Ben-Gurion University of the Negev
Department of Computer Science

Coevolving Solutions to the Shortest Common Superstring Problem

Thesis submitted as part of the requirements for the
M.Sc. degree of Ben-Gurion University of the Negev

by

Assaf Zaritsky

The research work for this thesis has been carried out at
Ben-Gurion University of the Negev
under the supervision of Prof. Moshe Sipper

December 2003

Subject: **Coevolving Solutions to the Shortest Common Superstring Problem**

This thesis is submitted as part of the requirements for the M.Sc. degree

Written by: **Assaf Zaritsky**

Advisor: **Prof. Moshe Sipper**

Department: **Computer Science**

Faculty: **Natural Sciences**

Ben-Gurion University of the Negev

Author signature: _____ Date: _____

Advisor signature: _____ Date: _____

Dept. Committee Chairman signature: _____ Date: _____

Abstract

Genetic algorithms (GAs) are a class of iterative search algorithms inspired by the biological process of evolution by natural selection. Generally speaking, GAs evolve a population of candidate solutions to a given problem by iteratively applying stochastic search operators. Coevolutionary GAs are a subclass of genetic algorithms that evolve simultaneously two or more populations with coupled fitness.

The Shortest Common Superstring (SCS) problem, known to be NP-Complete, seeks the shortest string that contains all strings from a given set. Finding the shortest common superstring has important applications in computational biology where the DNA-sequencing problem is to map a string of DNA.

In this dissertation we design and apply a number of advanced coevolutionary techniques on the SCS problem, empirically compare these algorithms' performance on instances of the problem inspired by DNA sequencing, and discuss the benefits of using our methods in the general field of evolutionary algorithms.

Our research advances along two main fronts: 1) better understanding of cooperative coevolutionary algorithms, and 2) a novel coevolutionary approach for "smart" selection of recombination loci in genetic algorithms.

We start by designing a novel cooperative coevolutionary algorithm and show that using a number of coevolving populations can result in markedly improved performance when considering large (SCS) problem instances [41].

We next present a novel coevolutionary algorithm—the *Puzzle Algorithm*—where a population of building blocks coevolves alongside a population of so-

lutions to aid with the selection of recombination loci. We show that Puzzle drastically improves the performance of a standard GA [40].

We subsequently combine the cooperative coevolutionary algorithm with the Puzzle approach to form the *Co-Puzzle Algorithm* and show that the latter proves to be top gun on large problem instances. Moreover, the results obtained by the Co-Puzzle algorithm help us obtain better insight into the behavior of cooperative coevolution.

We conclude by discussing the benefits of using the Puzzle approach in the general field of evolutionary algorithms, and propose several avenues for future research.

Acknowledgment

First I would like to thank my supervisor, Prof. Moshe Sipper, who had guided me throughout my studies and led me to my achievements. I wish to thank Neta Feinstain and Tzvika Katzenelson for the great project they have conducted and for the contributing talks and companionship, as well as Rotem Shacham and Erez Karpas for reading my papers and sharing their thoughts with me. I am grateful to Marco Tomassini for his many helpful remarks. I also wish to thank the BGU Computer Science department's members, with whom I have spent these lovely last two years.

I wish to thank my father, Arieh, and my sister, Noa, for helping me start writing in English and helping me prepare for my talk at IPCAT03. Also, I wish to thank the rest of my family, my mother Hilla, my sister Yael (Hipush) and my dog Zelcer for their support.

Lastly, I wish to thank my dearest wife, Shunit, who stood there for me with infinite love and (almost) unlimited patience and to my not-yet-born son "Jango".

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction to Dissertation

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 [36]. One such popular 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 [37], because data may be stored efficiently as a superstring. SCS also has important applications in computational biology [18], 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 input domain used throughout this dissertation was inspired by this process.

The SCS problem, which is NP-Complete [9], is also MAX-SNP hard [1]. The latter means that if $P \neq NP$ no polynomial-time algorithm exists, which can *approximate* the optimum to within a given (constant) factor.

Genetic Algorithms (GAs) are a computational model of Darwinian evolution, which evolve a population of competing individuals using gene-level representation, fitness-based selection, and genetic operators. In this dissertation, GAs are used to find solutions to the SCS problem.

Cooperative coevolutionary algorithms are genetic algorithms that evolve multiple populations of competing individuals, where the populations have to cooperate in order to achieve a common goal. Cooperative coevolutionary algorithms have improved problem-solving capabilities and lowered computational costs when compared with standard genetic algorithms on many problem domains [7, 28–31, 33].

In the first part of this dissertation (Chapter 4) three approaches for finding solutions to the SCS problem are empirically compared: a standard genetic algorithm (GA), a novel cooperative coevolutionary algorithm, and a benchmark greedy algorithm (GREEDY) on instances of the SCS problem inspired by DNA sequencing. The experimental results show, among other things, that evolutionary algorithms can be applied to the SCS problem and that a cooperative coevolutionary algorithm outperform a standard genetic algorithm. Some intuition behind the cooperative coevolutionary algorithm's success is explained in this part, too.

In the second part (Chapter 5) another novel coevolutionary algorithm is presented—the *Puzzle Algorithm*—where a population of genomic building blocks coevolves alongside a population of solutions. The experimental results show that our novel algorithm outperforms the GA and GREEDY on the same instances of the SCS problem. The next comparison was be-

tween the previously presented cooperative coevolutionary algorithm and the *Co-Puzzle Algorithm*—the Puzzle Algorithm coupled with cooperative coevolution—showing that the latter proves to be top gun. Finally, the benefits of using the puzzle approach in the general field of evolutionary algorithms are discussed.

1.2 Objectives

The primary goal of this dissertation is to develop evolutionary and coevolutionary algorithms for the SCS problem. The SCS problem is a hard, relative-ordering 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 the SCS problem. Therefore, development of superior evolutionary methods for the SCS problem can help in designing a generalized evolutionary algorithm (with better performance than previous approaches) for other similar problems.

The following objectives are our milestones in achieving this goal:

- To design and implement a standard genetic algorithm for the SCS problem.
- To apply Potter's & De Jong's [33] cooperative coevolutionary model to the SCS problem, and empirically check this architecture. Namely, study the effects of this model on problem decomposition of problems that can only be decomposed into sub-problems with complex interdependencies.
- To design and implement the *Puzzle Algorithm*, an extension of the

standard GA motivated by the desire to preserve good building blocks found by the GA, and to compare this approach with the standard GA on the SCS problem.

