Testing the Intermediate Disturbance Hypothesis: Effect of Asynchronous Population Incorporation on Multi-Deme Evolutionary Algorithms*

J.J. Merelo[1], A. Mora[1], P.A. Castillo[1], J.L.J. Laredo[1], L. Araujo[2], K. Sharman[3], A.I. Esparcia-Alcázar[3], E. Alfaro-Cid[3], and C. Cotta[4]

[1] Dept. ATC, Universidad de Granada, Spain
jj@merelo.net
[2] Dept. LSI, UNED, Madrid, Spain
[3] Instituto Tecnológico de Informática, Valencia, Spain
[4] Dept. LCC, Universidad de Málaga, Spain

Abstract. In P2P and volunteer computing environments, resources are not always available from the beginning to the end, getting incorporated into the experiment at any moment. Determining the best way of using these resources so that the exploration/exploitation balance is kept and used to its best effect is an important issue. The Intermediate Disturbance Hypothesis states that a moderate population disturbance (in any sense that could affect the population fitness) results in the maximum ecological diversity. In the line of this hypothesis, we will test the effect of incorporation of a second population in a two-population experiment. Experiments performed on two combinatorial optimization problems, MMDP and P-Peaks, show that the highest algorithmic effect is produced if it is done in the middle of the evolution of the first population; starting them at the same time or towards the end yields no improvement or an increase in the number of evaluations needed to reach a solution. This effect is explained in the paper, and ascribed to the *intermediate disturbance* produced by first-population immigrants in the second population.

1 Introduction

The volatility of resources is an important feature of some distributed computation environments, such as those based on P2P or voluntary computation: resources appear and disappear in a continuous and unpredictable manner. For instance, a new node might be added to an Evolving Agents (EvAg) [1] P2P distributed evolutionary computation experiment, or a new client might download the web page to start a browser-based evolutionary experiment [2,3]. Using these high-churn computing environment efficiently so that their contribution to

* Funded by the Spanish MICYT project NoHNES (Spanish Ministerio de Educación y Ciencia - TIN2007-68083) and the Junta de Andalucía P06-TIC-02025.

G. Rudolph et al. (Eds.): PPSN X, LNCS 5199, pp. 266–275, 2008.

the common compute pool does not get lost is obviously an important issue, and rules for using the node's computing resources efficiently (or at all) have to be researched. If the evolutionary experiment is sufficiently advanced, it might be the case that computation performed in a certain way by the new node is useless, and it will be best devoted to a new experiment (or to help the experiment in a different way). The same problem arises also in other heterogeneous and asynchronous computing experiments: even if all nodes start at the same time, those with less computing power will eventually *lag behind*, falling into a *less evolved* state that might render them useless, churning out individuals whose state would have made them eliminated in other nodes whose populations are more advanced.

There are, in principle, two different ways of creating this initial population: in a completely random way, or as an (imperfect) duplicate of the existing population. If we look at the set of the two (new and old) populations as a single one, it is obvious that these two ways correspond to tipping the exploration/exploitation balance in one way or another. The introduction of a new random population and the resulting application of the crossover operator would correspond to a hyper or macromutation operator [4,5], tipping the balance towards *exploration*, while a new population generated via application of genetic operators would correspond to an *exploitation* around the point in search space that has actually been reached. In any case, it is quite clear that the result of putting individuals from an existing evolved population in common with a new random one will result in a complex interaction, with varying results depending on the problem: it might be the case that different problems or even different phases in the execution of a problem will need different strategies.

In this paper, our objective is to find out what are the effects of the incorporation of a new population, at different times, into an existing evolution problem, and to eventually propose some heuristic rules to handle it. Our expected result will be some rule of thumb about when the addition of these new populations is most profitable or, in any case, a measure of how high is its influence on the final outcome.

As far as we know, the type of asynchrony this paper deals with has not been analyzed in depth in the existing literature. Certainly, asynchronous distributed genetic algorithms have been discussed extensively, for instance Giacobini et al. studied the selection intensity in asynchronous evolutionary algorithms [6], and Alba et al. compare them with synchronous parallel distributed genetic algorithms in [7]; a similar approach applied to distributed genetic programming was presented in [8]. In general, the conclusion is that asynchrony in evolution does not affect algorithm performance; however, that conclusion applies only if all computing nodes start at the same time, which is not the case that we want to address in this work.

