CSE 620 Mini-Project 2

Sima Shafaei
Student, CSE 620
University of Louisville
Louisville, KY
sima.shafaei@louisville.edu

Jim LeFevre
Student, CSE 620
University of Louisville
Louisville, KY
james.lefevre@louisville.edu

***Abstract*—This mini-project compares the performance of a basic genetic algorithm (GA) to that of niching GA algorithms. Sharing and deterministic crowding were chosen as representative niching algorithms. Experiments evaluated F_{max}, F_{min}, and F_{avg} of each algorithm when applied to two multimodal fitness landscapes. Crossover rate and mutation rate were separately tuned for each algorithm-landscape combination. Both niching methods outperformed the basic GA.**

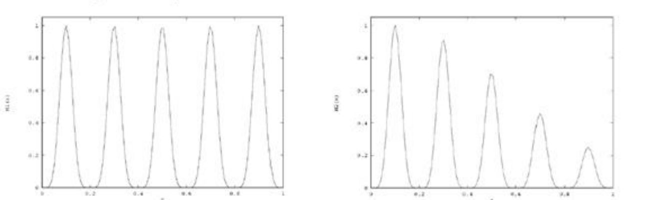
I. Introduction

A genetic algorithm (GA) is initialized with a set of n candidate solutions (individuals). The fitness of an individual is a measure of the quality of the solution. Each iteration consists of a cycle of variation and selection. First, new individuals are generated by a variation operator. The variation operator generally combines aspects of two existing individuals, and then introduces additional random variation, to produce each new individual. A selection operator determines which new individuals will replace which existing individuals, relying on an evaluation function that determines the fitness of each individual. The revised set of individuals is carried forward to the next iteration [1].

When multiple fitness optima exist, basic GA methods have the weakness that the population of individuals converges to a single optimum [2]. Niching methods are meant to increase the likelihood of finding all optima [3]. We selected sharing and deterministic crowding (DC) as niching methods to test.

II. Problem Statement

The *M1* and *M4* functions, which are shown in Figure 1 [3], are both continuous scalar-valued functions of a single continuous scalar parameter. In GA terms, the phenotype is the input x. Each function is multimodal, meaning that multiple local optima exist. In *M1*, there are multiple global optima with equal fitness, and in *M4* there is a single global optimum and four local optima. These functions are designed to make it difficult for a basic GA to find all optima because of its tendency to converge to a single optimum.



$$M1(x) = \sin^6(5\pi x) \qquad M4(x) = e^{-2(\ln 2)(\frac{x-0.08}{0.854})^2}\sin^6(5\pi[x^{0.75} - .05])$$

Fig. 1. *M1* and *M4* benchmark functions

III. Overview of Algorithms

A. Basic GA

The following is GA pseudocode, in which g is the number of generations/iterations, p is the number of individuals that reproduce per iteration, n is the population size, and the variation operator includes both crossover and mutation:

```
for i in 1:g
    for j in 1:p/2
        Randomly select parents p1 and p2
        Apply variation operator to p1 and p2 to generate c1 and c2
        Evaluate fitness of c1 and c2
    Add all p children to the population
    Select the n most fit individuals to carry forward to the next iteration
```

We wrote GA code in Python, adapting an example available online [4][5]. In this implementation, initial phenotypes are randomly assigned from a uniform distribution in the range [0,1].

Instead of converting phenotypes into a separate genotype representation, the variation operator works directly with the phenotype (a real number). Crossover and mutation occur with user-specified probabilities. When crossover is applied, if the parent phenotypes are p_1 and p_2, γ is a user-specified parameter (we used 0.1), and α is a random number from a uniform distribution in the range [$-\gamma$, $1 + \gamma$], then the phenotypes of children c_1 and c_2 are obtained as follows:

$$c_1 = \alpha p_1 + (1 - \alpha)p_2$$

$$c_2 = \alpha p_2 + (1 - \alpha)p_1$$

When mutation is applied to c_1, the output c_1' is obtained as follows, where σ is a user-specified parameter (we used 0.1) and α is a random number from a normal distribution with mean 0 and standard deviation 1:

$$c_1' = c_1 + \sigma\alpha$$

B. Sharing

Sharing modifies the fitness of each individual by subtracting a penalty for each other sufficiently similar individual. This provides a relative selection advantage for those new individuals that have emerged in unexplored regions [3].