- To combine the two approaches presented (cooperative coevolution and puzzle), to construct the *Co-Puzzle* algorithm, and compare it with the previous approaches on the SCS problem.

1.3 Contributions

The main contributions of this dissertation are as follows:

- We have designed and implemented a novel cooperative coevolutionary GA for the SCS problem, and empirically shown that cooperative coevolution is particularly useful in this domain.

- We have shown some preliminary observations about the scaling analysis of cooperative coevolution: how a cooperative coevolutionary algorithm performs as a function of the problem's size and the number of species coevolving. We have concluded that more species might help solve difficult problems, but too many species might actually prove deleterious.

- We have designed and implemented the *Puzzle Algorithm*, and shown that coevolving a population of building blocks alongside a population of candidate solutions, in order to find better loci for recombination, results in dramatic increase in problem-solving capabilities.

- We have implemented the *Co-Puzzle* algorithm, and showed that the Puzzle approach coupled with cooperative coevolution proves to be top gun, when considering large SCS instances.

- We have discussed the benefits of using the Puzzle approach in the general field of evolutionary algorithms and proposed a further generalization of the Puzzle approach based on messy genetic algorithms [11].

1.4 Dissertation Outline

In Chapter 2 we introduce the SCS problem. This introduction includes definitions, motivation, some theoretical results, and presentation of several algorithms for the problem. Also, the input domain which is inspired by DNA sequencing is presented. This input domain is used throughout the dissertation.

In Chapter 3 we provide some background and previous work on standard genetic algorithms and on coevolutionary algorithms. This chapter also includes the experimental design of the experiments performed throughout this dissertation.

In Chapter 4 we present a standard genetic algorithm and a cooperative coevolutionary algorithm for the SCS problem. Then, we empirically compare their performance on SCS instances from the chosen input domain. We conclude this chapter with an intuitive statement we believe is the main reason behind the cooperative coevolutionary algorithm's success.

In Chapter 5 we present the Puzzle Algorithm, and compare its performance with the standard GA from the previous chapter. Next, the combination of the puzzle approach and cooperative coevolution, the Co-Puzzle algorithm, is introduced, and experimentally compared with the cooperative coevolutionary algorithm. A short discussion on the benefits of using the puzzle approach in the general field of evolutionary algorithms concludes this chapter.

In Chapter 6 we begin with an introduction to messy genetic algorithms, another class of genetic algorithms. This is followed by a possible future extension of the Puzzle Algorithm based on messy genetic algorithms.

Finally, Chapter 7 summarizes the results obtained in the dissertation and suggests some directions for future research.

Chapter 2

The Shortest Common Superstring (SCS) Problem

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 [37], because data may be stored efficiently as a superstring. SCS also has important applications in computational biology [18], 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 input domain used throughout this dissertation was inspired by this process.

The SCS problem, which is NP-Complete [9], is also MAX-SNP hard [1]. The latter means that if $P \neq NP$ no polynomial-time algorithm exists, which can *approximate* the optimum to within a given (constant) factor.

In this chapter we describe previous work on the SCS problem and introduce the input domain used throughout this dissertation.

2.1 Preliminaries

Let $S = \{s_1, ..., s_n\}$ 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:
  - $overlap(cbcaca, cacac) = caca$.
  - $overlap(cacac, cbcaca) = c$.
  - $prefix(cbcaca, cacac) = cb$.
  - $merge(cbcaca, cacac) = cbcacac$.

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,...,s_n, we define the *superstring* $s = \langle s_1, s_2, ..., s_n \rangle$ to be the string $prefix(s_1, s_2) \cdot prefix(s_2, s_3) .... prefix(s_n, s_1) \cdot overlap(s_n, s_1)$. 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).

2.2 Approximation Algorithms for the SCS Problem

A number of linear approximations for the SCS problem have been described in the literature. Blum *et al.* [1] were the first to introduce an approximation algorithm, denoted TGREEDY, that produces a solution within a factor of 3 of the optimum (i.e., the superstring found is at most 3 times the length of the shortest common superstring). The best factor currently known is 2.5, and was achieved by Sweedyk [38].

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 Figure 2.1 for an example run, and Figure 2.2 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 algo-

Given the set $S_0 = \{s_1, s_2, s_3, s_4\}$; $s_1 = abba, s_2 = banabana, s_3 = our, s4 = urban$. GREEDY runs as follows:

1. $s_5 = merge(s_4, s_2) = urbanabana \implies S_1 = \{s_1, s_3, s_5\}$
2. $s_6 = merge(s_3, s_5) = ourbanabana \implies S_2 = \{s_1, s_6\}$
3. $s_7 = merge(s_6, s_1) = ourbanabanabba \implies S_3 = \{s_7\}$
4. *return* s_7

Figure 2.1: Example run of GREEDY. In this case GREEDY produces the SCS.

GREEDY(S)

parameter(s): S – set of blocks

output: superstring of set S

while $\|S\| > 1$

do $\begin{cases} choose\ s_1 \neq s_2 \in S\ such\ that\ overlap(s_1, s_2)\ is\ maximal \\ S \leftarrow (S \setminus \{s_1, s_2\}) \cup \{merge(s_1, s_2)\} \end{cases}$

return (*remaining string in* S)

Figure 2.2: Pseudocode of GREEDY algorithm.

rithm produces a solution within factor 2 of the optimum [1] [1, 38, 39] (Blum *et al.* [1] proved the factor-4-ness of GREEDY).

In their work, Blum *et al.* [1], first present a simple modified greedy procedure MGREEDY that also achieves a bound of 4 times the length of the SCS, and then they present another algorithm TGREEDY, based on

[1] An example that proves the lower bound: for the set $\{c(ab)^k, (ba)^k, (ab)^k c\}$ GREEDY produces the superstring $(ba)^k c(ab)^k c$ of length $4k+2$, while the SCS is $ca(ba)^k bc$ of length $2k + 4$.

MGREEDY, that achieves a factor-3 of the optimum.

The main difference between GREEDY and TGREEDY is that the latter allows checking the *overlap* between a string and itself. When the maximal overlap is between a string and itself, TGREEDY removes the string from the collection S (as denoted in the pseudocode of algorithm GREEDY in Figure 2.2), and inserts it into another collection, denoted T (initialized to the empty collection). When the maximal overlap is between two distinct strings, TGREEDY performs the same steps as GREEDY. When all strings are located in T, GREEDY is activated on the collection T and the resulting string is TGREEDY's output. (see Figure 2.3 for an example run, and Figure 2.4 for the formal pseudocode).