Cantú-Paz [9] found that the migration policy that causes the greatest reduction in total algorithmic work (expressed as total number of evaluations) is to choose as migrants the best individuals and to replace the worse individuals in the destination population, since this policy increases the selection pressure and

may cause the algorithm to converge significantly faster. However, too fast a convergence can lead to the algorithm's failure, as he states referring to parallel EAs: "rapid convergence is desirable, but an excessively fast convergence may cause the EA to converge prematurely to a suboptimal solution". In fact, Alba and Troya [10] found that migration of a random string prevents the "conquest" effect in the target island for small or medium sized sub-populations. In line with this, we study here the trade-off between selection pressure and diversity when we have nodes starting at different times; and since it has been proved the best strategy, the two nodes will migrate the best individual.

The rest of the paper is organized as follows: the experimental setup is described in Section 2, with results presented in Section 3. Finally, conclusions and future work are commented in Section 4.

2 Experimental Setup

Two functions have been used for testing: the problem generator P-Peaks and the massively multimodal deceptive problem (MMDP), two of the three discrete optimization problems presented by Giacobini et al. in [11]. These problems, while being both multimodal, represent different degrees of difficulty for parallel evolutionary optimization, and will be described next.

MMDP [12] is deceptive (that is, approached via hill-climbing algorithms would lead to a suboptimal solution) composed of k subproblems of 6 bits each. Each subproblem is evaluated on the basis of its unitation as follows:

$$f(n) = \begin{cases} 1.0 & n \in \{0, 6\} \\ 0.0 & n \in \{1, 5\} \\ 0.360384 & n \in \{2, 4\} \\ 0.640576 & n = 3 \end{cases}$$

The fitness value of a $6k$-bit string is defined as

$$f_{MMDP}(\boldsymbol{s}) = \sum_{i=0}^{k-1} f\left(\sum_{j=1}^{6} s_{6i+j}\right)$$

Note that the number of local optima is quite large (22^k), while there are only 2^k global solutions. In this paper, we consider a single instance with $k = 20$ (120 bits).

On the other hand, the P-Peaks problem is a multimodal problem generator proposed by De Jong in [13]; a P-Peaks instance is created by generating P random $N - bit$ strings where the fitness value of a string $\boldsymbol{x}$ is the number of bits that $\boldsymbol{x}$ has in common with the nearest peak divided by N.

$$f_{P-PEAKS}(\boldsymbol{x}) = \frac{1}{N} \max_{1 \le i \le p} \{N - H(\boldsymbol{x}, Peak_i)\} \tag{1}$$

where $H(\boldsymbol{x}, \boldsymbol{y})$ is the Hamming distance between binary strings $\boldsymbol{x}$ and $\boldsymbol{y}$. In the experiments made in this paper we will consider $P = 100$ and $N = 64$. Note that the optimum fitness is 1.0.

These two problems have been implemented and integrated in the public-domain `Algorithm::Evolutionary`[14] Perl library[1]. In order to simulate a parallel algorithm, the *cooperative multitasking* Perl module POE[2] has been used; each node is represented by a POE *session*. Thus, in fact, the conclusions obtained in this paper are algorithmic in nature; if runtime conclusions have to be made, this experiment should be repeated in a true parallel environment. In this

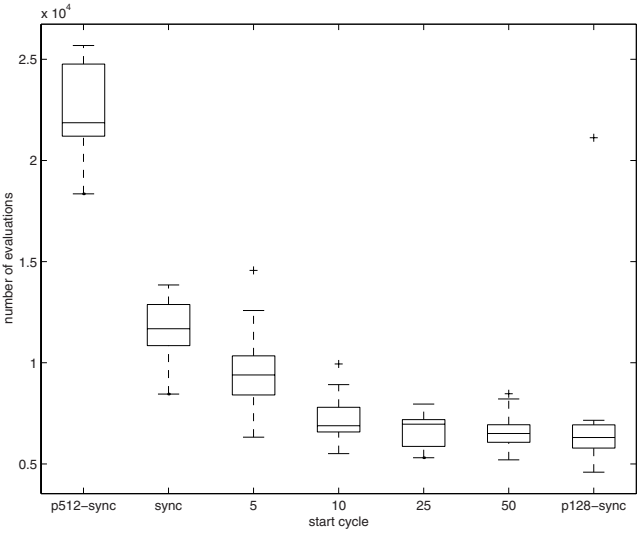

Fig. 1. Boxplot of the number of evaluations needed to find the solution in the P-Peaks problem starting at different cycles. *sync* labels cases with the two nodes starting synchronously: `p512-sync` with a population of 512, `sync` with 256, and `p128-sync` with 128 individuals; `50` represents the behavior of the experiment in a single node, since the algorithms finish before receiving any individual.

