ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

BioSystems 76 (2004) 209–216

BioSystems

www.elsevier.com/locate/biosystems

Coevolving solutions to the shortest common superstring problem

Assaf Zaritsky*, Moshe Sipper

Department of Computer Science, Ben-Gurion University, P.O. Box 653, Beer-Sheva 84105, Israel

Received 28 February 2003; received in revised form 11 July 2003; accepted 1 August 2003

Abstract

The shortest common superstring (SCS) problem, known to be NP-Complete, seeks the shortest string that contains all strings from a given set. In this paper we compare four approaches for finding solutions to the SCS problem: a standard genetic algorithm, a novel cooperative-coevolutionary algorithm, a benchmark greedy algorithm, and a parallel coevolutionary-greedy approach. We show the coevolutionary approach produces the best results, and discuss directions for future research.

Keywords: Coevolutionary algorithms; Shortest common superstring; Cooperative Coevolution

1. Introduction

In recent years we have been witness to the application of bio-inspired algorithms to the solution of a plethora of hard problems in computer science Sipper (2002). One such bio-inspired methodology—evolutionary algorithms—we apply herein to the NP-Complete problem known as the *shortest common superstring* (SCS). The SCS problem seeks the shortest string that contains all strings from a given set. Finding the shortest common superstring has applications in data compression Storer (1988), because data may be stored efficiently as a superstring. SCS also has important applications in computational biology Lesk (1988), where the DNA-sequencing problem is to map a string of DNA: laboratory techniques exist for reading relatively short strands of DNA; to map a longer sequence, many copies are made, which are then cut into smaller overlapping sequences that can be mapped. A typical approach is to reassemble them by finding a short (common) superstring. The SCS problem, which is NP-Complete Garey and Johnson (1979), is also MAX-SNP hard Blum et al. (1994). It is conjectured that no polynomial-time algorithm exists, that can approximate the optimum to within a predetermined constant. In this paper we compare four approaches for finding solutions to the SCS problem: a standard genetic algorithm (GA), a novel cooperative-coevolutionary algorithm, a benchmark greedy algorithm, and a parallel coevolutionary-greedy approach. We show that our coevolutionary approach is best when considering sets containing approximately 50–80 strings. This paper is organized as followsi the next section we present previous work on the SCS problem and on cooperative coevolution. Section 3 describes the three algorithms applied in Section 4 to the SCS problem. In Section 5 we combine said algorithms to obtain better results. Finally, we present concluding remarks and suggestions for future work in Section 6.

* Corresponding author. Tel.: +972 8 647 7820, fax: +972 8 647 7650.
E-mail address: assafza@cs.bgu.ac.il (A. Zaritsky).

0303-2647/$ – see front matter © 2004 Elsevier Ireland Ltd. All rights reserved.
doi:10.1016/j.biosystems.2004.05.013

2. Preliminaries and previous work

2.1. The shortest common superstring problem

Let $S = \{s_1, \dots, s_n\}$ be a set of strings (denoted *blocks*) over some alphabet Σ. A *superstring* of S is a string s such that each s_i in S is a substring of s. A trivial (and usually not the shortest) solution is the concatenation of all blocks, namely, $s_1 \cdots s_n$. For two strings u and v let $overlap(u, v)$ be the maximum overlap between u and v, i.e., the longest suffix of u (in terms of characters) that is a prefix of v; let $prefix(u, v)$ be the prefix of u obtained by removing its overlap with v; let $merge(u, v)$ be the concatenation of u and v with the overlap appearing only once. For example, given the alphabet $\Sigma = \{a, b, c\}$ and a set of strings $S = \{cbca, cac\}$, the shortest common superstring of S is the string *cbcac*. A longer superstring would be *cacbcab*. In addition, the following relations hold:

$$overlap(cbca, cac) = ca,$$
$$overlap(cac, cbca) = c,$$
$$prefix(cbca, cac) = cb,$$
$$merge(cac, cbca) = cacbca.$$