Given the same set from Figure 2.2: $S_0 = \{s_1, s_2, s_3, s_4\}$; $s_1 = abba, s_2 = banabana, s_3 = our, s4 = urban$. $T_0 \leftarrow \emptyset$. TGREEDY runs as follows:

1. $s_5 = merge(s_2, s_2) = banabana \implies S_1 = \{s_1, s_3, s_4\},\ T_1 = \{s_5\}$
2. $s_6 = merge(s_3, s_4) = ourban \implies S_2 = \{s_1, s_6\},\ T_2 = \{s_5\}$
3. $s_7 = merge(s_1, s_1) = abba \implies S_3 = \{s_6\},\ T_3 = \{s_5, s_7\}$
4. $s_8 = merge(s_6, s_6) = ourban \implies S_4 = \emptyset,\ T_4 = \{s_5, s_7, s_8\}$
5. applying GREEDY on the set T produces the superstring $ourbanabanabba$.

Figure 2.3: Example run of TGREEDY. Note that on this input set both GREEDY and TGREEDY obtained the same result. However, this is not the case in general.

The proof that the modified greedy algorithm, TGREEDY, finds a superstring of length at most 3 times the optimal involves a transformation of a SCS instance to a graph, and showing that TGREEDY "imitates" an

TGREEDY(S)

parameter(s): S – set of blocks

output: superstring of set S

Initiate $T \leftarrow \emptyset$

while $\|S\| > 0$

do
- *choose* $s_1, s_2 \in S$ *such that* $overlap(s_1, s_2)$ *is maximal*
- **if** $s_1 \neq s_2$
  - **then** $S \leftarrow (S \setminus \{s_1, s_2\}) \cup \{merge(s_1, s_2)\}$
  - **else**
    - $S \leftarrow S \setminus \{s_1\}$
    - $T \leftarrow T \cup \{merge(s_1, s_1)\}$

return $(GREEDY(T))$

Figure 2.4: Pseudocode of GREEDY algorithm.

algorithm based on graph theory (Concat-Cycles) that achieve the factor-4 approximation. Then, with another observation the factor-3 approximation is proven. More details can be found in [1].

Sweedyk's algorithm [38] is based on transforming SCS instances to graphs, and proving the upper bound of 2.5 directly by using graph-theory tools (namely, minimum-length cycle cover) on this graph.

So, which of the above algorithms is best: GREEDY, TGREEDY, or Sweedyk [38]? 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 first paper [41] were somewhat critical of our finally choosing GREEDY as the sole benchmark (a choice we repeat throughout this dissertation). Why not use the factor-3 or factor-2.5 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.* [1] show that neither TGREEDY or GREEDY is ultimately **The Best**[2].

2. We compared the performance of GREEDY and TGREEDY 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.

Due to the above reasons, and since Sweedyk's factor-2.5 algorithm is *much* more complicated to implement with little to no practical benefit, we chose to use GREEDY as our algorithm for benchmark comparisons with the evolutionary approaches.

2.3 DNA Sequencing and the Input Domain

The input domain of interest to us herein is inspired by the domain of DNA sequencing. All experiments were performed by comparing the performances

[2]On the input set $\{cab^k, ab^kab^ka, b^kdab^{k-1}\}$ GREEDY produces a superstring of length $4k+5$, while TGREEDY returns a superstring of length $3k+6$. While on the input set $\{c(ab)^k, (ab)^{k+1}a, (ba)^kc\}$ GREEDY (superstring of length $2k+5$) surpasses TGREEDY (superstring of length $4k+6$).

of different algorithms on this input domain.

DNA is a (double-stranded) sequence of nucleotides[3]. The sequence has an orientation and can be viewed as a string over the alphabet $\{A,C,G,T\}$. The *sequencing* problem in molecular biology is to "read" a string of DNA. Laboratory techniques exist for reading relatively short strands of DNA (up to 750 nucleotides); unfortunately, DNA fragments are much larger (in the human genome, the length is approximately 3×10^9 nucleotides). Thus, there is a need to cut the DNA to short fragments, which can later be sequenced. In order to perform these cuts restriction enzymes are used. Ordering the resulting fragments is not an easy task, because the loci at which the cuts were made are not traceable.

When sequencing a DNA strand in the laboratory, the DNA sequence already appears in millions of copies. In order to cut the DNA to small fragments, two different restriction enzymes[4] are used. The cutting is time-dependent, the longer the DNA is induced with the restriction enzyme, the more cuts will occur. The use of different restriction enzymes causes cuts on various locations on the DNA copies, which results in large overlaps between the pieces. If two restriction enzymes are not enough (i.e., there are still sequences longer than 750 nucleotides), an additional restriction enzyme is inserted (it is not inserted before so as not to produce fragments which are too small).

The resulting set of many large overlapping short sequences is given as input to a SCS algorithm. Intuitively, short superstrings preserve important biological structure and are good models of the original DNA strand. How-

[3]A small molecule which is one of the four building blocks of the DNA. Denoted G,C,A,T.

[4]Proteins that cut DNA at certain locations. Each cuts in a known small nucleotide sequence.

ever, when there are repeated regions in the DNA strand, the SCS algorithm will most likely "compress" those regions and thus, lose information, and return a wrong result. Luckily, the parts of DNA, which contain such repeats, are non-coding parts (do not code to genes), so the damage is less significant.

The input strings used in the experiments were generated in a manner similar to the one used in DNA sequencing: 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 Figure 2.5. 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. 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.

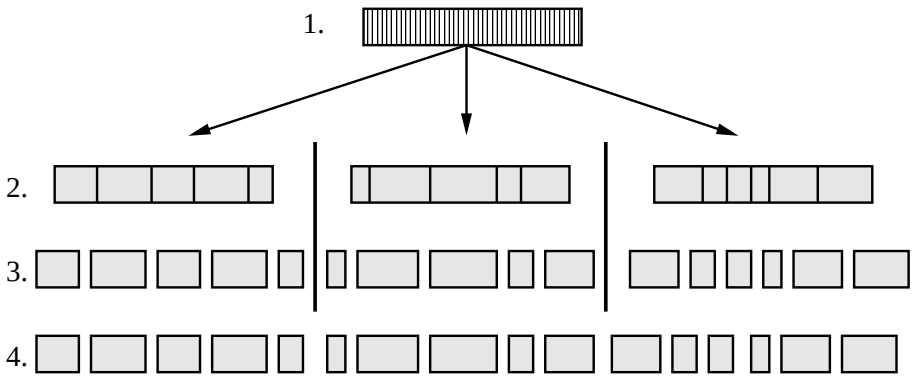


Figure 2.5: The input-generation procedure. (1) A random string is generated. (2) The string is duplicated a predetermined number of times (three, in the above example), (3) each copy is partitioned into (non-overlapping) blocks of random size between *Minimal block size* (20, in our case) and *Maximal block size* (30, in our case). (4) The resulting set of blocks is the input set.

