The Preservation of Favored Building Blocks in the Struggle for Fitness: The Puzzle Algorithm

Assaf Zaritsky and Moshe Sipper

***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 present a novel coevolutionary algorithm—the *Puzzle Algorithm*—where a population of building blocks coevolves alongside a population of solutions. We show experimentally that our novel algorithm outperforms a standard genetic algorithm (GA) and a benchmark greedy algorithm on instances of the SCS problem inspired by deoxyribonucleic acid (DNA) sequencing. We next compare our previously presented cooperative coevolutionary algorithm with the *Co-Puzzle Algorithm*—the puzzle algorithm coupled with cooperative coevolution—showing that the latter proves to be top gun. Finally, we discuss the benefits of using our puzzle approach in the general field of evolutionary algorithms.**

***Index Terms*—Building blocks, coevolutionary algorithms, cooperative coevolution, genetic algorithms (GAs), recombination loci, shortest common superstring problem.**

I. 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 [2]. One such popular bio-inspired methodology is evolutionary algorithms, which 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 [3], because data may be stored efficiently as a superstring. SCS also has important applications in computational biology [4], where the deoxyribonucleic acid (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 input domain used throughout this article was inspired by this process.

The SCS problem, which is NP-complete [5], is also MAX-SNP hard [6]. The latter means that if $P \neq \mathrm{NP}$ no polynomial-time algorithm exists, which can *approximate* the optimum to within a given (constant) factor.

In this paper, we present a novel coevolutionary algorithm—the *puzzle algorithm*—wherein a population of building blocks coevolves alongside a population of candidate solutions.

Manuscript received November 20, 2003; revised April 2, 2004.

The authors are with the Department of Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel (e-mail: assafza@cs.bgu.ac.il; sipper@cs.bgu.ac.il).

Digital Object Identifier 10.1109/TEVC.2004.831260

We show that the addition of a building-blocks population to a standard evolutionary algorithm results in notably improved performance on the hard SCS problem. We compare the performance of five algorithms on finding solutions to the SCS problem, on an input domain inspired by DNA sequencing: a standard genetic algorithm (GA), a cooperative coevolutionary algorithm, a benchmark greedy algorithm, the puzzle algorithm, and a combination of cooperative coevolution and the puzzle algorithm—co-puzzle—the latter of which is shown to produce the best results for large input sets.

This paper is organized as follows. In Section II, we present previous work on the SCS problem and on cooperative coevolution, describe the input domain, and delineate the experimental design for all our experiments. Section III describes the puzzle algorithm and compares it with a standard GA. In Section IV, we present the co-puzzle algorithm—the puzzle algorithm combined with cooperative coevolution—and compare it with a cooperative coevolutionary algorithm. Section V discusses the advantages of cooperation and the puzzle approach in the general field of evolutionary algorithms. Section VI proposes a possible future extension of the puzzle algorithm based on messy genetic algorithms. Finally, we present concluding remarks and suggestions for additional future work in Section VII.

II. Background and Previous Work

A. Shortest Common Superstring (SCS) Problem

Let $S = \{s_1, \ldots, s_n\}$ be a set of strings (denoted *blocks*) over some alphabet Σ. Without loss of generality, we assume that the set S is "substring-free" in that no string $s_i \in S$ is a substring of any other $s_j \in S$. 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.

As an example, consider the following (simple) case.

- Given:
  — alphabet $\Sigma = \{a, b, c\}$;
  — set of strings $S = \{cbcaca, cacac\}$.
- Shortest common superstring (SCS) of S: $cbcacac$.
- A longer superstring: $cacacbcaca$.
- The following relations hold:
  — $overlap(cbcaca, cacac) = caca$;
  — $overlap(cacac, cbcaca) = c$;

IEEE Proof

1089-778X/04$20.00 © 2004 IEEE

GREEDY(S)

parameter(s): S – set of blocks
output: superstring of set S

while $\|S\| > 1$
do $\begin{cases} \textit{choose } s_1, s_2 \in S \textit{ such that } overlap(s_1, s_2) \textit{ is maximal} \\ S \leftarrow (S \setminus \{s_1, s_2\}) \cup \{merge(s_1, s_2)\} \end{cases}$
return $(\textit{remaining string in } S)$

Fig. 1. Pseudocode of GREEDY algorithm.

— $prefix(cbcaca, cacac) = cb$;
— $merge(cbcaca, cacac) = cbcacac$.

Note that, in general, $overlap(A, B) \neq overlap(B, A)$ (the same holds for $prefix$ and $merge$). Given a list of blocks $s_1, s_2, \ldots, s_n$, we define the *superstring* $s = \langle s_1, s_2, \ldots, s_n \rangle$ to be the string $prefix(s_1, s_2) \cdot prefix(s_2, s_3) \ldots . prefix(s_n, s_1) \cdot overlap(s_n, s_1)$. 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 *superstring* operator).

A number of linear approximations for the SCS problem have been described in the literature. Blum *et al.* [6] were the first to introduce an approximation algorithm that produces a solution within a factor of 3 of the optimum (i.e., the superstring found is at most three times the length of the shortest common superstring). The factor has been successively improved to 2 8/9, 2 5/6, 2 50/63, 2 3/4, 2 2/3, and 2.596 (see, respectively, [7]–[12]). The best factor currently known is 2 1/2, and was achieved by Sweedyk [13].

A simple greedy algorithm—denoted GREEDY—is probably the most widely used heuristic in DNA sequencing (our domain of predilection). GREEDY repeatedly merges pairs of distinct strings with maximal *overlap* until a single string remains (see Fig. 1 for the formal pseudocode). It is an open question as to how well GREEDY approximates the shortest common superstring, although a common conjecture states that the algorithm produces a solution within factor 2 of the optimum [6], [13], [14] (Blum *et al.* [6] proved the factor-4-ness of GREEDY).

Frieze and Szpankowski [15] proved that when the input set comprises independently generated random blocks (resulting in very little interblock overlap), some greedy-type algorithms are asymptotically optimal. However, where DNA sequencing is concerned (the motivation for the input set, we study herein), blocks are not created independently, and interblock overlap is large (see Section II-C for details). Indeed, GREEDY is known to be nonoptimal where real-DNA sequencing is concerned.

So, which of the above algorithms is best? This question is highly relevant, as it pertains directly to the choice of algorithm(s) with which to compare our novel approaches. Indeed, reviewers of our previous paper [1] were somewhat critical of our finally choosing GREEDY as the sole benchmark (a choice we repeat herein). Why not use the factor-3 or factor-2 1/2 algorithms? Because, we argue, the proven bounds do not relate directly to their (actual) relative performance (at least on the input domain which interests us).

1) Counter examples given by Blum *et al.* [6] show that neither their algorithm nor GREEDY is ultimately **The Best**.
2) We compared the performance of GREEDY and Blum *et al.*'s factor-3 algorithm on our input domain, our results showing that both algorithms perform similarly with a slight advantage for GREEDY.
3) The fact that GREEDY is by far the most popular algorithm used by DNA sequencers, and the widely accepted conjecture regarding its factor-2-ness speak loudly in favor of GREEDY.
4) We believe that in designing his algorithm Sweedyk was mainly interested in improving the theoretical upper bound rather than in designing a truly workable algorithm of practical relevance.
5) The use of GREEDY in this paper is mostly as a yardstick conforming to that used by us in [1]. Our main point herein is the added power of the Puzzle approach *vis-a-vis* a standard GA.

Due to the above reasons, and since Sweedyk's factor-2 1/2 algorithm is *much* more complicated to implement with little to no practical benefit, simple GREEDY (Fig. 1) is quite sufficient for our purposes.

B. 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 [16]. Simplistically speaking, one can say that coevolving species either compete [17], [18], or cooperate, the cooperative type being of interest to us herein.

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 idea is to use several independently maintained populations (species), each specialized to a niche of the complete problem, with the fitness of an individual depending on its ability to collaborate with individuals from other species to construct a global solution (no individual within a single species comprises a solution to the problem at hand—all species must cooperate). A number of cooperative coevolutionary algorithms have been presented in recent years [16], [19]–[23].