Sharing pseudocode follows, in which g is the number of generations/iterations, p is the number of individuals that reproduce per iteration, e is an individual member of the population as of the beginning of an iteration, n is the population size, $d(a, b)$ is the distance between individuals a and b, σ is the user-specified sharing threshold (we used 0.5), and the variation operator includes both crossover and mutation:

```
for i in 1:g
    for j in 1:p/2
        Randomly select parents p1 and p2
        Apply variation operator to p1 and p2 to generate c1 and c2
        Evaluate unadjusted fitness of c1 and c2
        Penalize c1 fitness for each e for which d(c1, e) < σ
        Penalize c2 fitness for each e for which d(c2, e) < σ
    Add all p children to the population
    Select the n most fit individuals to carry forward to the next iteration
```

The adjusted fitness of a child c is determined by dividing the unadjusted fitness by s. The value of s is defined as follows, in which α is a user-specified parameter (we used 1) and the other variables are defined as above:

$$s = \sum_{i=1}^{n} 1 - \left(\frac{d(c, e_i)}{\sigma}\right)^{\alpha}, \forall\, e : d(c, e) < \sigma$$

C. DC

DC maintains phenotype diversity by making only one existing individual at risk of replacement by any given offspring. Specifically, a child competes with whichever of its parents is more similar to that child [3]. This protects individuals who are distant from a child, and perhaps located at other peaks, from being replaced.

DC pseudocode follows, in which g is the number of iterations/generations, n is the population size, $d(a, b)$ returns the distance between individuals a and b, and the variation operator includes both crossover and mutation [6]:

```
for i in 1:g
    for j in 1:n/2
        Randomly select (without replacement) parents p1 and p2
        Apply variation operator on p1 and p2 to generate c1 and c2
        Evaluate fitness f of c1 and c2
        if d(p1, c1) + d(p2, c2) <= d(p1, c2) + d(p2, c1)
                if f(c1) > f(p1), then select c1 and remove p1
                if f(c2) > f(p2), then select c2 and remove p2
        else
        if f(c2) > f(p1), then select c2 and remove p1
        if f(c1) > f(p2), then select c1 and remove p2
```

An iteration of DC proceeds as follows. Two individuals from the existing population are selected as a set of parents. They are then labeled as ineligible for parentage for the rest of the iteration. The variation operator operates as it does in the basic GA. There are 2 possible tournament schemes; the one chosen is the one in which the sum of between-competitor distances is minimized. If a child has the greater fitness than the competing parent, the parent is deleted from the population and the child is added. Otherwise, the child is discarded and the parent remains. This sequence is repeated until all individuals that were carried forward from the prior generation have been selected as parents.

IV. Experimental Protocol

A. Static Parameters

Each experiment had a population size of 100 individuals and was run for 15 iterations.

B. Parameter Tuning

In order to find the best crossover rate (p_c) and mutation rate (p_m) for each algorithm, four candidate p_c values (1, 0.75, 0.5, 0.25) and three candidate p_m values (0.05, 0.1, 0.2) were evaluated.

C. Performance Comparison

Each algorithm was run 10 times for each combination of candidate parameters. Mean values of maximum fitness (F_{max}), average fitness (F_{avg}), and minimum fitness (F_{min}) were calculated. The algorithm with the highest F_{avg} was chosen as the best run.

V. Analysis of Results

A. Parameter Turning

1) Benchmark M1

For each combination of p_c and p_m, the results (mean values over 10 runs) for *M1* are displayed in Figure 2. In tables, each cell shows the average fitness for a given p_c (row) and p_m (column). The red box indicates the cell with the best performance.

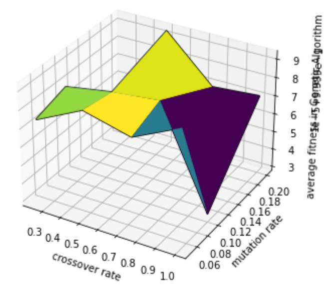
2) Benchmark M4

For each combination of p_c and p_m, the results (mean values over 10 runs) for *M4* are displayed in Figure 3. In tables, each cell shows the average fitness for a given p_c (row) and p_m (column). The red box indicates the cell with the best performance.