Note that, in general, $overlap(A, B) \not\equiv overlap(B, A)$ (the same holds for *prefix* and *merge*). Given a list of blocks $s_1, s_2, \dots, s_n$, we define the *superstring* $s = \langle s_1, s_2, \dots, s_n \rangle$ to be the string $prefix(s_1, s_2) \cdot prefix(s_2, s_3) \dots .prefix(s_n, s_1) \cdot overlap(s_n, s_1)$. To wit, *superstring* is the concatenation of all strings, “minus” the overlapping duplicates. Each superstring of a set of strings defines a permutation of the set’s elements (the order of their appearance in the superstring), and every permutation of the set’s elements corresponds to a single superstring (derived by applying the {lit superstring} operator). Several linear approximations for the SCS problem have been proposed. Blum et al. (1994) were the first to introduce an approximation algorithm that produces a solution within a factor of 3 from the optimum. The best factor currently known is 2.5, and was achieved by Sweedyk (1999). Our bibliography research revealed no application of evolutionary algorithms to the SCS problem.

2.2. Cooperative coevolution

Coevolution refers to the simultaneous evolution of two or more species with coupled fitness. Such coupled evolution favors the discovery of complex solutions whenever complex solutions are required Paredis (1995). Simplistically speaking, one can say that coevolving species either compete (e.g., to obtain exclusivity on a limited resource) or cooperate (e.g., to gain access to some hard-to-attain resource). In a competitive coevolutionary algorithm the fitness of an individual is based on direct competition with individuals of other species, which in turn evolve separately in their own populations. Increased fitness of one of the species implies a diminution in the fitness of the other species. This evolutionary pressure tends to produce new strategies in the populations involved so as to maintain their chances of survival. This “arms race” ideally increases the capabilities of each species until they reach an optimum. For further details on competitive coevolution, the reader is referred to Rosin and Belew (1997). Cooperative (also called symbiotic) coevolutionary algorithms involve a number of independently evolving species, which together form complex structures, well-suited to solving a problem. The fitness of an individual depends on its ability to collaborate with individuals from other species. In this way, the evolutionary pressure stemming from the difficulty of the problem favors the development of cooperative strategies and individuals. Single-population evolutionary algorithms often perform poorly—manifesting stagnation, convergence to local optima, and computational costliness—when confronted with problems presenting one or more of the following features Peña-Reyes and Sipper (2000, 2001b): (1) the sought-after solution is complex, (2) the problem or its solution is clearly decomposable, (3) the genome encodes different types of values, (4) strong interdependencies among the components of the solution, (5) components-ordering drastically affects fitness. Cooperative coevolution addresses effectively these issues, consequently widening the range of applications of evolutionary computation. citetparedis95 applied cooperative coevolution to problems which involved finding simultaneously the values of a solution and their adequate order. In his approach, a population of solutions coevolves alongside a population of permutations performed on the genotypes of the solutions. Potter (1997) and Potter and De Jong (2000) developed a model in which a number of populations explore different decompositions of the problem. More recently, Peña-Reyes

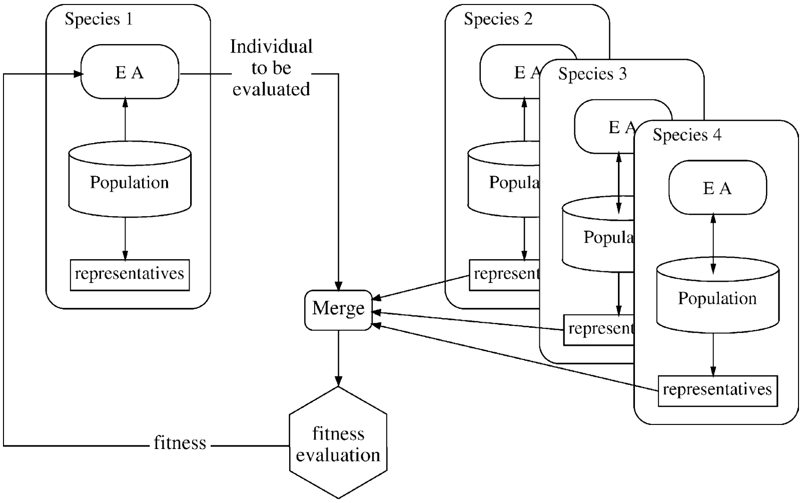

Fig. 1. Potter's cooperative coevolutionary system. The figure shows the evolutionary process from the perspective of Species 1. The individual being evaluated is combined with one or more *representatives* of the other species so as to construct several solutions which are tested on the problem. The individual's fitness depends on the quality of these solutions.