Potter [19] and Potter and De Jong [24] developed a model in which a number of populations explore different decompositions of the problem. Below, we detail their framework as we build upon it later in the article.

In Potter's and De Jong's system, each species represents a subcomponent of a potential (global) 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 has 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. 2 shows the general architecture of Potter's and De Jong's cooperative

IEEE Proof

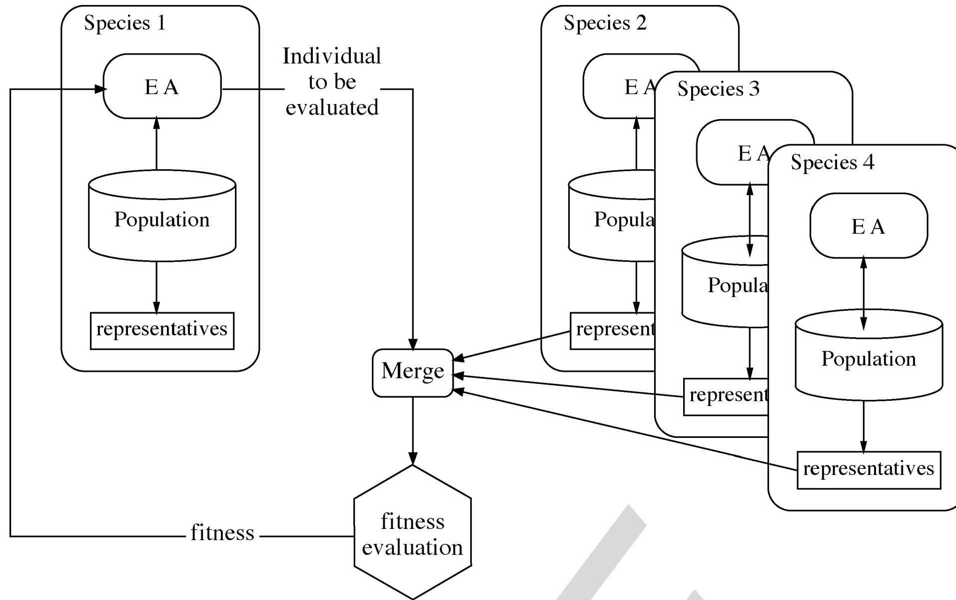

Fig. 2. Potter's and De Jong's cooperative coevolutionary system. The figure shows the evolutionary process from the perspective of Species 1. The individual being evaluated is combined with one or more *representatives* of the other species so as to construct several solutions which are tested on the problem. The individual's fitness depends on the quality of these solutions.

coevolutionary framework, and the manner in which each evolutionary algorithm computes the fitness of its individuals by combining them with selected representatives from the other species. Representatives are usually selected via a greedy strategy as the fittest individuals from the last generation.

Results presented by Potter and De Jong [24] show that their approach adequately addresses issues like problem decomposition and interdependencies between subcomponents. The cooperative coevolutionary approach performs as good as, and often better than, single-population evolutionary algorithms. Finally, cooperative coevolution usually requires less computation than single-population evolution because the populations involved are smaller, and convergence—in terms of number of generations—is faster.

C. Input Domain

The input domain of interest to us herein is inspired by the field of DNA sequencing. All experiments were performed by comparing the performance of different algorithms on this input domain.

The input strings used in the experiments were generated in a manner similar to the one used in DNA sequencing (Section I): A random **binary** string of fixed length is generated, duplicated a predetermined number of times, whereupon the copies are randomly divided into blocks of size within a given range. The set of all these blocks is the input to the SCS problem. The process is shown in Fig. 3. Note that the SCS of such a set is not necessarily the original string (it may be shorter), though it is likely to be very close to it due to the original string's randomness. (On large problem instances, with very high probability, the SCS is **precisely** the original string). The hardness of the SCS problem is retained even under this input restriction.

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. Finally, 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.

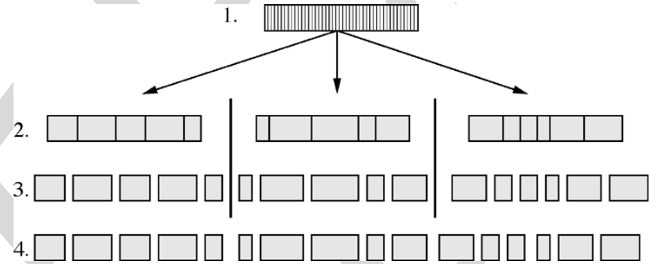

Fig. 3. Input-generation procedure. 1) Random string is generated. 2) The string is duplicated a predetermined number of times (three, in the above example). 3) Each copy is partitioned into nonoverlapping blocks of random size between *minimal block size* (20, in our case) and *maximal block size* (30, in our case). 4) The resulting set of blocks is the input set. (Note that while blocks produced from the same copy are nonoverlapping, large overlaps can—and do—exist between blocks derived from different copies.).

D. Previous Evolutionary Approaches to the SCS Problem

To the best of our knowledge, we were the first to apply an evolutionary approach to solve the SCS problem. In [1], we presented two GAs for the SCS problem: a standard GA and a cooperative coevolutionary algorithm. We briefly describe them as follows.

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).

```
GA(S)

parameter(s): S – set of blocks
output: superstring of set S

Initialization:
t ← 0
Initialize P_t to random individuals from S*
EVALUATE-FITNESS-GA(S, P_t)

while termination condition not met
   do { Select individuals from P_t (fitness proportionate)
        Recombine individuals
        Mutate individuals
        EVALUATE-FITNESS-GA(S, modified individuals)
        P_{t+1} ← newly created individuals
        t ← t + 1
return (superstring derived from best individual in P_t)

procedure EVALUATE-FITNESS-GA(S, P)
   S – set of blocks
   P – population of individuals
 for each individual i ∈ P
   do { generate derived string s(i)
        m ← all blocks from S that are not covered by s(i)
        s'(i) ← concatenation of s(i) and m
        fitness(i) ← 1 / ||s'(i)||^2
```

Fig. 4. Pseudocode of the standard genetic algorithm (GA).

In the standard GA, 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; thus, this is not a permutation-based representation.

At first glance, a permutation-based representation would seem more natural since the problem is in actuality a permutation problem. The reason for eschewing such a representation stems from the fact that where our chosen input domain is concerned, not all blocks are necessarily required to construct a short superstring. Indeed, only a small portion of blocks is usually needed to construct the SCS. Moreover, we performed some preliminary experiments that showed that permutation-based GAs perform very badly.

The chosen representation has a few additional advantages.

1) This representation enables the use of simple genetic operators similar to the ones used in binary based-representation GAs, e.g., two-point crossover (which allows both growth and reduction in individual genome lengths), and flipping-block mutations, without the need to preserve the permutation property.
2) The flexibility of the genome length allows the progressive construction of better global solutions.

Each individual derives a corresponding superstring by applying the *superstring* relation (Section II-A) 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. When the derived string does not cover all blocks, the remaining blocks are concatenated (without overlaps) to the back end of the derived string (hence, increasing its size). See Fig. 4 for the formal pseudocode.

```
Cooperative coevolutionary GA(S)

parameter(s): S – set of blocks
output: superstring of set S

Initialization:
t ← 0
for each species g ∈ G
  do Initialize P_t(g) to random individuals from S*
for each species g ∈ G
  do EVALUATE-FITNESS-COCO(S, P_t(g), G)

while termination condition not met
   do { for each species g ∈ G
          do { Select individuals from P_t(g) (fitness
                   proportionate)
               Recombine individuals
               Mutate individuals
               EVALUATE-FITNESS-COCO(S, modified
                   individuals, G)
               P_{t+1}(g) ← newly created individuals
        t ← t + 1
for each species g ∈ G
  do get best individual from P_t(g)
return (superstring derived from best individuals of species)

procedure EVALUATE-FITNESS-COCO(S, P, G)
   S – set of blocks
   P – population of individuals
   G – all species
 for each individual i ∈ P
   do { c ← ∅
        for each species g ∈ G
          do if g ≠ species of individual i
          then c ← c ∪ representative(g)
          else c ← c ∪ i
        j ← ordered concatenation of c
        generate derived string s(j)
        m ← all blocks from S that are not covered by s(j)
        s'(j) ← concatenation of s(j) and m
        fitness(i) ← 1 / ||s'(j)||^2
```

Fig. 5. Pseudocode of the cooperative coevolutionary genetic algorithm with two species: $G = \{G_1, G_2\}$. G_1: population of prefixes, G_2: population of suffixes.