B. Algorithm Performance

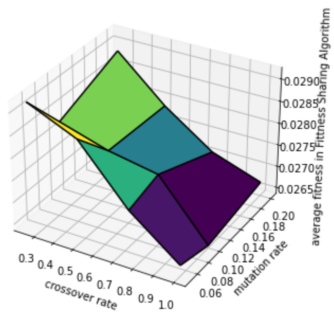
The performance of each algorithm, when using the best parameters found for that algorithm and benchmark, is displayed visually in Figures 7-12. Figures 13 and 14 display F_{min}, F_{avg}, and F_{max} for each generation, when the algorithms were run on *M1* and *M4*, respectively.

Genetic Algorithm



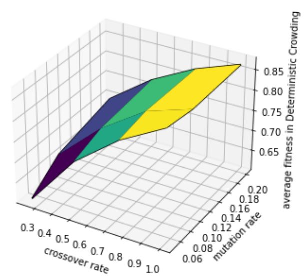
	0.05	0.1	0.2
1	0.999992	0.999932	0.999971
0.75	0.99998	0.999986	0.999969
0.5	0.999986	0.999984	0.999993
0.25	0.999974	0.999979	0.999929

Fitness Sharing Algorithm



	0.05	0.1	0.2
1	0.0266053	0.0264156	0.0266718
0.75	0.0274789	0.0276874	0.0270609
0.5	0.0287117	0.027893	0.0278338
0.25	0.0292007	0.0282303	0.028821

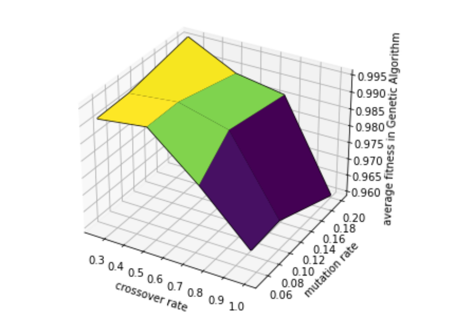
Deterministic Crowding



	0.05	0.1	0.2
1	0.87419	0.863335	0.86661
0.75	0.814347	0.828005	0.827339
0.5	0.731376	0.756284	0.771711
0.25	0.605771	0.659279	0.694566

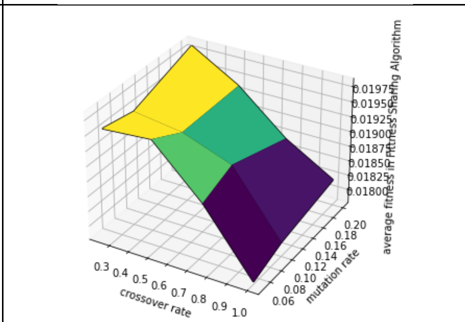
Fig. 2. Average fitness as a function of mutation rate and cross over rate with benchmark *M1*

Genetic Algorithm



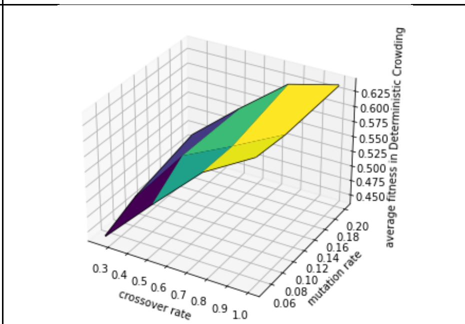
	0.05	0.1	0.2
1	0.96691	0.967546	0.959472
0.75	0.981149	0.9895	0.985859
0.5	0.993425	0.993431	0.988442
0.25	0.991638	0.992035	$\boxed{0.995638}$

Fitness Sharing Algorithm



	0.05	0.1	0.2
1	0.0178163	0.018012	0.0182113
0.75	0.0188584	0.0190602	0.018692
0.5	0.0196479	0.0193629	0.0193762
0.25	0.0195883	0.0194817	$\boxed{0.0198476}$

Deterministic Crowding



	0.05	0.1	0.2
1	$\boxed{0.64156}$	0.636062	0.63814
0.75	0.594531	0.594743	0.617971
0.5	0.518299	0.555885	0.555165
0.25	0.438813	0.462071	0.488124

Fig. 3. Average fitness as a function of mutation rate and cross over rate with benchmark *M4*