and Sipper (2000, 2001b,a) used cooperative coevolution to evolve fuzzy systems. Below we detail the framework of Potter and DeJong as it forms the basis of our own approach. In Potter's system, each species represents a subcomponent of a potential solution. Complete solutions are obtained by assembling *representative* members of each of the species (populations). The fitness of each individual depends on the quality of (some of) the complete solutions it participated in, thus measuring how well it cooperates to solve the problem. The evolution of each species is controlled by a separate, independent evolutionary algorithm. Fig. 1 shows the general architecture of Potter's cooperative coevolutionary framework, and the way each evolutionary algorithm computes the fitness of its individuals by combining them with selected representatives from the other species. The representatives can be selected via a greedy strategy as the fittest individuals from the last generation. Results presented by Potter and De Jong (2000) show that their approach addresses adequately issues like problem decomposition and interdependencies between subcomponents. The cooperative coevolutionary approach performs as good as, and sometimes better than, single-population evolutionary algorithms. Finally, cooperative coevolution usually requires less computation than single-population evolution as the populations involved are smaller, and convergence—in terms of number of generations—is faster.

3. Description of the three algorithms

This section describes the three algorithms—genetic, cooperative coevolutionary, and greedy—used in the next section to seek solutions to the SCS problem.

3.1. The genetic algorithm

Given a set of (binary) strings as an input to the SCS problem, the algorithm generates an initial population of random candidate solutions, the fitness of each individual depending on its length and accuracy. As is standard, the genetic algorithm uses selection, crossover, and mutation to evolve the next generation, each individual of which is then evaluated and assigned a fitness value. These steps are repeated a predefined number of times or until the solution is satisfactory. The members (strings, or *blocks*) of the input set are atomic components as far as the GA is concerned, namely, there is no change—either via crossover or mutation—within a block, only between blocks (i.e., their order changes). An individual in the population is a candidate solution to the SCS problem, its genome represented as a sequence of blocks. An individual may contain missing blocks or duplicate copies of the same block. The genetic algorithm seeks to evolve an individual that is as *close* as possible to the shortest common superstring (the *closeness* relation is defined below). Each individual derives a corresponding superstring, by applying the *superstring* relation (Section 2.1) on the blocks within the genome (this is called the *derived* string). The *fitness* of an individual is a function of two parameters: length of derived string (shorter is better), and number of blocks it contains (more is better); thus, the goal is to maximize the number of blocks "covered" and to minimize the length of the derived string. Other parameters such as average (or maximal) block length might also be considered. We used standard fitness-proportionate selection, with an elitism rate of 1 (i.e., the best individual is always copied to the next generation). Two-point crossover was applied, wherein two crossover points are chosen randomly (but at block boundaries), the offspring being composed of the first parent's flanks with the second parent's interior. Crossover allows both growth and reduction in an individual genome's length. Mutation occurs with low probability, exchanging a block with a randomly chosen block. *A note about representation*: We mentioned earlier that the shortest common superstring of a set of blocks is a permutation of the set's elements. The major deficiency of our genomic representation is its affording a block to appear more than once. This transforms the problem from one of finding a permutation, with a search space of size *n!*, where *n* is the number of blocks, to a problem in which the search space is much larger. We have, nonetheless, chosen this representation due to two reasons:

(1) Despite being a permutation on the set of blocks (i.e., each block appears exactly once), the SCS might have many near-optimal solutions that contain repeated instances of the same block. We have found that the GA easily filters out most solutions containing repeated blocks, and yields solutions that are close to permutations. In addition, the solution's length is flexible, allowing the progressive construction of building blocks.
(2) This representation enables the use of other GA techniques, e.g., bit-flip mutations. When flipping a bit in a permutation representation (meaning, in our case, exchanging one block for another) an individual loses its permutation property, necessitating a repair mechanism. In the present representation such a problem does not occur. Moreover, cooperative coevolution can easily be applied here, as explained below, in contrast to a permutation representation.