The cooperative coevolutionary GA usually evolves two species simultaneously [1] (although a single population may also be used [25]). The first contains prefixes of candidate solutions to the SCS problem at hand, while the second species contains candidate suffixes. The fitness of an individual in each of the species depends on how well it collaborates with representatives from the other species to construct the global solution (Section II-B).

Each species nominates its fittest individual as the *representative*. When computing the fitness of an individual in the prefix (suffix) population, its genome is concatenated with the representative of the suffix (prefix) population, to construct a full-blown candidate solution to the SCS problem at hand. This solution is then evaluated in the same manner as in the standard GA. See Fig. 5 for the formal pseudocode.

In most cooperative coevolutionary algorithms presented to date, the (usually two) species are predefined by the designer and no single species contains individuals that singlehandedly

IEEE Proof

solve the problem. It is interesting to note that in our case, both species start out with random individuals composed of blocks, these individuals comprising possible (albeit bad) problem solutions in full. The division to prefixes and suffixes is not fixed in advance but rather comes about dynamically, and may, thus, be regarded as a form of *speciation*.

Our experiments compared the performance of both evolutionary algorithms and GREEDY (Section II-A), with cooperative coevolution proving to be best, surpassing in performance both the GA and GREEDY [1].

To summarize, standard GAs experience difficulties with large problem instances, especially when there are interdependencies among the components. We believe the main reason behind the cooperative coevolutionary algorithm's success is that it **automatically** and **dynamically** decomposes a hard problem into a number of easier problems, with less interdependencies, which are then each solved efficiently using a standard GA.

E. Experimental Design

For comparative purposes, in all experiments performed, we used exactly the same problem instances as in [1], where two series of experiments were performed differing only in the initial-string length. The parameters used in the input-generation phase were:

- *size of random string*: 250 (50-block experiment), 400 (80-block experiment)[1];
- *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 whichever parameter change) results in exponential growth of the problem's complexity.

The evolutionary parameters used for all experiment were as follows:

- *population size*: 500;
- *number of generations*: 5000;
- *crossover rate*: 0.8;
- *mutation rate*: 0.03;
- *unique problem instances per experiment*: 50.

The experiments described in the next two sections each compares the performance of a number of algorithms on a set of 50 different problem instances of given problem size. On each problem instance, each type of evolutionary algorithm was executed twice and the better run of the two was used for statistical purposes. (As argued by Sipper [26], what ultimately counts when solving a truly hard problem by an evolutionary algorithm is the *best* result.)

The evolutionary algorithms considered in this paper are much slower than nonevolutionary ones (including GREEDY, Blum *et al.* [6], and Sweedyk [13]). An evolutionary run takes 1–3 h on a run-of-the-mill PC, as opposed to a few seconds for nonevolutionary runs. Hence, if you are looking for a fast algorithm—use GREEDY (see Section II-A). However, if you wish to *drastically* improve performance—and are willing to exercise patience—then, as we shall shortly show, our evolutionary algorithms are best. (For this reason, we provide no time comparisons with GREEDY and the like as we forfeit in this domain *a priori*.)

[1]This may be deemed rather small, but since our input is generated with much interblock overlap (see paragraph on Frieze and Szpankowski in Section II-A), larger sizes might possibly be mapped to smaller ones.

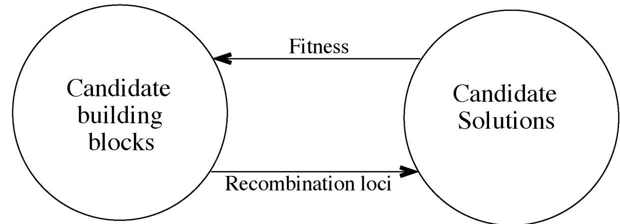

Fig. 6. The puzzle algorithm's general architecture involves two coevolving species (populations): candidate solutions and candidate building blocks. The fitness of an individual in the building-blocks population depends on individuals from the solutions population. The choice of recombination loci in the solutions species is affected by individuals from the building-blocks population.

III. Puzzle Algorithm

Theoretical evidence accumulated to date suggests that the success of GAs stems from their ability to combine quality subsolutions (building blocks) from separate individuals in order to form better global solutions. This conclusion presupposes that most problems in nature have an inherent structural design. Even when the structure is not known explicitly, GAs detect it implicitly and gradually enhance good building blocks.

Nonetheless, there are many problems that standard GAs fail to solve, even though a solution can be attained through the juxtaposition of building blocks. This phenomena is widely studied and is known as the *linkage problem*.

The Puzzle Algorithm is an extension of the standard GA motivated by the desire to address the linkage problem. It should especially improve a GA's performance on relative-ordering problems, such as the SCS problem (where the *order* between genes is crucial, and not their *global locus* in the genome).

The main idea is to preserve good building blocks found by the GA, an idea put into practice by placing constraints on the choice of recombination loci, such that good subsolutions have a higher probability of "surviving" recombination. This process should promote the assembly of increasingly larger good building blocks from different individuals. Reminiscent of assembling a puzzle by combining individual pieces into ever-larger blocks, our novel approach has been named the *Puzzle Algorithm*.

Apparently, nature, too, does not "choose" recombination loci at random. Experimental results suggest that human DNA can be partitioned into long blocks, such that recombinants within each block are rare or altogether nonexistent [27], [28].

Added to the standard GA's population of candidate solutions is a population of candidate building blocks coevolving in tandem. Interaction between **solutions** and building blocks is through the fitness function, while interaction between **building blocks** and solutions is through constraints on recombination points. Fig. 6 shows the general architecture of the Puzzle Algorithm.

IEEE Proof

The solutions population is identical to that of a standard GA (Section II-D): representation, fitness evaluation, and genetic operators. The single—but crucial!—difference is the selection of the recombination points, which is influenced by the building-blocks population.

A building-block individual is represented as a sequence of blocks. The initial population comprises random pairs of blocks, each appearing as a subsequence of at least one individual from the solutions population. Fitness of an individual depends entirely on the solutions population, and is the average fitness of the solutions that contain its genome.

Individuals from the building-blocks population are "unisex," i.e., no recombination is performed. Rather, the following two modification operators are defined.

1) *Expansion*: An addition of another block to increase the individual's genome length by one block. The added block is selected in such a way that the genome will still be a subsequence of at least one candidate solution. This operator is applied with high probability (set to 0.8 in our experiments), its purpose being to construct larger and fitter building blocks.
2) *Exploration*: With much lower probability (set to 0.1 in our experiments) an individual may "die" and be reinitialized as a new, two-block individual. This operator acts as a mutative force within the building-blocks population, injecting noise and, thus promoting exploration of the huge building-blocks search space.

Selection of recombination loci. The selection of recombination loci within an individual of the solutions population depends on individuals from the building-blocks population. Our goal is that a good section of the individual not be destroyed in the process of mating, hence, each possible recombination point is assigned with the maximal fitness of an individual from the other species that corresponds to that point. The higher the fitness, the lower the probability that the point will be selected in the recombination process.

Each individual from the solutions population maintains an additional *recombination-aid* vector (of size genome-length + 1), whose value at position i represents the likelihood of choosing locus i as a crossover point (a higher value entails a lower probability of being chosen).

During evaluation of the building-blocks population, each of its individuals is assigned a fitness value based on the average fitness of all solutions that contain its genome (as described earlier).

Each building-block individual updates all *recombination-aid* vectors of individuals within the solutions population that contain its genome as part of their genome: the individual scans each solution genome. Whenever a match is found between the building block and a solution-genome segment, each locus i corresponding to the building block (which is several blocks long and, thus, spans several loci) in the *recombination-aid* vector is updated, thus, if the fitness of the building-blocks individual is greater than locus i's current value, that value is replaced with this new fitness; otherwise, no change is effected.