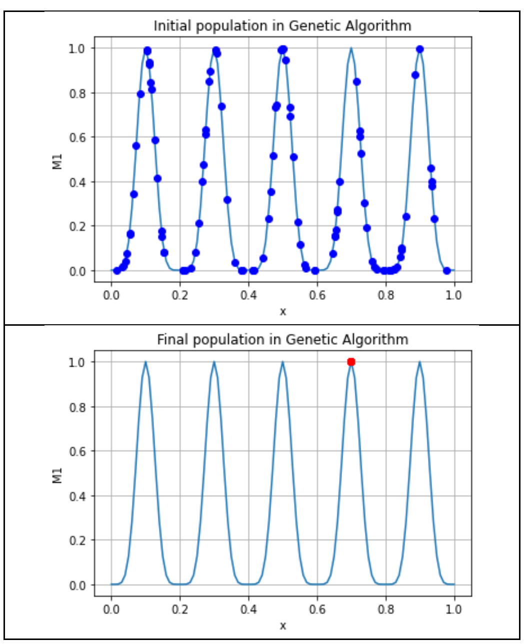


Fig. 4. Initial and final population in genetic algorithm after running on benchmark *M1* with best obtained parameters

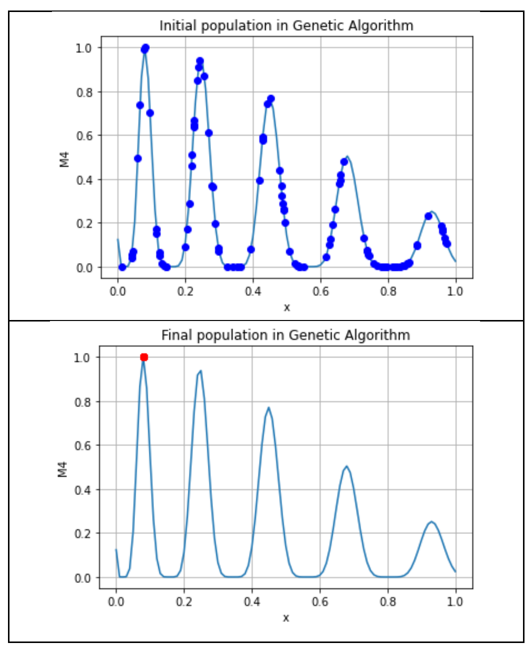


Fig. 5. Initial and final population in genetic algorithm after running on benchmark *M4* with best obtained parameters

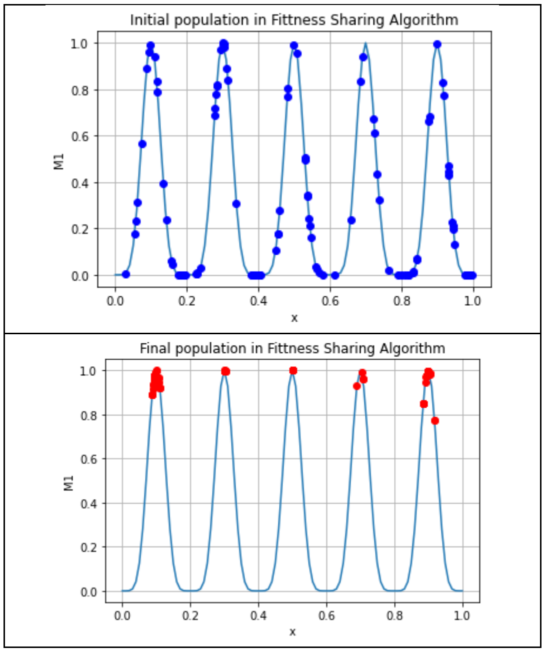


Fig. 6. Initial and final population in fitness sharing algorithm after running on benchmark *M1* with best obtained parameters