simulated parallel scenario we have implemented two nodes, each one applying a rank-based substitution steady state algorithm [15] to a single population. We do not think that using only two nodes represents a loss of generality, since migrations are always performed between only two nodes, independently of how many are running at a time. At the end of a preset number of generations (which we will call a cycle), each node sends a single individual (the best one) to the other in a theoretically synchronous manner (that is, both nodes evolve in lock-step). Algorithmic efficiency will be measured summming up the total number of evaluations performed in each node until the solution is found in one of them.

[1] Freely available under the GPL license from `http://tinyurl.com/3v4gj7`. The program, along with some configuration files and experiment results, can be downloaded from `http://tinyurl.com/4ttaow`; released versions can also be downloaded from your closest CPAN repository.

[2] Perl Object Environment; also available from CPAN.

In order to simulate the asynchronous start of the second population, several experiments were made in which the second population did not start until after a certain number of cycles. Population 1 was left running for n cycles (with g generations each), and then Population 2 started running and interchanging individuals with it. This asynchronous starting point is fixed (does not depend on the state the evolutionary algorithm is), so it could happen that Population 1 has already found the solution.

3 Experimental Results

Every configuration was run 30 times in order to obtain statistically significant results. All experiments were performed in Linux desktop and laptop machines (Ubuntu 7.04 and Fedora Core 6 and 8), with statistical analysis performed using the open source statistical package R.

For the P-Peaks experiment we have chosen the evolutionary algorithm parameters shown in Table 1 (middle column). Figure 1 shows the results of the

Table 1. Evolutionary algorithm parameters used in the P-Peaks experiments. The `Algorithm::Evolutionary` Perl library uses *priorities* for operators, that once normalized, correspond to operator rates: to 40% mutation, 60 % crossover.

Parameter	Value	
	P-Peaks	MMDP
Chromosome length	64	120
Population	256	1024
Selection rate	20%	10%
Generations to migration (cycle size)	10	
Mutation priority	2	
2-point crossover priority	3	

experiments performed with P-Peaks. For the sake of comparison, the total number of evaluations for the synchronous start experiments with population = 512 (leftmost box, labeled `p512-sync`) and population = 128 (rightmost box, labeled `p128-sync`) have also been plotted. Comparing them with the `sync` experiment (2 nodes, population = 256, synchronous start), it can be seen that lowering the population size also improves the number of evaluations (y axis). However, if we start by the `p128-sync` figure and proceed from right to left, we see that splitting the population in two (that is, going from a single population with 256 individuals – start cycle = 50[3] – to two parallel populations with 128 individuals does not yield any improvement) increases the number of evaluations needed to find the solution. Once again, moving from that experiment to its left shows what happens if, instead of letting a single population proceed, we introduce a second population by the 25th cycle (25×10 generations). What we see is a

[3] Which, in fact, would correspond to a single 256 individuals population, since by the 50th cycle, a single population has already reached the target fitness.

decrease in the quality of the algorithm, i.e., an increase of the median number of evaluations needed to reach target. A strikingly similar response is reached if the second population starts any time before that: a higher number of evaluations are going to be needed. Some other conclusions can be reached by looking at this graph from left to right, and starting by the `sync` glyph (which represents the behavior of two populations starting at the same time): whenever the second population is started *after* the first one has already run a bit of its course, the results are going to be better; however, the improvement is going to stall by the time a few cycles have already run (in this case, after the 10th cycle – 100^{th} generation, when around 50% of the runs have already finished). The Wilcoxon rank-sum test confirms that there is no difference among the four last experiments, and that the difference among the three first and the rest is significative.