After this process is repeated for all building blocks, each locus value in the *recombination-aid* vector of a solutions individual equals the fitness of the **fittest** building block that "covers" that locus. The border positions (left and right) in each vector are assigned their neighbor's value. Notice that this process does not involve any fitness evaluations. Fig. 7 shows this process.

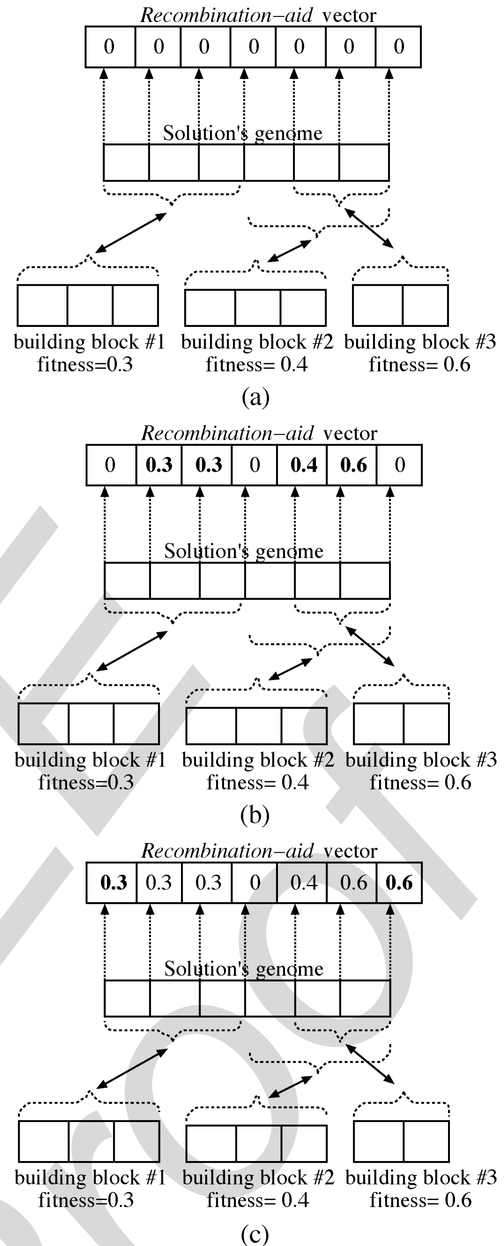

Fig. 7. *Recombination-aid* vector update procedure per a specific solutions individual, based upon fitness values (obtained before) of the building-block population. (a) Before updating. (b) After all building blocks have performed an update. (c) Border updating is done by duplicating values of neighboring cells.

When recombination occurs the selection of crossover loci is done using the *recombination-aid* vector. The probability of a specific locus within an individual solution to be chosen depends directly upon the value that is stored in the *recombination-aid* vector's corresponding position.

Two alternatives for recombination exist: with low probability (set to 0.3 in our experiments), the standard random two-point crossover operator is applied (without resorting to the *recombination-aid* vector); for the rest of the cases (i.e.,

IEEE Proof

0.7 in our experiments), the two points for crossover are the two loci with the minimal values among the *recombination-aid* vector positions (ties are broken arbitrarily).

Other possibilities for choosing recombination loci also come to mind, e.g., fitness-proportionate within the *recombination-aid* vector (herein, we did not test this). Fig. 8 shows the usage of the *recombination-aid* vector during recombination. Fig. 9 formally presents the pseudocode of the Puzzle Algorithm.

DevRev algorithm. Recently, de Jong [29] and de Jong and Oates [30] presented what they called the *DevRev* algorithm, wherein modules are developed (coevolved) alongside assemblies (solutions). The DevRev algorithm uses recursive module formation—leading to hierarchy, and uses Pareto-coevolution for module evaluation.

De Jong's and Oates's motivation stems from the desire to incrementally build hierarchical solutions from primitives. Their primitives are essentially building blocks, albeit in a more restricted sense. Under their scheme, the most frequent pair of modules (or building blocks) in the assemblies population is considered for addition to the building-blocks population (this latter can only grow, as opposed to our approach, which is more dynamic). The combined pair of modules is accepted as a new module only if it passes a stringent test (to avoid the formation of spurious building blocks), a method which has a number of drawbacks (including suitability for specific problems, the difficulty with which new building blocks are added, and the inability to remove added building blocks).

In their work, assemblies are constructed from fixed-length sequences of building blocks. Our approach seems to be more flexible in that the interpopulation interactions and the evolutionary operators allow for more complex evolutionary dynamics, resulting in better problem-solving capabilities. The choice of a sample problem also reflects the difference in motivation between our work and theirs: de Jong and Oates applied their method to pattern-recognition tasks and hierarchical test problems, stemming from their interest in hierarchical problem solving, whereas we have tackled a standard NP-complete problem, representing our interest in general problem solving via building-block manipulation.

A. Experimental Results: Standard GA Versus Puzzle Algorithm

One can understand the Puzzle Algorithm in two equivalent ways: as an addition of a building-blocks population to a standard GA, or as a novel and complex recombination operator used in a standard GA. Under both points of view, the Puzzle Algorithm is an extension of the standard GA and, thus, should be compared to it in order to test its efficiency. Such a comparison focuses on the extra power gained by selecting better recombination loci. We, thus, leave the comparison with the cooperative coevolutionary algorithm (Section II-D) for later.

The Puzzle Algorithm's performance was compared with the standard GA (Section II-D) and GREEDY (Section II-A) on sets of approximately 50 blocks, and sets of approximately 80 blocks,[2] generated as explained in Section II-C. The only difference between the two series of experiments is in the input-generation process, specifically, in the length of the initial random

[2]Since blocks are randomly created within a certain range (see Section II-E) the block count is approximate.

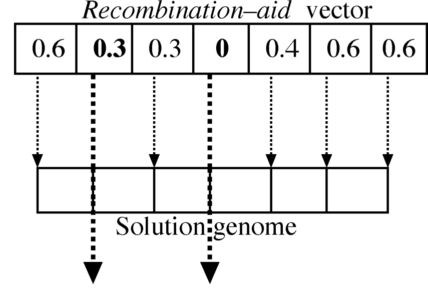

Fig. 8. Choosing recombination loci using the *recombination-aid* vector. The two loci chosen are those that are least destructive, in that good (higher fitness) building blocks are preserved.

```
Puzzle(S)

parameter(s): S – set of blocks
output: superstring of set S

Initialization:
t ← 0
Initialize SO_t to random individuals from S*
for each i ∈ SO_t
  do initialize recombination − aid vector(i) to zeros
EVALUATE-FITNESS-GA(S, SO_t)
Initialize BB_t to pairs extant in SO_t
EVALUATE-FITNES-BB(BB_t, SO_t)
for each i ∈ SO_t
  do update recombination − aid vector(i)

while termination condition not met
     { Evolve solutions population:
     {  { Select individuals from SO_t (fitness proportionate)
     {  { Recombine individuals based on
     {  {     recombination − aid vector
     {  { Mutate individuals
     {  { EVALUATE-FITNESS-GA(S, modified individuals)
     {  { SO_{t+1} ← newly created individuals
     {
     { Evolve building − blocks population:
  do {  { EVALUATE-FITNESS-BB(BB_t, SO_{t+1})
     {  { Select individuals from BB_t (fitness proportionate)
     {  { Apply Expansion operator on individuals
     {  { Apply Exploration operator on individuals
     {  { EVALUATE-FITNESS-BB(modified individuals,
     {  {     SO_{t+1})
     {  { BB_{t+1} ← newly created individuals
     { t ← t + 1
     { for each i ∈ SO_t(G)
     {   do update recombination − aid vector(i)
return (superstring derived from best individual in SO_t)

procedure EVALUATE-FITNESS-BB(B, P)
    B − population of candidate building blocks
    P − population of candidate solutions
 for each individual i ∈ B
   do fitness(i) ← average fitness of all individuals j ∈ P
        such that building block i ⊆ solution j
```

Fig. 9. Pseudocode of the puzzle algorithm, at the heart of which lie two coevolving populations: SO-candidate solutions, and BB-candidate building blocks. EVALUATE-FITNESS-GA is the same as in Fig. 4. Fig. 7 explains the particulars of the *recombination-aid* vector. Note that during the evolution of the building-blocks population EVALUATE-FITNESS-BB is applied twice: once at the beginning (since the solutions population has just evolved) and then again after expansion and exploration have been applied.