Chapter 3

Genetic Algorithms

We start this chapter with a brief introduction to evolutionary computation. Then we present the standard genetic algorithm, the basic evolutionary approach which forms the starting point for all evolutionary algorithms presented throughout the dissertation. Next, coevolutionary genetic algorithms are described, with a focus on cooperative coevolution. We conclude this chapter with the setup of the experiments performed throughout the dissertation.

3.1 Evolutionary Computation

Evolutionary algorithms are a broad class of stochastic optimization algorithms, inspired by biology and especially by genetic inheritance and survival of the fittest (introduced in the 19th century by Charles Darwin [4]). The idea of applying the biological principle of natural evolution to artificial systems was first introduced in the mid-1960's, when Ingo Rechenberg began to work on *Evolution Strategies* [12]. About the same time, Lawrence Fogel started working independently on a technique later called *Evolutionary*

Programming, and John Holland introduced a technique we now call *Genetic Algorithms* [15]. In the mid-1980's, Lynn Cramer [3] devised another class of evolutionary algorithms, extended later by John Koza [17], and given the name *Genetic Programming*.

Evolutionary computation is ubiquitous nowadays, having been successfully applied to numerous problems from different domains, including optimization, automatic programming, machine learning, economics, operations research, ecology, population genetics, studies of evolution and learning, and social systems [36].

Although there are major differences among these classes of algorithms, they are all based on on the same fundamental principles of Darwinian evolution:

1. Organisms have a finite lifetime.

2. Offspring vary to some degree from their parents.

3. The organisms exist in an environment in which survival is a struggle and the variations among them will enable some to better adapt to the harsh environment.

4. Through natural selection the better-adapted organisms will tend to live longer and produce more offspring.

5. Offspring inherit beneficial characteristics from their parents, enabling members of the species to become increasingly well adapted to their environment over time.

In summary, evolutionary computation is the application of the above Darwinian principles to the solution of hard problems through computer simulation.

3.2 The Standard Genetic Algorithm

A genetic algorithm (GA) is an iterative search procedure that consists of a constant-size population of individuals, each one represented by a finite string of symbols, known as the *genome*, encoding a possible solution in a given problem space. This space, referred to as the *search space*, comprises all possible solutions to the problem at hand. Generally speaking, the genetic algorithm is applied to spaces which are too large to be exhaustively searched. The symbol alphabet used is often binary, though other representations have also been used, including character-based encodings, real-valued encodings, and – most notably – tree representations.

The standard genetic algorithm proceeds as follows: an initial population of individuals is generated at random or heuristically. Every evolutionary step, known as a *generation*, the individuals in the current population are *decoded* and *evaluated* according to some predefined quality criterion, referred to as the *fitness*, or *fitness function*. To form a new population (the next generation), individuals are *selected* according to their fitness. Many selection procedures are currently in use, one of the simplest being Holland's original *fitness-proportionate selection*, where individuals are selected with a probability proportional to their relative fitness [1]. This ensures that the expected number of times an individual is chosen is approximately proportional to its relative performance in the population. Thus, high-fitness ("good") individuals stand a better chance of "reproducing", while low-fitness ones are more likely to disappear.

Selection alone cannot introduce any new individuals into the population, i.e., it cannot find new points in the search space. These are generated by

[1] If the fitness of individual i is $f(i)$, then the probability of selecting individual i at each selection step is $\frac{f(i)}{\sum_j f(j)}$.

genetically-inspired operators, of which the most well known are *recombination* and *mutation*. Recombination (also known as *crossover*) is performed with probability p_{cross} (the "crossover probability" or "recombination rate") between two selected individuals, called *parents*, by exchanging parts of their genomes (i.e., encodings) to form two new individuals, called *offspring*. In its simplest form, substrings are exchanged after a randomly selected recombination locus (Figure 3.1 presents two simple recombination operators). This operator tends to enable the evolutionary process to move toward "promising" regions of the search space.

```
(a) 1-point recombination
Parent 1:             1 1 0 1 0 1 1 0 0 0 1
Parent 2:             1 0 0 0 1 1 0 1 1 0 0
recombination locus:          *
Offspring 1:          1 1 0 1 1 1 0 1 1 0 0
Offspring 2:          1 0 0 0 0 1 1 0 0 0 1

(b) 2-point recombination
Parent 1:             1 1 0 1 0 1 1 0 0 0 1
Parent 2:             1 0 0 0 1 1 0 1 1 0 0
recombination locus:        *         *
Offspring 1:          1 1 0 0 1 1 0 1 0 0 1
Offspring 2:          1 0 0 1 0 1 1 0 1 0 0
```

Figure 3.1: An example of two recombination operators: (a) 1-point recombination. (b) 2-point recombination (the genomes of the parents are cut at two randomly chosen loci).

The mutation operator is introduced to prevent premature convergence to local optima by randomly sampling new points in the search space. It is

usually carried out by flipping bits at random, with some (small) probability p_{mut} (see Figure 3.2).

```
Offspring:             1 1 0 1 0 1 1 0 0 0 1
Mutation point:                  *
Mutated offspring:     1 1 0 1 1 1 1 0 0 0 1
```

Figure 3.2: Bit-flip mutation.

Genetic algorithms are stochastic iterative processes that are not guaranteed to converge; the termination condition may be specified as some fixed, maximal number of generations or as the attainment of an acceptable fitness level. Figure 3.3 presents the standard genetic algorithm in pseudo-code format.