Let us check these results running again the distributed evolutionary algorithm (parameters shown in Table 1, right-most column) with a more difficult problem, MMDP. The picture is quite different here, although the trend is more or less the same: there is an average trend towards decreasing the number of evaluations when the start cycle of the second population is delayed, which stops when the evolution of the first population is too advanced (in this case, after 50 cycles or 500 generations). However, the situation is not exactly the same. The main difference arises from the fact that there is a non-null set of experiments (among the

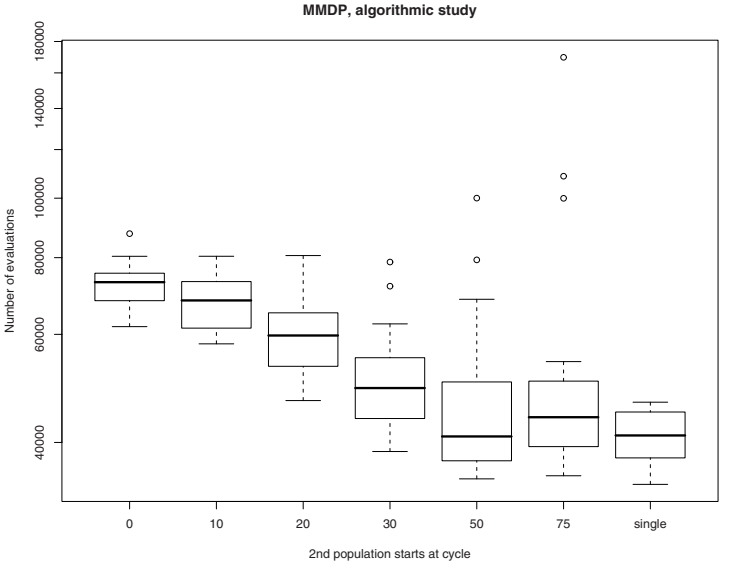

Fig. 2. Logarithmic boxplot of the number of evaluations needed to find the solution in the MMDP, after those that have not found it have been eliminated. x labels indicate the cycle when the second population has been inserted, with "single" indicating results for a single population. Once again, the Wilcoxon rank-sum test confirms the differences among the three first, and its abscense among the 4 last.

30 runs for each parameter set) that does not find the solution before the maximum number of evaluations allowed (200000). The size of this set is represented in Figure 3, which shows a rather jagged scenario, but if we look at it from right to

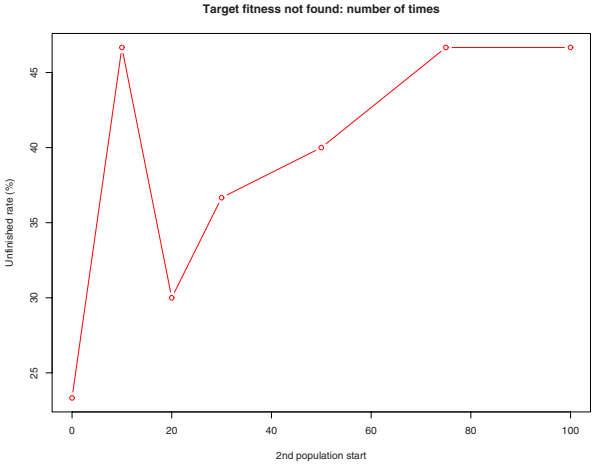

Fig. 3. Percentage of runs, for different start cycles of the second population, where the target fitness was not found in the MMDP problem

left, we see that it confirms the effect of the moment of introduction of the second population on the total quality of results: from a single population (x label $= 100$) to a late introduction of the second population ($x = 75, 50$), there is a very small improvement (from 45% to 40%). The situation gets a bit better if the second population is introduced at cycle # 20 or 30, but worsens again when it is introduced too early (cycle # 10). In this case, the best worst-case scenario is given by the synchronous population, although the best median number of evaluations is found when the second population is introduced at cycle # 30. Putting both effects together, we find that the best situation is in the *intermediate* area: lowest number of evaluations, without an excessive raise in the number of unsuccessful runs (which might be changed if the evaluation limit is set higher).