ence between the two series of experiments is in the input-generation process, specifically, in the length of the initial random

IEEE Proof

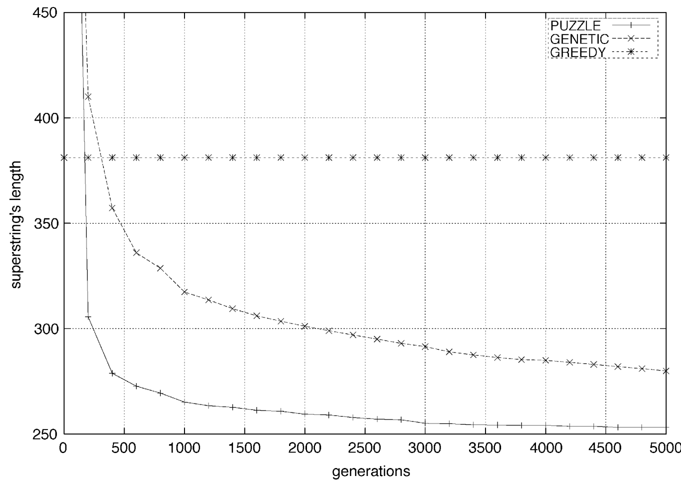

Fig. 10. Experiment I: 50 blocks. (Average of) best superstrings as a function of time (generations). Each point in the figure is the average of the best superstring lengths (at the given time); this average is computed over 50 runs on 50 different randomly generated problem instances (for each such instance, two runs were performed, i.e., a total of 100, the better of which was considered for statistical purposes). Shown are results for three algorithms: puzzle (PUZZLE), GA (GENETIC), and GREEDY (GREEDY). The straight line for GREEDY is shown for comparative purposes only (GREEDY involves no generations, and—as noted earlier—computes the answer rapidly).

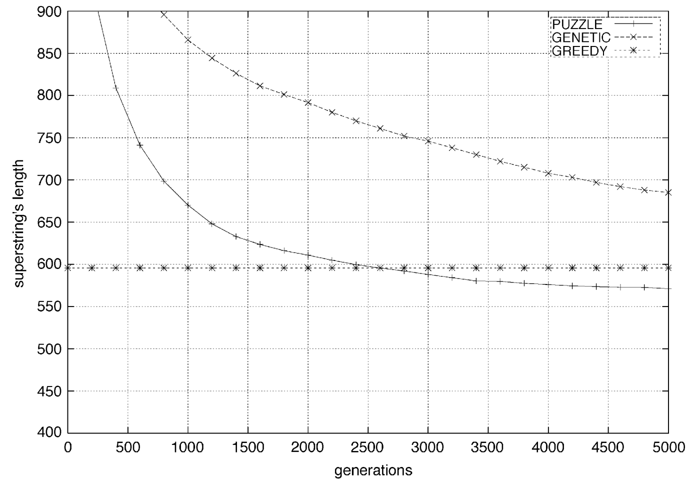

Fig. 11. Experiment II: 80 blocks. Best superstring as a function of time. Shown are results for three algorithms: puzzle algorithm (PUZZLE), genetic algorithm (GENETIC), and GREEDY (GREEDY). Interpretation of graphs is as in Fig. 10.

generated string, which entails a difference in the number of blocks given as input to the algorithm.

The standard GA's setup is detailed in Section II-E. As for the building-blocks population:

- *population size*: 1000;
- *selection*: fitness-proportionate, with elitism rate of 1;
- *expansion rate*: 0.8;
- *exploration rate*: 0.1.

The results presented in Fig. 10 show the average length of the superstrings found for the 50-block input set. The SCS of each of the problem instances is (with high probability) of length 250 bits. The Puzzle Algorithm almost attains this optimum, generating an average superstring of length 253 bits over the 50 problem instances. In 37 out of the 50 problem instances, the Puzzle Algorithm produced a superstring of length 250 bits.

The Puzzle Algorithm dramatically outperforms both GREEDY (averaging a superstring of length 381 bits), and the standard GA (averaging a superstring of length 280 bits). It even surpasses the cooperative coevolutionary algorithm (averaging a superstring of length 275 bits), although the two were not compared (as discussed above).

The results presented in Fig. 11 show the average length of the superstrings found for the 80-block input set. The SCS of each of the problem instances is (with high probability) of length 400 bits.

Again, the Puzzle Algorithm emerges as the best, finding an average superstring of length 571 bits. This is much better than the standard GA (averaging a superstring of length 685 bits), and is still better than GREEDY (averaging a superstring of length 596 bits). Comparing distance-from-optimum rather than absolute superstring length underscores the Puzzle Algorithm's victory: 171 bits from optimum versus 285 bits of the standard GA.

The Puzzle Algorithm loses in this case to the cooperative coevolutionary algorithm (which averages a superstring of length 547 bits), but, again, we must not compare the two. In Section IV, we will show how to "beat" the cooperative coevolutionary algorithm using an enhanced Puzzle Algorithm.

IV. Adding Cooperative Coevolution

When comparing results obtained by the Puzzle Algorithm with the cooperative coevolutionary algorithm presented in [1], we see that on small (50-block) problem instances the Puzzle Algorithm is best, while on larger (80-block) problem instances the cooperative coevolutionary algorithm outshines its rival.

Since the Puzzle Algorithm, which contains a single solutions population, dramatically surpasses the standard (single-population) GA, and since cooperative coevolution has proven itself highly worthwhile, the next intuitive step is to combine the two approaches. The resulting algorithm—Cooperative Coevolution + Puzzle—we denote *Co-Puzzle*.

Co-Puzzle is identical to the cooperative coevolutionary algorithm (Section II-D) with one difference: a Puzzle Algorithm evolves the prefixes and suffixes populations. Thus, there are four populations in toto: 1) prefixes; 2) suffixes (as in the cooperative algorithm); 3) prefixes building blocks; and 4) suffixes building blocks, the latter two guiding the selection of recombination loci (detailed in Section III). See Fig. 12 for the formal pseudocode.

A. Experimental Results: Cooperative Coevolution Versus Co-Puzzle

The Co-Puzzle algorithm's performance was compared with the cooperative coevolutionary algorithm (Section II-D) and GREEDY (Section II-A) on the same 50-block and 80-block sets of Section III-A. The cooperative coevolutionary algorithm's setup is detailed in Section II-E. The setup for the building-blocks population is detailed in Section III-A.

The results presented in Fig. 13 show the average length of the superstrings found for the 50-block input set. The SCS of each of the problem instances is (with high probability) of length 250 bits. The Co-Puzzle algorithm produces superstrings with

Co-Puzzle(S)

```
parameter(s): S – set of blocks
output: superstring of set S

Initialization :
t ← 0
for each species g ∈ G
   do { Initialize SO_t(g) to random individuals from S*
        for each i ∈ SO_t(g)
           do initiate recombination − aid vector(i) to zeros
for each species g ∈ G
   do { EVALUATE-FITNESS-COCO(S, SO_t(g), G)
        Initialize BB_t(g) based on SO_t(g)
        EVALUATE-FITNES-BB(BB_t(g), SO_t(g))
        for each i ∈ SO_t(g)
           do update recombination − aid vector(i)

while termination condition not met
   do { for each species g ∈ G
           do { Evolve solutions population :
                   { Select individuals from SO_t(g)
                        (fitness proportionate)
                     Recombine individuals based on
                        recombination − aid vector
                     Mutate individuals
                     EVALUATE-FITNESS-COCO(S,
                        modified individuals, G)
                     SO_{t+1}(g) ← newly created individuals

                Evolve building − blocks population :
                   { EVALUATE-FITNESS-BB(BB_t(g),
                        SO_{t+1}(g))
                     Select individuals from BB_t(g)
                        (fitness proportionate)
                     Apply Expansion operator on individuals
                     Apply Exploration operator on individuals
                     EVALUATE-FITNESS-BB(modified
                        individuals, SO_{t+1}(g))
                     BB_{t+1} ← newly created individuals
                for each i ∈ SO_t(g)
                   do update recombination − aid vector(i)
        t ← t + 1
for each species g ∈ G
   do get best individual from SO_t(g)
return (superstring derived from best solutions of species)
```

Fig. 12. Pseudocode of Co-Puzzle. G, SO, and BB are as defined in Figs. 5 and 9. EVALUATE-FITNESS-COCO is defined in Fig. 5. EVALUATE-FITNESS-BB is defined in Fig. 9.

average length of 268 bits. This is slightly better than the cooperative coevolutionary algorithm (averaging a superstring of length 275 bits), and much better than GREEDY (averaging a superstring of length 381 bits).