3.2.1 An example

Let us consider the following simple example, due to [20], demonstrating the genetic algorithm's workings. The population consists of 4 individuals, which are binary-encoded strings (genomes) of length 8. The fitness value equals the number of ones in the bit string, with $p_{cross} = 0.7$, and $p_{mut} = 0.001$. More typical values of the population size and the genome length are in the range 50-1000. Also note that fitness computation in this case is extremely simple since no complex decoding nor evaluation is necessary. The initial (randomly generated) population might look like this:

Label	Genome	Fitness
A	00000110	2
B	11101110	6

Generic GA( )

$t \leftarrow 0$ {*generation counter*}

Initialize $Population_t$

$Evaluate(Population_t)$

while *termination condition not met*

do
- *Select individuals from* $Population_t$
- *Recombine individuals*
- *Mutate individuals*
- $Population_{t+1} \leftarrow$ *newly created individuals*
- $t \leftarrow t + 1$
- $Evaluate(Population_{generation})$

return (*solution derived from best individual in* $Population_t$)

Figure 3.3: Pseudocode of the standard genetic algorithm.

```
C      00100000      1
D      00110100      3
```

Using fitness-proportionate selection we must choose 4 individuals (two sets of parents), with probabilities proportional to their relative fitness values. In our example, suppose that the two parent pairs are B,D and B,C (note that A did not get selected as our procedure is probabilistic). Once a pair of parents is selected, recombination is effected between them with probability p_{cross} , resulting in two offspring. If no recombination is effected (with probability 1-p_{cross}), then the offspring are exact copies of each parent. Suppose, in our example, that crossover takes place between parents B and D at the (randomly chosen) first bit position, forming offspring E=10110100

and F=01101110, while no recombination is effected between parents B and C, forming offspring that are exact copies of B and C. Next, each offspring is subject to mutation with probability p_{mut} per bit. For example, suppose offspring E is mutated at the sixth position to form E'=10110000, offspring B is mutated at the first bit position to form B'=01101110, and offspring F and C are not mutated at all. The next generation population, created by the above operators of selection, recombination, and mutation is therefore:

Label	Genome	Fitness
E'	10110000	3
F	01101110	5
C	00100000	1
B'	01101110	5

Note that in the new population, although the best individual with fitness 6 has been lost, the average fitness has increased. Iterating this procedure, the genetic algorithm will eventually find a perfect string, i.e., with maximal fitness value of 8.

3.3 Coevolution

Coevolution refers to the simultaneous evolution of two or more species with coupled fitness. This is in contrast with the customary evolutionary paradigm where a single population evolves under the selection pressure of a given fitness function that plays the role of the environment. However, in nature, various feedback mechanisms between the individuals undergoing selection provide a strong driving force toward complexity (consider, for example, prey-predator or host-parasite relationships). Under these conditions, evolution

can be thought as a *co-evolutionary* process where changes in a certain species influence other species, i.e, the environment is dynamically changing. Thus, a kind of "arms race" develops in which evolutionary changes in one species trigger counter-adaptive changes in other species.

These observations have been exploited in the last decade for creating a subclass of more robust artificial evolutionary algorithms, *coevolutionary algorithms*. Simplistically speaking, one can say that in coevolutionary algorithms the 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).

3.3.1 Competitive coevolution

Predator-Prey relations are the most well-known coevolution examples. There is a strong evolutionary pressure for prey to better defend themselves (e.g., by running faster, growing better shields, etc.) in response to which future generations of predators must develop better attacking capabilities (e.g, better eye-sight, running faster, etc.). In such relations, success for one side means failure to the other side. Thus, improvement of predator puts pressure on the prey to better defend itself in order to maintain its chances of survival, and when prey improves, predator must also gain better capabilities in order to maintain its chances of survival. This process of coevolution can result in significant improvement of both the prey and the predator.

Competitive coevolutionary algorithms are inspired by predator-prey relations. The fitness of an individual from one of the species 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.

Hillis [14] was the first to propose the computational use of predator-prey coevolution. The problem consisted in evolving a sorting network for 16 integers involving a minimum number of exchanges. Hillis evolved two populations, the first consisting of sorting networks, the second of sorting problems (test cases). Each population evolved separately, where interaction occurs only through the fitness function. The fitness of a sorting network is defined as how well it sorts the sorting problems; a sorting problem is scored according to how hard it is for individuals from the first population to sort it. In this way, the best networks learn to sort increasingly difficult sets of numbers, thus avoiding the need for testing all 16! possible sorting problems. This approach was able to evolve an almost optimal sorting network with 61 exchanges (the best known sorting network for this task has 60 exchanges).

Paredis [23–25] introduced a partial, continuous fitness evaluation, called *life-time fitness evaluation* (LTFE). This technique is inspired by Nature, whereas the "fitness" of an individual consists of a continuous series of tests during its life. Furthermore, these tests are not strictly predefined, but are determined based on the environment, which is influenced by the individual under evaluation as well as by other individuals. This is different from the "all at once" fitness evaluation in standard genetic algorithms.

Paredis used this technique, combined with competitive coevolution, to improve the power of artificial search on a class of problems that involve the search for a solution that meets certain a-priori given criteria (this class of problems is named by Paredis as "test-solutions problems"). Instead of checking all tests in order to evaluate a single solution, LTFE enables finding

a few well-chosen tests that provide sufficient information for evaluating the solution. This is true during different stages of the genetic search. Intuitively, the algorithm focuses on the more difficult, not-yet-solved tests.

In [24, 25] Paredis uses competitive coevolution and LTFE for a classification problem. The task was to find appropriate connection weights such that a given neural network represents the correct mapping from points in the plane to a set of predetermined classes. For that matter, two species coevolved; a candidate solutions population and a population of preclassified training examples. The evaluation of a candidate solution checked only a few tests (selected based on LTFE), this approach reveled itself as superior compared to checking all the training examples in evaluating a solution.

In [23, 25] Paredis shows that competitive coevolution and LTFE nicely complement each other in coevolving constraints and candidate solutions in a constraint satisfaction problem. In there, the constraints are used as the "tests", and the interaction between the constraints and solutions is through a competitive fitness function. In [27] Paredis used similar techniques to coevolve a neural network for a well-known bioreactor control problem.

Rosin and Belew [34] use two games (Nim and 3-D Tic-Tac-Toe) as test problems to explore three new techniques in competitive coevolution: *competitive fitness sharing* changes the way the solution fitness is measured, *shared sampling* provides a method for selecting a strong, diverse set of "tests" for solution evaluation, and *hall of fame* encourages arms races by keeping fit individuals from prior generations. A main idea behind the first two techniques mentioned is assigning better fitness to individuals (respectively tests) that solve tests (respectively fail solutions) which are hard for most of the solutions (respectively tests) even if its global fitness is lower than the average.

3.3.2 Cooperative coevolution

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 [7, 25, 26, 28–32].

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 [28–30]: (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.

