the problem within 350 CGA generations. Therefore we report the average number of runs in which the global optimum is found, along with the average fitness of the best strings found at the end of the runs. The poor performance of the Island-CGA on the Griewank function suggested that it would not be advantageous to try Island-CGA on the TSP.

Performance summary

Table 4 shows the performance rankings for the fixed topology and random walk CGA's tested, by problem. We list the functions left-to-right in increasing order of difficulty by problem type (numeric vs. TSP). The rankings at the right are broken down by CGA type (fixed topology vs. random walk) to illustrate observed locality effects. How locality factors impact performance is key to understanding which CGA configurations are more desirable.

Avg generations to solve (Avg) and standard dev. (Std)						
	30 blind popsize=1600		105 2-opt popsize=1600		105 2-opt popsize=4225	
CGA	Avg	Std	Avg	Std	Avg	Std
D-4	57	3.4	108	12.4	97	6.8
D-4+mig	58	4.1	111	20.2	99	8.0
D-12	46	4.0	91	6.1	89	5.4
RW-3	52	4.0	117	11.1	108	6.7
RW-5	42	2.9	91	8.7	89	4.5
RW-7	40	2.2	87	10.6	81	4.0

Number solved (ns) and avg best (avg)				
	105 blind popsize=1600		105 blind popsize=4225	
CGA	ns	avg	ns	avg
D-4	0	14569	0	14561
D-4+mig	0	14725	0	14564
D-12	20	14471	27	14387
RW-3	3	14515	3	14418
RW-5	23	14430	30	14383
RW-7	17	14448	27	14385

Table 3: CGA performance on 30 and 105-city TSP functions. Edge-3 recombination, population sizes as indicated, crossover probability = 100%, replacement pressure = 90% (where applicable). For Deme-4+mig, migration interval = 50.

Discussion and Conclusions

It is interesting to note that, for the numeric functions, the algorithms with the most locality (smallest fixed demes or shortest walks) perform worst on the easiest function (Schwefel) and best on the hardest function (Griewank). This is consistent with the strategy of decreasing the selective pressure for harder problems. But this trend is not observed for the TSP problems, where algorithms with *less* locality consistently outperform algorithms with high locality. More experiments need to be done to determine how (or whether) performance differences are related to problem type.

While all of the TSP algorithms perform better when 2-opt is applied to the initial population, the only observed effect on the rankings is that the CGA with a random walk of length 3 (RW-3) performs much worse. One explanation for this effect could be related to Mathias and Whitley's observation that, on larger problems, 2-opt generates tours that are difficult to improve via edge-3 crossover due to accumulated failure distances [8]. The very high locality in RW-3 might then promote the development of suboptimal stable niches. Another

CGA	Sh	Rs	Gr	30	105_s^2	105_l^2	105_s^b	105_l^b
D-4	3	2	4	5	4	4	5	5
D-4+mg	2	1	3	6	5	5	6	6
D-12	1	5	6	3	3	2	3	3
RW-3	7	6	1	4	6	6	4	4
RW-5	4	4	2	2	2	3	1	1
RW-7	6	7	5	1	1	1	2	2
I-CGA	5	3	7	-	-	-	-	-

Table 4: Ranked performance of each of the CGAs on Schwefel (Sh), Rastrigin (Rs), Griewank (Gr), and the 30 and 105-city TSP. On the 105-city problems, superscripts indicate blind (b) or with initial 2-opt (2), and subscripts indicate the population size ($s = 1600$ and $l = 4225$).

explanation could be that "moats" exist between niches (i.e., strings that are different from the strings in either niche), and that RW-3 simply cannot reach beyond the moat to a neighboring niche. The problem would be exacerbated by 2-opt because it would cause the niches to be of relatively high fitness, and strings in the moat might rarely be selected for recombination. Further research on niche formation would be useful, as well as a characterization of the nature of strings at niche boundaries.

The Island-CGA performs rather poorly on the numeric suite, which comes as somewhat of a surprise. It may be that a CGA forms niches more effectively when the grid size is large, and thus further subdivision into arbitrary islands reduces computational power.

Another interesting observation is that adding migration to the Deme-4 model improves performance on the numeric functions. This could be explained by considering that niches are unable to interact when they are physically separated on the grid. Periodic migration allows two high-fitness niches to interact whereas they might not otherwise. Again, the TSP data does not corroborate the numeric data, and migration was detrimental in all TSP cases.

It would be convenient if CGA's solved hard problems better when a high degree of locality is imposed, since that would imply that CGA's which incur the least computational overhead are also the most effective at solving hard problems. The results for numeric problems support this hypothesis, but preliminary results on TSP problems raise some doubt.

References

[1] Y. Davidor. A Naturally Occurring Niche & Species Phenomenon: The Model and First Results. *Proc. of the 4th ICGA*, Morgan-Kaufmann, 1991

[2] V. Gordon, D. Whitley, and A. Böhm. Dataflow Parallelism in Genetic Algorithms. *PPSN2*, North Holland, 1992

[3] V. Gordon and D. Whitley Serial and Parallel Genetic Algorithms as Function Optimizers. *Proc. of the 5th ICGA*, Morgan Kaufmann, 1993

[4] V. Gordon and D. Whitley. A Machine-Independent Analysis of Parallel Genetic Algorithm Performance. *submitted for publication.* (1993)

[5] M. Gorges-Schleuter. Comparison of Local Mating Strategies in Massively Parallel Genetic Algorithms. *PPSN2*, North Holland, 1993

[6] D. Hillis. Co-Evolving Parasites Improve Simulated Evolution as an Optimizing Procedure. *Physica D 42*, North-Holland, 1990

[7] S. Lin and B. Kernighan. An Efficient Heuristic Procedure for the Traveling Salesman Problem. *Operations Research*, 21:498-516, 1973

[8] K. Mathias and D. Whitley. Genetic Operators, the Fitness Landscape, and the Traveling Salesman Problem. *PPSN2*, North Holland, 1992

[9] H. Mühlenbein, M Gorges-Schleuter, and O. Kramer. Evolution Algorithms in Combinatorial Optimization. *Parallel Computing 7*, North Holland, 1988

[10] H. Mühlenbein, M. Schomisch, and J. Born. The Parallel Genetic Algorithm as Function Optimizer. *Parallel Computing 7*. North-Holland, 1991

[11] A.Z. Spector. Achieving Application Requirements, in *Distributed Systems.* Ed. S. Mullender, ACM Press, ©1989

[12] D. Whitley. Cellular Genetic Algorithms. *Proc. of the 5th ICGA*, Morgan Kaufmann, 1993