In this case, overlaying Puzzle with cooperative coevolution slightly improves upon cooperative coevolution alone but is degraded *vis-a-vis* Puzzle alone. This seems strange, *prima facie*, since, separately, both cooperative coevolution and Puzzle each outperforms the standard GA. We provide an explanation for this in Section V.

The results presented in Fig. 14 show the average length of the superstrings found for the 80-block input set. The SCS of each of the problem instances is (with high probability) of length 400 bits.

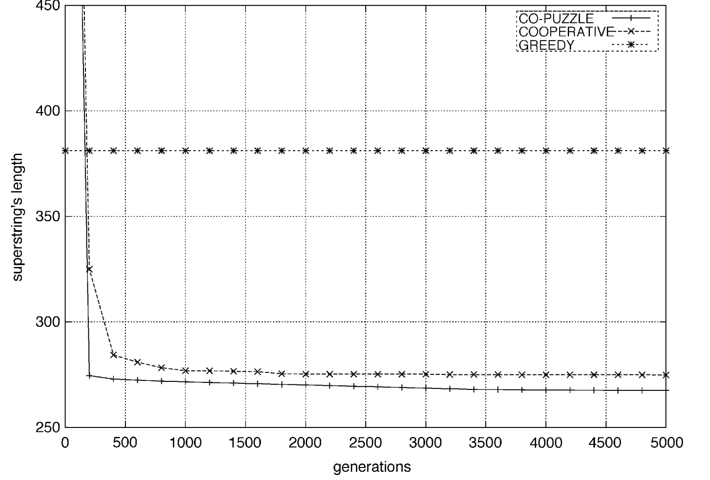

Fig. 13. Experiment III: 50 blocks. Best superstring as a function of time. Shown are results for three algorithms: combination of the puzzle algorithm and cooperative coevolution (CO-PUZZLE), cooperative coevolutionary algorithm (COOPERATIVE), and GREEDY (GREEDY). Interpretation of graphs is as in Fig. 10.

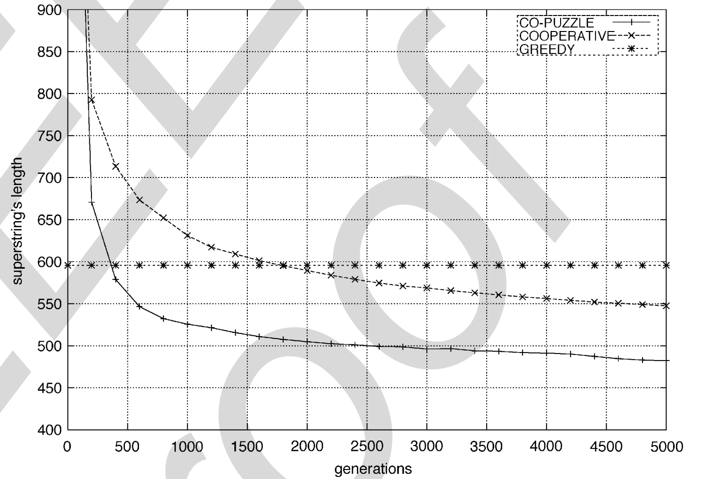

Fig. 14. Experiment VI: 80 blocks. Best superstring as a function of time. Shown are results for three algorithms: combination of the puzzle algorithm and cooperative coevolution (CO-PUZZLE), cooperative coevolutionary algorithm (COOPERATIVE), and GREEDY (GREEDY). Interpretation of graphs is as in Fig. 10.

The Co-Puzzle algorithm discovers superstrings with average length of 482 bits—a distance of 82 bits from the optimum. This is a 42% improvement over the cooperative coevolutionary algorithm, which produces superstrings of length 547 bits (i.e., 147 bits from the optimum). GREEDY comes last with 596-bit-long superstrings on average.

V. Discussion

Fig. 15 shows the four GAs presented in this paper, along with the relations between them. In [1], we presented the transformation from a simple GA to a cooperative GA (bottom-left arrow in Fig. 15). In this paper, we added the other three arrows (transformations). These relations are general, in the sense that the same transformational diagram of Fig. 15 applies not only to the SCS problem but to a variety of different problem domains.

IEEE Proof

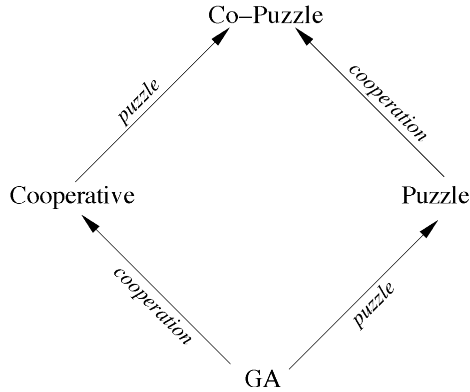

Fig. 15. "Moving" between the algorithms presented in this paper.

TABLE I
BEST AVERAGE RESULTS (ALONG WITH BITWISE DISTANCE FROM OPTIMUM IN PARENTHESIS) OBTAINED BY THE FIVE ALGORITHMS PRESENTED IN THIS PAPER: GREEDY, GA, COOPERATIVE COEVOLUTION, PUZZLE, AND CO-PUZZLE. FOR EACH OF THE 50 RANDOMLY GENERATED PROBLEM INSTANCES EACH GENETIC ALGORITHM WAS RUN TWICE, THE WORSE OF THE TWO DISCARDED (AVERAGE VALUE IS, THUS, COMPUTED OVER 50 RUNS)

Problem size	GREEDY	GA	Cooperative	Puzzle	Co-Puzzle
50	381 (131)	280 (30)	275 (25)	**253 (3)**	268 (18)
80	596 (196)	685 (285)	547 (147)	571 (171)	**482 (82)**

A summary of the results we obtained is given in Table I. We note that on large (hence, more difficult) problem instances it "pays" for cooperative coevolution to cooperate with Puzzle: Cooperative coevolution automatically decomposes the problem into a number of smaller subproblems with good interactions (i.e., the decomposition is a viable one); each subproblem is optimized using the Puzzle Algorithm. The simultaneous process of decomposition (cooperative coevolution) and optimization of the pieces (Puzzle) ultimately improves global performance.

When the problem instance is small (hence, easier), cooperative coevolution might actually prove deleterious. This is because the decomposition it performs is reasonable but **not** optimal. Hence, combining together the subproblem solutions to construct a global solution involves the use of a suboptimal decomposition. Since the problem is already easy as-is for the Puzzle Algorithm, adding coevolution only proves harmful.

To further assess our above conclusions concerning Co-Puzzle's behavior on large problem instances, we performed a series of experiments on 90- and 100-block problems, the SCS of each instance being (with very high probability) of lengths 450 and 500 bits, respectively. The input set was generated as described in Section II-C, using the setup from Section II-E. The Co-Puzzle algorithm's setup is as detailed in Section IV-A.

The results, presented in Table II, show that while GREEDY, cooperative coevolution, and Puzzle perform similarly, at the end of the day the overall winner is still Co-Puzzle: 25% better (considering distance from optimum) than its competitors on the 90-block problems, and 13% better on the 100-block problems.

Our results show that on the 50-block SCS problems two species are better than one with cooperative coevolution, but when inserting the Puzzle approach, two species prove harmful. We have not yet examined the issue of performance as a function of number of species and problem size (this is left for future work).