Paredis [25] 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. A solution is evaluated, once and for all, immediately after its birth (as in a standard GA), while a permutation is evaluated on how well it places related genes nearby each other using LTFE (introduced in Subsection 3.3.1).

Moriarty [21, 22] used a cooperative coevolutionary approach to evolve

neural networks. Each individual in one species corresponds to a single hidden neuron of a neural network and its connections with the input and output layers. This population coevolves alongside a second one whose individuals encode sets of hidden neurons (i.e., individuals from the first population) forming a neural network.

Potter [31] and Potter & DeJong [32, 33] developed a model in which a number of populations explore different decompositions of the problem. Below we detail the framework of Potter and DeJong as we build upon it later in the dissertation.

In Potter's & 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. Figure 3.4 shows the general architecture of Potter's & De Jong's cooperative 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. Figure 3.5 shows the generic pseudocode of Potter's & De Jong's system.

Results presented by Potter and De Jong [33] show that their approach addresses adequately 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 computa-

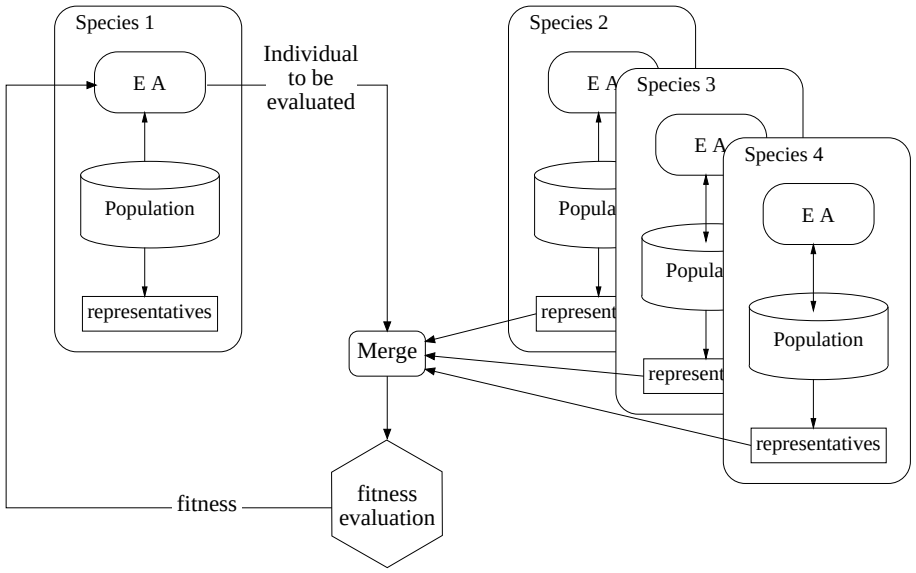


Figure 3.4: Potter's & De Jong's cooperative coevolutionary system. The figure shows the evolutionary process from the perspective of Species 1. The individual being evaluated is combined with one or more *representatives* of the other species so as to construct several solutions which are tested on the problem. The individual's fitness depends on the quality of these solutions.

tion than single-population evolution because the populations involved are smaller, and convergence—in terms of number of generations—is faster.

Eriksson and Olsson [7] applied Potter's & De Jong's model to on an inventory control optimization problem. Their main result was showing that gathering a number of strong-related species, and evolving them as a single species in the cooperative coevolutionary model might prove beneficial.

More recently, Peña-Reyes and Sipper [28–30] used cooperative coevolution to evolve fuzzy systems for breast cancer diagnosis and for Fuzzy Iris Classifiers.

Generic Cooperative coevolutionary GA( )

$t \leftarrow 0$

for each *species* $g \in G$

do *Initialize* $P_t(g)$

for each *species* $g \in G$

do EVALUATE-COCO($P_t(g)$)

while *termination condition not met*

do
- **for each** *species* $g \in G$
  - **do**
    - *Select individuals from* $P_t(g)$
    - *Recombine individuals*
    - *Mutate individuals*
    - $P_{t+1}(g) \leftarrow$ *newly created individuals*
    - EVALUATE-COCO($P_{t+1}(g)$)
- $t \leftarrow t + 1$

for each *species* $g \in G$

do *get best individual from* $P_t(g)$

return (*solution derived from collaboration of best individuals of species*)

procedure EVALUATE-COCO(P)

P – *population of individuals*

Choose representatives from other species

for each *individual* $i \in P$

do
- *Form collaboration between* i *and representatives from other species*
- *Evaluate fitness of collaboration by applying it to target problem*
- *Assign fitness of collaboration to* i

Figure 3.5: Pseudocode of the generic cooperative coevolutionary genetic algorithm.

3.4 Experimental Design

For comparative purposes, in all experiments performed we use exactly the same problem instances as in our previous articles [40, 41], where two series of experiments were performed differing only in the initial-string length. The parameters used in the input-generation phase (described in Section 2.3) are:

- *Size of random string*: 250 bits (50-block experiments), 400 bits (80-block experiments)
- *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 are 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 throughout this dissertation 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 [35] what ultimately counts when solving a truly hard problem by an evolutionary algorithm is the *best* result.)

(**"Caveat lector":** The evolutionary algorithms considered in this dissertation are much slower than non-evolutionary ones (including GREEDY, Blum *et al.* [1], and Sweedyk [38]). An evolutionary run takes 1-3 hours on a run-of-the-mill PC, as opposed to a few seconds for non-evolutionary runs. Hence, if you are looking for a fast algorithm—use GREEDY (see Section 2.2). However, if you wish to *drastically* improve performance—and are willing to exercise patience—then, as we shall 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*.))

Chapter 4

Solving SCS with Evolution and Coevolution

In the previous chapter we introduced the standard GA and the cooperative coevolutionary model. In this chapter we apply these evolutionary methods on the SCS problem. We start this chapter by briefly describing evolutionary approaches for a SCS-related problem, next, a standard GA for the SCS problem is introduced, then, a cooperative coevolutionary GA is described, and last, we experimentally compare the performance of these two algorithms on instances from the input domain.

4.1 Evolutionary Approaches to Related Problems

To the best of our knowledge, we were the first to apply an evolutionary approach to solve the SCS problem [41]. In this section we shall briefly describe a similar problem – *the Shortest Common Supersequence Problem* – and briefly present two evolutionary approaches for this problem.

Let $L = \{s_1, ..., s_n\}$ be a set of strings over some alphabet Σ. The Shortest Common Supersequence Problem is to find the shortest string S^* such that each string in L can be obtained from S^* by doing only deletions of characters.