3.2. *The coevolutionary algorithm*

The second algorithm is based on cooperative coevolution, wherein two species evolve simultaneously. The first species contains prefixes of candidate solutions to the SCS problem at hand, and the second species contains candidate suffixes. The fitness of an individual in each of the species depends on how good it interacts with representatives from the other species to construct the global solution (Section 2.2). Each species evolves separately (i.e., selection, crossover, and mutation are performed independently) while the only interaction is through the fitness function. Each species nominates its fittest individual as the *representative*. When computing the fitness of an individual in the prefix (suffix) population, its genome is simply concatenated with the representative of the suffix (prefix) population, to construct a full candidate solution for the SCS problem at hand; this solution is then evaluated in the same manner as described in Section 3.1. Basically, all individuals of one population are combined with the best individual of the second population, the resulting fitness values assigned to the first-population individuals. Crossover and mutation, applied in each population, are identical to those described in Section 3.1.

3.3. *The greedy algorithm*

The greedy algorithm repeatedly merges pairs of distinct strings with maximum *overlap* (Section 2.1) until only one string remains. This algorithm finds an approximate solution within factor 4 of the optimum Blum et al. (1994). A common conjecture states that the superstring produced by the greedy algorithm is of length at most two times the optimal Storer (1988), Tarhio and Ukkonen (1988), Turner (1989). This is why we chose to implement it rather than Sweedyk's Sweedyk (1999) factor-2.5 algorithm: the latter is *much* more complex and most likely does not entail a bona fide benefit (since the greedy algorithm is most likely to be factor-2). Moreover, the simple greedy algorithm is probably the most widely used heuristic used in DNA-sequencing.

4. Results

The following section describes results for 50-block sets, and 80-blocks sets. We first discuss an issue related to the input blocks.

4.1. The input

All experiments were performed using a binary alphabet. The results were compared to those of the greedy algorithm that pairs up blocks according to their maximal overlap. The input strings used in the experiments were generated in a manner similar to the one used in DNA sequencing (Section 1): A random string is generated, duplicated a predetermined number of times, and the copies are randomly divided into blocks of a given size. The set of all these blocks is the input to the SCS problem. The process is explained in Fig. 2. Note that the SCS of such a set is not necessarily the original string (it may be shorter), though it is likely to be close to it due to the original string's randomness. We chose to generate such inputs for a number of reasons. First, our interest in a real-world application, namely DNA sequencing. Second, this input domain is interesting because there are many large overlapping blocks, thus rendering difficult the decision of choosing and ordering the blocks needed to construct a short superstring. Lastly, the length of a SCS of a set of blocks drawn from this particular input domain is with very high probability, simply the length of the initial string (in the input generation process). This enables us to generate many different problems, all with a predetermined SCS length. We performed two experiments differing only in the length of the initial randomized string generated. This causes a difference in the number of blocks given as input to the algorithm. The parameters used in the input-generation phase are as given further.

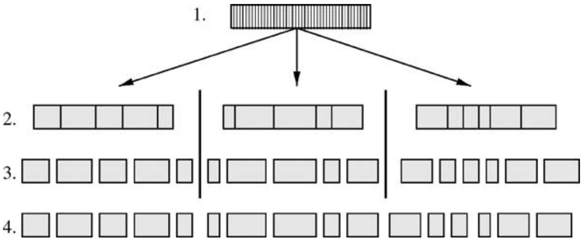

Fig. 2. The input-generation procedure. A random string is generated (1). The string is duplicated a predetermined number of times (three, in the above example), each copy divided into blocks of a random size between *minimal block size* (20, in our case) and *maximal block size* (30, in our case) (2). The resulting set of blocks (3) is the input set (4).