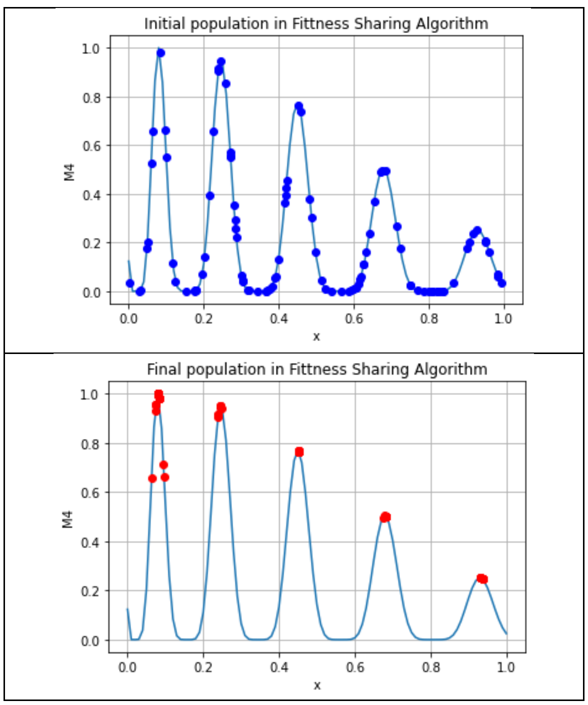


Fig. 7. Initial and final population in fitness sharing algorithm after running on benchmark *M4* with best obtained parameters

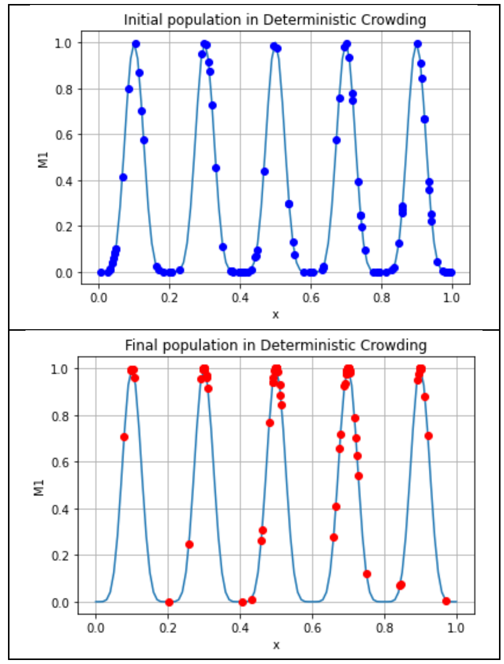


Fig. 8. Initial and final population in deterministic crowding algorithm after running on benchmark *M1* with best obtained parameters

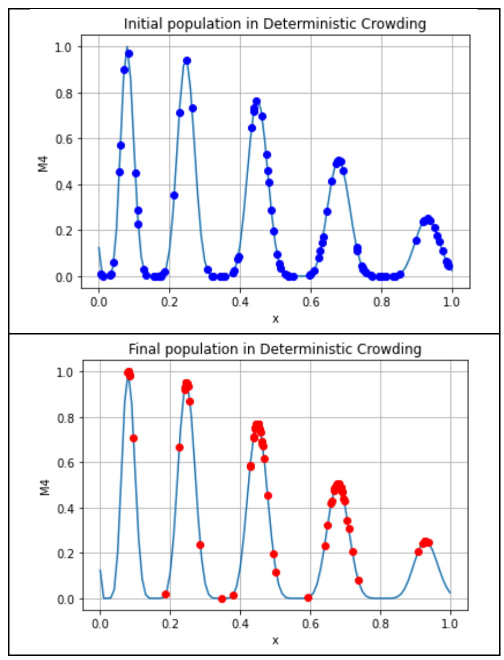


Fig. 9. Initial and final population in deterministic crowding algorithm after running on benchmark *M4* with best obtained parameters

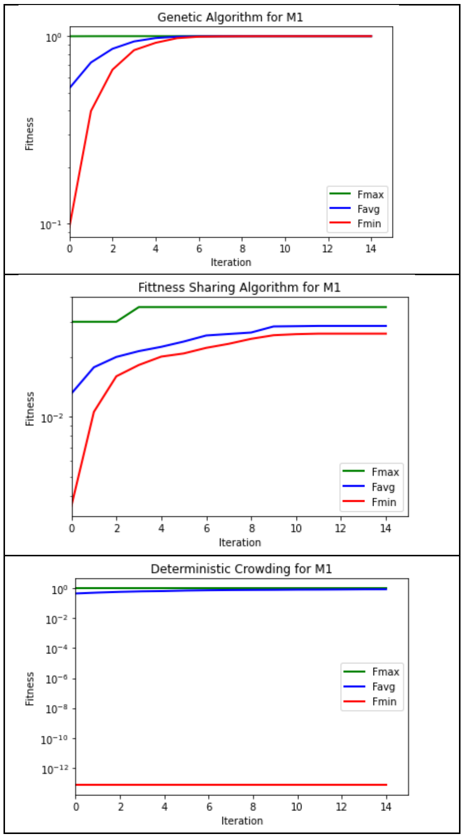


Fig. 10. F_{min}, F_{max} and F_{avg} for genetic algorithm, fitness sharing, and deterministic crowding when run on benchmark *M1* with best obtained parameter