Two kinds of "classical" heuristics have been proposed for the problem:

- At each iteration remove a pair of strings from L, compute the shortest common supersequence of this pair (e.g., by dynamic programming), and insert the result back to L. This process is repeated until L contains a single string. The heuristics in this class differ in their strategy for selecting the pairs of strings.

- Build a supersequence starting from the empty string. At each iteration choose the the next character to be appended to the end of the string from the set of first characters in the current position in each input string. This process is repeated until all strings are exhausted. The selection of the next character depends on the number of strings that have the character in their current position, and on the strings' remaining length.

Branke, Middendorf, and Schneider [2] proposed a GA for the problem that is based on the the second heuristic. The general idea is to use a heuristic for the problem that is influenced by many parameters and then optimize the parameter values by the genetic algorithm.

More recently, Michel and Middendorf [19] presented an Ant Colony optimization algorithm that transforms a Shortest Common Supersequence instance to an equivalent graph representation. A path in this graph corresponds to the outcome of a sequence of decisions that are made by an ant when constructing a solution; thus, there is a direct correspondence between a path in the graph and a solution of the problem. Ants that find a solu-

tion deposit some artificial pheromone along the edges of the solution-path in the graph (the exact amount depends on the quality of the solution). An ant's decision on where to go depends on the amount of pheromone on the graph-edges and on the second heuristic described above.

4.2 A Standard GA for the SCS Problem

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 string 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).

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 blocks 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 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. 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).

Let s denote the length of the *derived* string. Let m denote the total length of blocks that are *not* covered by the derived string. The fitness value, f, of an individual is computed as follows:

$$f = \frac{1}{(s+m)^{\alpha}}$$

α was set empirically to $\alpha = 2$ after preliminary test runs (of the standard GA and the other GAs presented throughout this dissertation). This fitness function drives evolution towards shorter superstrings covering as many blocks as possible.

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. See Figure 4.1 for the formal pseudocode.

4.3 A Cooperative Coevolutionary GA for the SCS Problem

The cooperative coevolutionary GA evolves two species simultaneously. 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 3.3).

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. 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 4.2. See Figure 4.2 for

GA(S)

parameter(s): S – set of blocks

output: superstring of set S

Initialization :

$t \leftarrow 0$

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$(S, modified\ individuals)$
- $P_{t+1} \leftarrow$ *newly created individuals*
- $t \leftarrow t + 1$

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* $i \in P$

do
- *generate derived string* $s(i)$
- $m \leftarrow$ *all blocks from* S *that are not covered by* $s(i)$
- $s'(i) \leftarrow$ *concatenation of* $s(i)$ *and* m
- $fitness(i) \leftarrow \frac{1}{\|s'(i)\|^2}$

Figure 4.1: Pseudocode of the standard genetic algorithm (GA).

the formal pseudocode.

4.4 Experimental Results: Cooperative Coevolutionary GA Versus Standard GA

Our experiments compared the performance of both evolutionary algorithms and GREEDY (Section 2.2), on sets of approximately 50 blocks, and sets of approximately 80 blocks[1], generated as explained in Section 2.3. The only difference between the two series of experiments is in the input-generation process, specifically, in the length of the initial random generated string, which entails a difference in the number of blocks given as input to the algorithm. The GA's setup is detailed in Section 3.4.

The results presented in Figure 4.3 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. 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.

The results presented in Figure 4.4 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. 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

[1]Since blocks are randomly created within a certain range (see Section 3.4) the block count is approximate.

Cooperative coevolutionary GA(S)

parameter(s): S – set of blocks

output: superstring of set S

Initialization :
$t \leftarrow 0$
for each *species* $g \in G$
  do *Initialize* $P_t(g)$ *to random individuals from* S^*
for each *species* $g \in G$
  do EVALUATE-FITNESS-COCO$(S, P_t(g), G)$

while *termination condition not met*
do
- **for each** *species* $g \in G$
  - **do**
    - *Select individuals from* $P_t(g)$ *(fitness proportionate)*
    - *Recombine individuals*
    - *Mutate individuals*
    - EVALUATE-FITNESS-COCO$(S, modified\ individuals, G)$
    - $P_{t+1}(g) \leftarrow$ *newly created individuals*
- $t \leftarrow t + 1$

for each *species* $g \in G$
  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* $i \in P$
do
- $c \leftarrow \emptyset$
- **for each** *species* $g \in G$
  - **do if** $o \neq$ *species of individual* i
  - **then** $c \leftarrow c \cup representative(g)$
  - **else** $c \leftarrow c \cup i$
- $j \leftarrow$ *ordered concatenation of* c
- *generate derived string* $s(j)$
- $m \leftarrow$ *all blocks from* S *that are not covered by* $s(j)$
- $s'(j) \leftarrow$ *concatenation of* $s(i')$ *and* m
- $fitness(i) \leftarrow \frac{1}{\|s'(j)\|^2}$

Figure 4.2: 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.

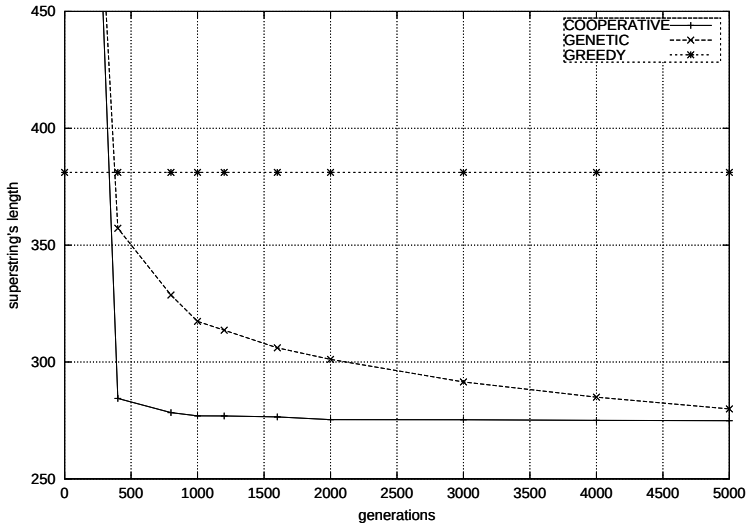


Figure 4.3: Experiment I: 50 blocks. (Average of) best superstrings as a function of time (generations). Each point in the figure is the average of the best superstring lengths (at the given time); this average is computed over 50 runs on 50 different randomly generated problem instances (for each such 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: cooperative coevolutionary algorithm (`COOPERATIVE`), 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).