Find out the reason why this happens is a different problem, and the classical, synchronous start, two-population distributed evolutionary algorithm comes out as a worse algorithm. In order to check what is going on, we did several runs with another program where we logged the diversity after each cycle in the P-Peaks experiment. The results are plotted in Figure 4, which shows the different evolution paths of phenotypic entropy (computed using the Shannon formula) in an asynchronous (left) and synchronous (right) start experiment, in two typical cases that finished in roughly the same amount of cycles (around 30). In time, entropy tends to equalize; however, the level it reaches is that of the less diverse population (Population 1, in both cases). However, it is interesting to see that the highest effect in diversity is that of immigrants of Population 1 on Population 2; in general, the effects of the less diverse (more converged, and thus,

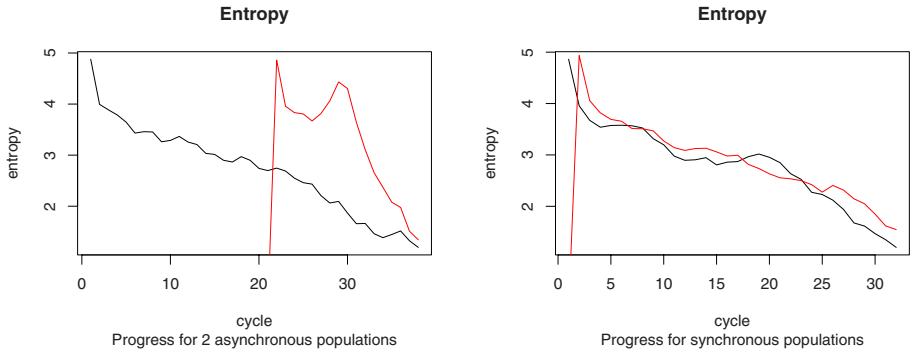

Fig. 4. Shannon Entropy ($H(P) = -\sum_{g \in P} p(f(g)) log_b p(f(g))$, with g a member of the population, $f(g)$ its fitness, and $p(f(g))$ the frequency of that fitness) of both populations in a typical run of the P-Peaks problem, with asynchronous start (right) and second population starting at cycle 20 (left); Population 1 is plotted in black and Population 2 in red or light color. Please note that the total number of evaluations will be lower in the first case, since Population 2 will have performed less evaluations.

further up the evolution ladder) on the more diverse (less evolved) population. The bend found before cycle 30 in both cases indicates a quick exploitation that eventually finds the solution. This leads us to think that the reduction in the number of evaluations is mainly due to the effect of highly-fit individuals falling and eventually mating with a pool of highly-diverse ones. This effect does not take place if both populations start at the same time (diversity and fitness degrees reached are more or less the same across all the experiment), and where exploration and exploitation take place roughly synchronously (more exploitation at the beginning, more exploration at the end); or if one population is introduced too late into the simulation, where the combination of highly fit individual with low-fitness ones will amount to exploration, thus raising the total number of evaluations. However, it is the combination of highly-fit individuals with a diverse population with the right difference in fitness which produces the best algorithmic result in the shape of the best median number of evaluations.

This result is interesting, being roughly in accordance with the Intermediate Disturbance Hypothesis [16], which states that the right amount of disturbance produces the maximum diversity in ecosystems. In this case, low disturbance (migration among similar populations) and high disturbance (in-migration of an individual too highly fit) yields worse results than a better-fit individual introduced a pool of individuals that have already evolved for some time.

4 Conclusions

In this paper we have tested the influence that the introduction of a new population has on the fragile exploitation/exploration equilibrium that reigns in a single population undergoing evolution. We have used a simulated two-node

parallel population, with the second population introduced at different times and tested it on two different discrete optimization problems: P-Peaks and MMDP.

The result has been rather counter-intuitive: introducing a second population a little after the first population always improves the algorithmic efficiency of the set, while doing it close to the end of evolution, as was our *a priori* hypothesis, does no good algorithmically and might even impact negatively on the overall performance. The cause of this effect has been studied and we have concluded that it might be related to the intermediate disturbance hypothesis applied to the second late-coming population (the effect of immigrants from the second population to the first being rather negligible): receiving high-fitness immigrants from the first, already evolved population will be beneficial only if the fitness difference between that immigrant and the current genetic pool is just right. If it is too high (first population highly evolved) or too low (first population started at the same time), the increase in diversity (and thus the speedup in finding the solution) in this second population will be negligible.

This yields the rule of thumb that additional populations should be started later at regular (short) intervals, instead of at the same time; and that if there is a new node arriving in a distributed computation experiment, it should be used for *outsourcing*, by doing just fitness evaluations or some other expensive task, and not for *offshoring*, by spawning a whole new population that will perform its own evolution in parallel.