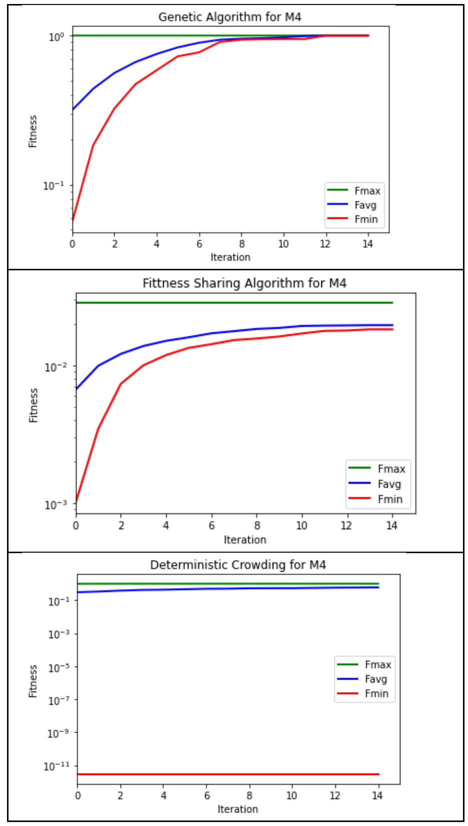


Fig. 11. F_{min}, F_{max} and F_{avg} for genetic algorithm, fitness sharing, and deterministic crowding when run on benchmark *M4* with best obtained parameter

VI. Conclusions

The optimal values of p_c and p_m varied widely among algorithms. The basic GA worked best with high p_m and relatively low p_c, for both *M1* and *M4*. The sharing algorithm worked best with low p_c; a low p_m worked best for *M1* and a high p_m was best for *M4*. The DC algorithm preferred a high p_c and a low p_m.

The tendency of the basic GA to drift toward a single peak was evident when run on both *M1* and *M4*. GA did find the global optimum in *M4*, at least. Sharing demonstrated a vast improvement over GA, finding each peak in *M1* and *M4*. DC likewise found each peak in *M1* and *M4*.

Given that sharing and DC both identified all peaks, can one algorithm be chosen as the best? The final population was more tightly concentrated at the peaks with sharing than with DC, which might be valuable in some applications. DC has lower computational complexity than sharing, which is $O(N^2)$ because of the need to search for nearby samples in order to adjust fitness. Also, DC converged to near its peak value of F_{avg} sooner than sharing. This suggests that DC may be more effective than sharing when the total number of generations in a run is low. Sharing might be more useful when tight convergence to peaks is important; DC might be preferred when computational efficiency is valued.

One limitation of this project is that the benchmarks were not highly challenging tests, given the population size. The initial population was generally distributed such it was not necessary to escape from a local optimum in order to find all peaks. The algorithms did not need to seek out new peaks in order to succeed, they only needed to not lose track of the peaks that were known to the initial population.

F_{min}, F_{max} and F_{avg} can be misleading if the goal is to find all fitness peaks. Future research could consider alternative measures of performance, and could tune parameters such as population size and number of generations (and for sharing, σ and α).

References

[1] Z. Michalewicz and D. B. Fogel. How to Solve It: Modern Heuristics, 2nd ed. Berlin: Springer, 2004.

[2] O. Nasraoui. Evolutionary Algorithms: Slides for CECS 694: Topics in Combinatorial Optimization, unpublished.

[3] O. Nasraoui. Advanced Topics in Evolutionary Algorithms: Slides for CECS 694: Topics in Combinatorial Optimization, unpublished.

[4] M. K. Heris. Genetic Algorithm in Python – Part A – Practical Genetic Algorithms Series. In YouTube.

[5] M. K. Heris. Genetic Algorithm in Python – Part B – Practical Genetic Algorithms Series. In YouTube.

[6] S. W. Mahfoud. A Comparison of Parallel and Sequential Niching Methods. In Proceedings of the Sixth International Conference on Genetic Algorithms, pages 136-143, 1995.