- *Size of random string*: 250 bits (experiment I), 400 bits (experiment II).
- *Minimal block size*: 20 bits.
- *Maximal block size*: 30 bits.
- *Number of duplicates created from random string*: 5.

Note that increasing the number of blocks (through whatever parameter change) results in exponential growth of the problem's complexity. The evolutionary parameters used are as given further.

- *Population size*: 500.
- *Number of generations*: 5000.
- *Crossover rate*: 1 (i.e., crossover always performed between a selected pair).
- *Mutation rate*: 0.03.
- *Problem instances per experiment*: 50.

Let l denote the length of the *derived* string in the genetic-algorithm case (Section 3.1), or the combined *derived* string (individual + representative) in the cooperative-coevolutionary case (Section 3.2). Let m denote the number of blocks *not* covered by the derived string, and let b denote the maximal block size (30, in our case). The fitness value, f, of an individual is computed as follows:

$$f = \frac{1}{(l + mb)^{\alpha}}$$

α was set empirically to $\alpha = 2$ after preliminary test runs, for both the genetic and coevolutionary algorithms. This fitness function drives evolution towards shorter superstrings covering as many blocks as possible. The experiments described in the next two subsections each applied the three algorithms described in Section 3 to a given problem size. On each problem

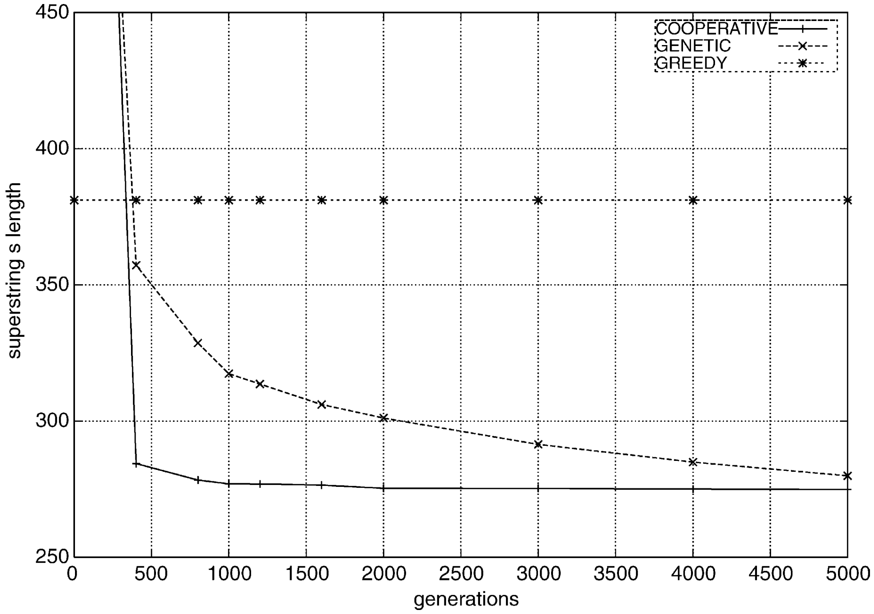

Fig. 3. Experiment I: 50 blocks. Best superstring as a function of time (generations). Each point in the figure represents the average of 50 runs on 50 different randomly generated problem instances. For each such instance, two runs were performed (i.e., a total of 100), the better of which was considered for statistical purposes. Shown are results for all three algorithms: cooperative coevolution (COOPERATIVE), genetic algorithm (GENETIC), greedy algorithm (GREEDY).

instance each type of genetic algorithm was executed twice and the better run of the two was used for statistical purposes. (As argued by Sipper (2000) what ultimately counts in problem solving by an evolutionary algorithm is the *best* result.)