In the future, we will try to confirm the results obtained in these simulations by applying it to more discrete and continuous optimization experiments, and also using more simultaneous populations, although this addition should not essentially alter the results. It would be also interesting to test them in a real parallel environment, to match not only the algorithmic gain, but also the time gain obtained by starting two populations asynchronously and in heterogeneous computers. In principle, the effect of having a second population in a slow computer would be akin to having a late-start second population, and the working hypothesis would be that the effect could be beneficial if the performance differences are not too high, but this would have to be tested. Eventually, our intention is to create a distributed computation framework that would self-adapt to late comers, asynchrony and differences in performance extracting the most from it.

References

1. Laredo, J., Castillo, P., Mora, A., Merelo, J.: Exploring population structures for locally concurrent and massively parallel evolutionary algorithms. In: [17], pp. 2610–2617
2. Merelo, J.J., García, A.M., Laredo, J.L.J., Lupión, J., Tricas, F.: Browser-based distributed evolutionary computation: performance and scaling behavior. In: GECCO 2007: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation, pp. 2851–2858. ACM Press, New York (2007)
3. Merelo, J., Castillo, P., Laredo, J., Mora, A., Prieto, A.: Asynchronous distributed genetic algorithms with Javascript and JSON. In: [17], pp. 1372–1379

4. Morrison, R., De Jong, K., Syst, M., McLean, V.: Triggered hypermutation revisited. In: Proceedings of the 2000 Congress on Evolutionary Computation, vol. 2 (2000)
5. Jones, T.: Crossover, macromutation, and population-based search. In: Proceedings of the Sixth International Conference on Genetic Algorithms, pp. 73–80. Morgan Kaufmann, San Francisco (1995)
6. Giacobini, M., Alba, E., Tomassini, M., et al.: Selection intensity in asynchronous cellular evolutionary algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference, Chicago, USA, pp. 955–966 (2003)
7. Alba, E., Troya, J.: Analyzing synchronous and asynchronous parallel distributed genetic algorithms. Future Generation Computer Systems 17(4), 451–465 (2001)
8. Fernandez, F., Galeano, G., Gomez, J.: Comparing Synchronous and Asynchronous Parallel and Distributed Genetic Programming Models. In: Foster, J.A., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A.G.B. (eds.) EuroGP 2002. LNCS, vol. 2278. Springer, Heidelberg (2002)
9. Cantú-Paz, E.: Migration policies, selection pressure, and parallel evolutionary algorithms. Journal of Heuristics 7(4), 311–334 (2001)
10. Alba, E., Troya, J.M.: Influence of the migration policy in parallel distributed GAs with structured and panmictic populations. Appl. Intell. 12(3), 163–181 (2000)
11. Giacobini, M., Preuss, M., Tomassini, M.: Effects of scale-free and small-world topologies on binary coded self-adaptive CEA. In: Gottlieb, J., Raidl, G.R. (eds.) EvoCOP 2006. LNCS, vol. 3906, pp. 85–96. Springer, Heidelberg (2006)
12. Goldberg, D.E., Deb, K., Horn, J.: Massive multimodality, deception, and genetic algorithms. In: Männer, R., Manderick, B. (eds.) Parallel Problem Solving from Nature, vol. 2. Elsevier Science Publishers, B. V, Amsterdam (1992)
13. Jong, K.A.D., Potter, M.A., Spears, W.M.: Using problem generators to explore the effects of epistasis. In: Bäck, T. (ed.) Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA 1997). Morgan Kaufmann, San Francisco (1997)
14. Merelo-Guervós, J.J.: Evolutionary computation in Perl. In: Perl Mongers, M. (ed.) YAPC, Europe 2002, pp. 2–22 (2002)
15. Syswerda, G.: A Study of Reproduction in Generational and Steady-State Genetic Algorithms. Foundations of Genetic Algorithms (1991)
16. Ward, J., Stanford, J.: Intermediate-Disturbance Hypothesis: An Explanation for Biotic Diversity Patterns in Lotic Ecosystems. Dynamics of Lotic Systems, Ann Arbor Science, Ann Arbor MI. 1983. 347-356 p, 2 fig, 35 ref. (1983)
17. IEEE Congress on Evolutionary Computation (CEC2008). In: WCCI 2008 Proceedings. IEEE Press, Los Alamitos (2008)