TABLE II
BEST AVERAGE RESULTS (ALONG WITH BITWISE DISTANCE FROM OPTIMUM IN PARENTHESIS) OBTAINED BY THE FOUR ALGORITHMS ON 90- AND 100-BLOCK PROBLEM INSTANCES: GREEDY, COOPERATIVE COEVOLUTION, PUZZLE, AND CO-PUZZLE. FOR EACH OF THE 20 RANDOMLY GENERATED PROBLEM INSTANCES EACH ALGORITHM WAS RUN TWICE, THE WORSE OF THE TWO DISCARDED (AVERAGE VALUE IS, THUS, COMPUTED OVER 20 RUNS)

Problem size	GREEDY	Cooperative	Puzzle	Co-Puzzle
90	677 (227)	673 (223)	683 (233)	**617 (167)**
100	768 (268)	768 (268)	813 (313)	**732 (232)**

The Puzzle approach undoubtedly improves a simple GA (at least for the SCS problem). There are numerous other relative-ordering problems (some even of commercial interest), including the traveling salesperson problem (TSP), and scheduling and timetabling problems. Most such problems are permutation-based, and have much in common with SCS. Therefore, the crucial idea of adding a building-blocks population to the standard GA (the Puzzle Approach) may well prove beneficial for a whole slew of problems. Moreover, from our experience, the approach is easy to implement and requires little programming.

In the Puzzle Algorithm, a candidate building block is a consecutive segment found in candidate solutions, hence, it should perform well on problems with building-block genes that are close together, i.e., *tightly linked genes*. When building-block genes are farther away, we might recourse to a "messier" mode of operation inspired by messy GAs [31], [32].

VI. PROPOSAL: THE MESSY PUZZLE ALGORITHM

The following section discusses a possible future extension of the work presented herein, based on messy GAs. We first present the latter and then present our proposed extension.

A. Brief Introduction to Messy GAs (mGAs)

In this section, we briefly review the original messy GA. Readers interested in more details should consult other sources (messy GAs [31], fast messy GAs [32]). Messy GAs (mGAs) are a class of iterative optimization algorithms that make use of a local search template, adaptive representation, and a decision theoretic sampling strategy. The work on mGAs was initiated in 1990 by Goldberg *et al.* [31] to eliminate some major problems of the standard GA. In [32], Goldberg *et al.* addressed a major deficiency of mGAs, the initialization bottleneck. During the past decade, mGAs have been applied successfully to a number of problem domains, including permutation-based problems [33].

In general, mGAs evolve a single population of building blocks (NB: no coevolution with a solutions population as in our case). During each iteration, a population of building blocks of a predefined length k is initialized, and a *thresholding selection* operator increases the number of fitter building blocks, while discarding those with poor fitness. Then, *cut*

IEEE Proof

and *slice* operators are applied to (hopefully) construct global solutions by combining good building blocks together. The best solution obtained is kept as the *competitive template* for the next iteration. In the next iteration, the length of the building blocks is increased by one (i.e., set to $k+1$). The mGA can be run level by level (i.e., k by k) until a good-enough solution is obtained or the algorithm may be terminated after a given stop criteria is met.

Messy GAs relax the fixed-locus assumption of the standard GA. Unlike a standard GA, which uses a genome of fixed length, a mGA represents the genome by variable length genes. Each gene is an ordered pair of the form $\langle allele_locus, allele_value \rangle$. Thus, a "messy" genome may be *overspecified* when more than one gene corresponds to the same allele locus, or *underspecified* when certain genes are missing. To evaluate overspecified genomes only the first appearance of a gene is considered. When evaluating underspecified genomes, the missing genes are filled with the corresponding values of a *competitive template*, which is a completely specified genome locally optimal for the previous level. This representation allows the mGA to find building blocks that include genes which are far apart in the genome by rearranging the genes of a building block in close proximity to one another.

The messy GA is organized into two nested loops: the *outer loop* and the *inner loop*. The outer loop iterates over the length k of the processed building blocks. Each cycle of the outer loop is denoted as an *era*. When a new era starts the inner loop is invoked, which is divided into three phases: 1) initialization phase; 2) primordial phase; and 3) juxtapositional phase.

Initialization phase. Initialization in the original mGA creates a population with a single copy of all substrings of length k. This ensures that all building blocks of the desired length are represented. The downside of having all these building blocks is that each must be evaluated. Since the number of these building blocks is huge it forms a bottleneck for the mGA. In [32], Goldberg *et al.* addressed this problem, presenting the *Fast Messy GA*.

Primordial phase. In this phase, *thresholding selection* alone is run repeatedly. *Thresholding selection* tries to ensure that only building blocks belonging to a particular equivalence relation are compared with one another together with selection of fitter building blocks for the next phase. A similarity measure θ is used to denote the number of common genes among two building blocks. Two building blocks are allowed to compete with each other only if their θ is greater than some threshold value $\bar{\theta}$. This method prevents the competition of building blocks belonging to different subfunctions. No fitness evaluation is performed during the primordial phase.

Juxtapositional phase. After the population is rich in copies of the best building blocks, processing proceeds in a manner similar to a standard GA. During this phase *thresholding selection* is applied together with the *cut* and *slice* operators. Good building blocks are selected, cut, and then sliced to generate better solutions. Evaluation of the objective function values is required in every generation.

The *cut* operator breaks a messy genome into two parts with a cut probability that grows as the genome's length increases. The cut position is randomly chosen along the genome. The splice operator joins two genomes with a certain splice probability.

To conclude, the advantages of mGAs include:

- obtaining and evaluating tight building blocks;
- increasing proportions of the best building blocks;
- better exchange of building blocks.

The mGA uses a building-blocks population along with a single competitive template—the best solution obtained so far. This is quite different than our approach where two bona fide populations coevolve (thus solving, along the way, the initialization problem). Nonetheless, the messy approach might be combined with our own.

B. mGAs and the Puzzle Algorithm

Using "messy" genes may help generalize the Puzzle approach to fit problems where building-block genes are far away from each other, i.e., *loosely linked genes*. This can be done by defining a candidate building-block genome as a sequence of "messy" genes that are all contained in at least one candidate solution (instead of constraining the sequence to be consecutive in the candidate solutions as in the current Puzzle approach). Essentially, we are enhancing the definition of a building block to include nonconsecutive segments in the spirit of mGAs.

It should be interesting to compare the performance of this modified Puzzle approach with the mGA. Instead of trying to assemble together good building blocks explicitly (as in the mGA), it might be better to do the same thing in a different (implicit) manner using the generalized Puzzle approach.

Intuitively, the combined messy-puzzle approach may provide benefits over Puzzle or mGA alone.

- Using a single, local-search, competitive template in the evaluation process of the building blocks in an mGA is problematic. In the Puzzle approach, there is no such single template, but many candidate solutions from the other population.
- In the mGA, one needs to define a similarity scale between different candidate building blocks (in *thresholding selection* during the primordial phase). Defining such a scale is hard. In the Puzzle approach, there is no need to understand the relation between candidate building blocks since the mixing is done implicitly in the solutions population.

Also, the dynamic nature of the building-blocks population in the Puzzle approach might deal better with the exploration of the huge building-blocks search space.

VII. Concluding Remarks and Future Work

In this paper, we presented two novel coevolutionary algorithms—Puzzle and Co-Puzzle—and tested them on the SCS problem. Both proved successful when compared with previous approaches. By bringing to the fore the building blocks—which are never dealt with directly in most GAs—the Puzzle approach is a step forward in addressing the linkage problem.

Our work opens up several avenues for future research.

- The Messy Puzzle Algorithm (Section VI-B).
- Scaling analysis of cooperative coevolution. How exactly a cooperative coevolutionary algorithm performs as a

IEEE Proof

function of the problem's size and the number of species coevolving. A good start can be with experiments on the SCS problem. Also, the effects of the Puzzle approach on the scaling behavior is quite interesting.
- Tackling larger problem instances using the Co-Puzzle algorithm with a mutable number of species. Toward this end, finding the relation between problem size and the optimized number of populations in the Co-Puzzle algorithm should be automatic. This can be done by the construction of new species on the fly as convergence is encountered (as suggested by Potter and De Jong [24]).
- We designed the Puzzle approach with relative-ordering problems in mind (where the *order* between genes is crucial, and not their *absolute locus* in the genome). Testing the approach on other relative-ordering problems and on problems from different domains is an obvious path to follow.
- A hybrid GA, where Puzzle or Co-Puzzle searches for a portion of the solution and a locally oriented greedy algorithm fills in the rest.