4.2. Experiment I: 50 blocks

The results presented in Fig. 3 show the average length of the superstrings found when the input set contained 50 blocks (generated as explained in Fig. 2). Both genetic algorithms dramatically outperformed the greedy approach. The cooperative coevolutionary algorithm converges much faster than the simple GA, leveling off in less than 1000 generations (approximately 15 min on a 667 MHz PC).

4.3. Experiment II: 80 blocks

The results presented in Fig. 4 show the average length of the superstrings found when the input set contained 80 blocks. The cooperative coevolutionary algorithm again wins top marks, whereas the simple GA comes last. The cooperative coevolutionary algorithm converges more slowly this time, and continues to improve right up to the last generation (set at 5000). Note that although the complexity of the search space here is much larger than in experiment I, we set the maximal computational effort at the same level (namely, 5000 generations); a decrease in the quality of solutions was therefore anticipated. And yet, our cooperative approach still comes out on top.

5. Coevolution, parallelism, and greed

Reflecting upon the results we designed an improved coevolutionary algorithm, which incorporates both parallelism and greed. The algorithm consists of three stages:

(1) The first stage consists of three parallel runs of the cooperative coevolutionary algorithm—executed independently—with *number of generations* set to 2000 (instead of 5000), and *population size*

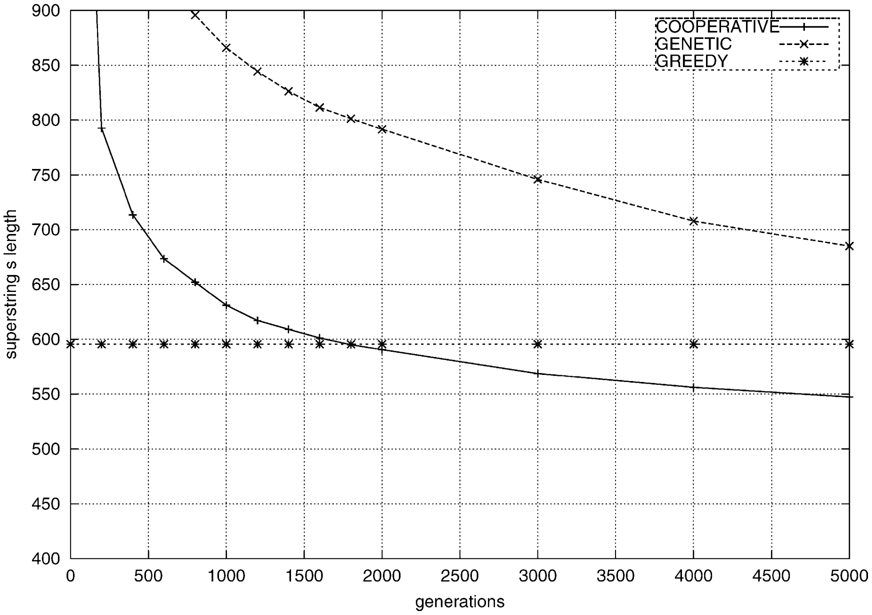

Fig. 4. Experiment II: 80 blocks. Best superstring as a function of time (generations). Each point in the figure represents the average of 50 runs on 50 different randomly generated problem instances. For each such instance, two runs were performed, the better of which was considered for statistical purposes. Shown are results for all three algorithms: cooperative coevolution (COOPERATIVE), genetic algorithm (GENETIC), greedy algorithm (GREEDY).

set to 300 (instead of 500); all other parameters remain unchanged. In addition, the greedy algorithm is run (ending deterministically as per the algorithm). At the end of this stage there are three evolved prefix populations, three evolved suffix populations, and a greedy solution.

(2) Two new populations are constructed: (a) a prefix population consisting of one third of the individuals of each evolved prefix population (stage 1), chosen fitness-proportionately (in their respective populations); (b) an analogously created suffix population. The greedy solution is split in the middle, each half added to the appropriate population.

(3) The cooperative coevolutionary algorithm is run with the two populations created in stage 2 serving as initial populations (again, with *number of generations* set to 2000).

We tested the combined algorithm on the 25 80-block problem instances that were hardest for the greedy algorithm (i.e., on which its performance was the worst). Table 1 shows that the results obtained by our combined approach are best.

Table 1
Best average results

Problem size	Greedy	Genetic	Cooperative	Parallel
(a)				
50	381	280	275	
80	596	685	547	
(b)				
80	625	683	542	510

(a) Obtained by the three algorithms (Section 4): greedy, genetic, and cooperative coevolution. For each of the 50 randomly generated problem instances each algorithm was run twice, the worse of the two discarded; average is, thus, over 50 runs. (b) Best average results obtained by the four algorithms (previous three + parallel combined coevolution) on the 25 hardest problems for the greedy algorithm (Section 5). Again, two runs were performed per problem instance, and the worse of the two runs discarded. Note that in this case the cooperative coevolutionary algorithm and the parallel one surpass the greedy and genetic algorithms by a much more impressive margin than in (a). As can be seen, the parallel combined algorithm is the best.

6. Concluding remarks and future work

We applied four algorithms to the shortest common superstring problem. The results are summarized in

Table 1. As can be seen, cooperative coevolution has revealed itself as a powerful tool, surpassing both the genetic algorithm and the greedy algorithm. Combining the latter with coevolution yields even better results. These results, although preliminary, have encouraged us to consider two major lines of further research.

- Tackling larger problem instances using an improved version of the coevolutionary algorithm (and, possibly, of the combined algorithm).
- Another possible improvement would be to construct new species on the fly as convergence is encountered (as suggested by Potter and De Jong (2000)).

References

Blum, A., Jiang, T., Li, M., Tromp, J., Yannakakis, M., 1994. Linear approximation of shortest superstrings. Journal of the ACM 41, 634–647.

Garey, M., Johnson, D., 1979. Computers and Intractability. Freeman, New York.

Lesk, A., 1988. Computational Molecular Biology. Sources and Methods for Sequence Analysis. Oxford University Press, Oxford.

Paredis, J., 1995. Coevolutionary computation. Artificial Life 2 (4), 355–375.

Peña-Reyes, C.-A., Sipper, M., 2000. Applying fuzzy CoCo to breast cancer diagnosis. In: Proceedings of the 2000 Congress on Evolutionary Computation (CEC00), vol. 2. IEEE Press, Piscataway, NJ, pp. 1168–1175.

Peña-Reyes, C.-A., Sipper, M., 2001a. The flowering of fuzzy CoCo: evolving fuzzy iris classifiers. In: Proceedings of the Fifth International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA 2001). Springer-Verlag, Vienna, pp. 304–307.

Peña-Reyes, C.-A., Sipper, M., 2001b. Fuzzy CoCo: a cooperative-coevolutionary approach to fuzzy modeling. IEEE Trans. Fuzzy Systems 9 (5), 727–737.

Potter, M.A., 1997. The Design and Analysis of a Computational Model of Cooperative Coevolution. George Mason University.

Potter, M.A., De Jong, K.A., 2000. Cooperative coevolution: an architecture for evolving coadapted subcomponents. Evolution. Comput. 8 (1), 1–29.

Rosin, C.D., Belew, R.K., 1997. New methods for competitive coevolution. Evolution. Comput. 5 (1), 1–29.

Sipper, M., 2000. A Success story or an old wives' tale? On judging experiments in evolutionary computation. Complexity 5 (4), 31–33.

Sipper, M., 2002. Machine Nature: The Coming Age of Bio-Inspired Computing. McGraw-Hill, New York.

Storer, J., Data Compression: Methods and Theory. Computer Science Press.

Sweedyk, Z., 1999. A 2.5-approximation algorithm for shortest superstring. SIAM J. Computing 29 (3), 954–986.

Tarhio, J., Ukkonen, E., 1988. A Greedy Approximation Algorithm for Constructing Shortest Common Superstring Problem. Theoretical Computer Science 57, 131–145.

Turner, J., 1989. Approximation algorithms for the shortest common superstring problem. Inform. Comput. 83, 1–20.