ACKNOWLEDGMENT

The authors are grateful to M. Tomassini and the anonymous reviewers for their many helpful remarks. They thank N. Feinstein and T. Katzenelson for their help in programming some of the experiments.

REFERENCES

[1] A. Zaritsky and M. Sipper. (2004) Coevolving solutions to the shortest common superstring problem. BioSystems. [Online]. Available: http://www.cs.bgu.ac.il/~assafza

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

[3] J. Storer, *Data Compression: Methods and Theory*. Rockville, MD: Computer Science, 1988.

[4] M. Li, "Toward a DNA sequencing theory," in *Proc. 31st Annu. IEEE Symp. Foundations Computer Science*, St. Louis, MO, 1990, pp. 125–134.

[5] M. Garey and D. Johnson, *Computers and Intractability*. New York: Freeman, 1979.

[6] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, "Linear approximation of shortest superstrings," *J. ACM*, vol. 41, pp. 634–647, 1994.

[7] S. Teng and F. Yao, "Approximating shortest superstrings," in *Proc. 34th Annu. IEEE Symp. Foundations Computer Science*, Palo Alto, CA, 1993, pp. 158–165.

[8] A. Czumaj, L. Gasieniec, M. Piotrow, and W. Rytter, "Parallel and sequential approximations of shortest superstrings," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1994, vol. 824, Proc. 1st Scandinavian Workshop Algorithm Theory, pp. 95–106.

[9] R. Kosaraju, J. Park, and C. Stein, "Long tours and short superstrings," in *Proc. 35th Annu. IEEE Symp. Foundations Computer Science*, Sante Fe, NM, 1994, pp. 166–177.

[10] C. Armen and C. Stein, "Improved length bounds for the shortest superstring problem," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1995, vol. 955, Proc. 5th Int. Workshop Algorithms and Data Structures, pp. 494–505.

[11] ——, "A 2 2/3 approximation algorithm for the shortest superstring problem," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 1996, vol. 1075, Proc. Combinatorial Pattern Matching, pp. 87–101.

[12] D. Breslauer, T. Jiang, and Z. Jiang, "Rotations of periodic strings and short superstrings," *J. Algorithms*, vol. 24, pp. 340–353, 1997.

[13] Z. Sweedyk, "A 2 1/2 approximation algorithm for shortest superstring," *SIAM J. Comput.*, vol. 29, no. 3, pp. 954–986, Dec. 1999.

[14] J. Turner, "Approximation algorithms for the shortest common superstring problem," *Inform. Comput.*, vol. 83, pp. 1–20, 1989.

[15] A. Frieze and W. Szpankowski, "Greedy algorithms for the shortest common superstring that are asymptotically optimal," *Algorithmica*, vol. 21, pp. 21–26, May 1998.

[16] J. Paredis, "Coevolutionary computation," *Artif. Life*, vol. 2, no. 4, pp. 355–375, 1995.

[17] W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," *Physica D*, vol. 42, pp. 228–234, 1990.

[18] C. D. Rosin and R. K. Belew, "New methods for competitive coevolution," *Evol. Comput.*, vol. 5, no. 1, pp. 1–29, 1997.

[19] M. A. Potter, "The design and analysis of a computational model of cooperative coevolution," Ph.D. dissertation, George Mason Univ., Fairfax, VA, 1997.

[20] C.-A. Peña-Reyes and M. Sipper, "Applying fuzzy CoCo to breast cancer diagnosis," in *Proc. Congr. Evolutionary Computation*, vol. 2, La Jolla, CA, 2000, pp. 1168–1175.

[21] ——, "Fuzzy CoCo: A cooperative-coevolutionary approach to fuzzy modeling," *IEEE Trans. Fuzzy Syst.*, vol. 9, pp. 727–737, Oct. 2001.

[22] ——, "The flowering of fuzzy CoCo: Evolving fuzzy iris classifiers," in *Proc. 5th Int. Conf. Artificial Neural Networks Genetic Algorithms (ICANNCA 2001)*, V. Kurková, N. C. Steele, R. Neruda, and M. Kárný, Eds., Vienna, 2001, pp. 304–307.

[23] R. Eriksson and B. Olsson, "Cooperative coevolution in inventory control optimization," in *Proc. 3rd Int. Conf. Artificial Neural Networks and Genetic Algorithms (ICANNGA 1997)*, G. D. Smith, N. C. Steele, and R. F. Albrecht, Eds., Norwich, U.K., 1997, pp. **XXX–XXX**.

[24] M. A. Potter and K. A. De Jong, "Cooperative coevolution: An architecture for evolving coadapted subcomponents," *Evol. Comput.*, vol. 8, no. 1, pp. 1–29, Spring 2000.

[25] P. Darwen and X. Yao, "Speciation as automatic categorical modularization," *IEEE Trans. Evol. Comput.*, vol. 1, pp. 100–108, July 1997.

[26] M. Sipper, "A success story or an old wives' tale? On judging experiments in evolutionary computation," *Complexity*, vol. 5, no. 4, pp. 31–33, Mar./Apr. 2000.

[27] S. B. Gabriel *et al.*, "The structure of haplotype blocks in the human genome," *Science*, vol. 296, no. 5576, pp. 2225–2229, 2002.

[28] L. Helmuth, "Genome research: Map of the human genome 3.0," *Science*, vol. 293, no. 5530, pp. 583–585, 2001.

[29] E. D. de Jong, "Representation development from Pareto-coevolution," in *Proc. Genetic Evolutionary Computation Conf.*, 2003, pp. 265–276.

[30] E. D. de Jong and T. Oates, "A coevolutionary approach to representation development," in *Proc. Workshop Development Representations*, E. D. de Jong and T. Oates, Eds., Sydney, Australia, 2002, [Online]. Available: http://www.demo.cs.brandeis.edu/icml02ws.

[31] D. E. Goldberg, B. Korb, and K. Deb, "Messy genetic algorithms: Motivation, analysis, and first results," *Complex Syst.*, vol. 3, no. 5, pp. 493–530, 1990.

[32] D. E. Goldberg, K. Deb, H. Kargupta, and G. Hank, "Rapid accurate optimization of difficult problems using fast messy genetic algorithms," in *Proc. 5th Int. Conf. Genetic Algorithms*, S. Forrest, Ed., San Mateo, CA, 1993, pp. 56–64.

[33] D. Kujazew and D. E. Golberg, "OMEGA—ordering messy GA: Solving permutation problems with the fast messy genetic algorithm and random keys," in *Proc. Genetic Evolutionary Computation Conf.*, D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, Eds., Las Vegas, NV, July 10–12, 2000, pp. 181–188.

Assaf Zaritsky received the B.Sc. degree in computer science from Ben-Gurion University, Beer-Sheva, Israel, in **YEAR**. He is currently working towards the M.Sc. degree in computer science from Ben-Gurion University.

His research interests include evolutionary computation and bio-inspired computing.

IEEE Proof

Moshe Sipper received the B.A. degree in computer science from the Technion—Israel Institute of Technology, Haifa, Israel, in **YEAR**, and the M.Sc. and Ph.D. degrees in computer science from Tel-Aviv University, Tel-Aviv, Israel, in **YEARS**.

He is currently an Associate Professor in the Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel. From 1995 to 2001, he was a Senior Researcher in the Swiss Federal Institute of Technology, Lausanne. He has published over 100 scientific papers, and is the author of *Machine Nature: The Coming Age of Bio-Inspired Computing* (New York: McGraw-Hill, 2002) and *Evolution of Parallel Cellular Machines: The Cellular Programming Approach* (Heidelberg, Germany: Springer-Verlag, 1997). His major research interests include evolutionary computation, bio-inspired computing, and artificial life; minor interests include cellular computing, cellular automata, and artificial self-replication, along with a smidgen of evolutionary robotics, artificial neural networks, and fuzzy logic.

Dr. Sipper is the recipient of the 1999 EPFL Latsis Prize. He is an Associate Editor of the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION and an Editorial Board Member of *Genetic Programming and Evolvable Machines*.

IEEE Proof