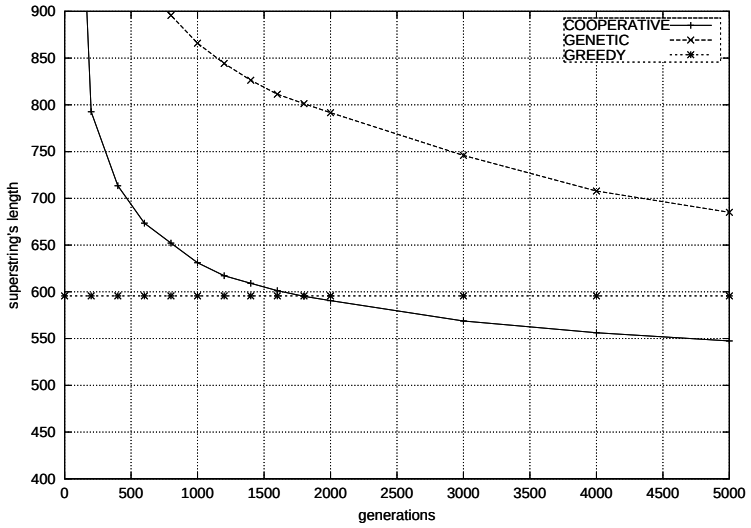


Figure 4.4: Experiment II: 80 blocks. Best superstring as a function of time. Shown are results for three algorithms: cooperative coevolutionary algorithm (`COOPERATIVE`), GA (`GENETIC`), and GREEDY (`GREEDY`). Interpretation of graphs is as in Figure 4.3.

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.

4.4.1 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* 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 4.1 shows that the results obtained by our combined approach are best.

4.5 Concluding Remarks

We applied four algorithms to the shortest common superstring problem. The results are summarized in Table 4.1. As can be seen, cooperative coevolution has revealed itself as a powerful tool, surpassing both the genetic algorithm and GREEDY. Combining the latter with coevolution yields even better results.

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.

Table 4.1: (a) Best average results obtained by the three algorithms: GREEDY, genetic, and cooperative coevolution. For each of the fifty 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. 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.

(a)

Problem size	Greedy	Genetic	Cooperative
50	381	280	275
80	596	685	547

(b)

Problem size	Greedy	Genetic	Cooperative	Parallel
80	625	683	542	510

Chapter 5

The Puzzle Algorithm

In this chapter we present the *Puzzle Algorithm*, a novel approach which coevolves a population of building blocks alongside the standard candidate-solutions population. This approach is different from the standard GA only in the selection of recombination loci, but this single difference is crucial to the GA's performance, as we empirically show.

Next, we present a combination of the puzzle approach and cooperative coevolution, the *Co-Puzzle* algorithm, and compared its performance with the cooperative coevolutionary algorithm.

Last, we discuss the advantages of cooperation and the Puzzle approach in the general field of evolutionary algorithms.

5.1 Description of Algorithm

Theoretical evidence accumulated to date suggests that the success of GAs stems from their ability to combine quality sub-solutions (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 sub-solutions 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 have suggested that human DNA can be partitioned into long blocks, such that recombinants within each block are rare or altogether non-existent [8, 13].

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. Figure 5.1 shows the general architecture of the

Puzzle Algorithm.

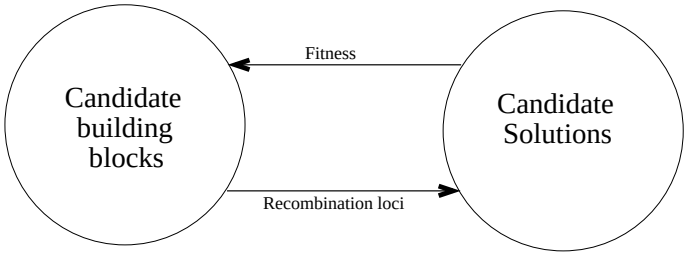


Figure 5.1: The puzzle algorithm's general architecture involves two coevolving species (populations): candidate solutions and candidate building blocks. The fitness of an individual in the building-blocks population depends on individuals from the solutions population. The choice of recombination loci in the solutions species is affected by individuals from the building-blocks population.

The solutions population is identical to that of a standard GA (Subsection 4.2): 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 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 re-initialized 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.

5.1.1 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 within the solutions population that contain its genome as part of their genome as follows: 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 fit-

ness 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. Figure 5.2 shows this process.

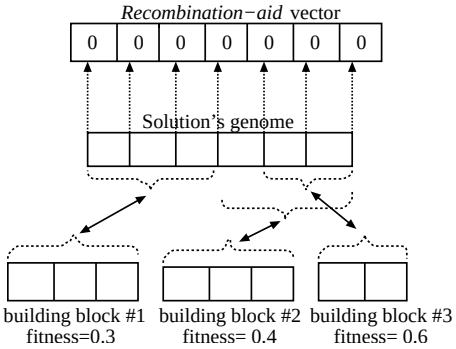
When recombination occurs the selection of crossover loci is done using the *recombination-aid* vector. The probability of a specific locus within an individual solution to be chosen depends directly upon the value that is stored in the *recombination-aid* vector's corresponding position.

Two alternatives for recombination exist: with low probability (set to 0.3 in our experiments) the standard random 2-point crossover operator is applied (without resorting to the *recombination-aid* vector); for the rest of the cases (i.e., 0.7 in our experiments) the 2 points for crossover are the two loci with the minimal values among the *recombination-aid* vector positions (ties are broken arbitrarily).

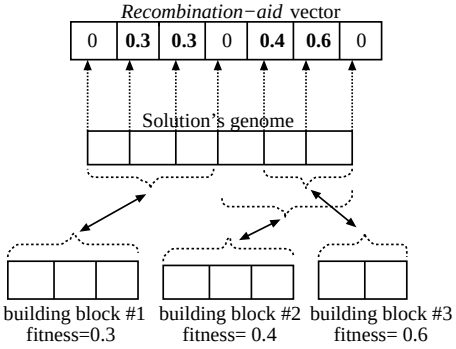
Other possibilities for choosing recombination loci also come to mind, e.g., fitness-proportionate within the *recombination-aid* vector (herein, we did not test this). Figure 5.3 shows the usage of the *recombination-aid* vector during recombination. Figure 5.4 formally presents the pseudocode of the Puzzle Algorithm.

5.1.2 The DevRev algorithm.

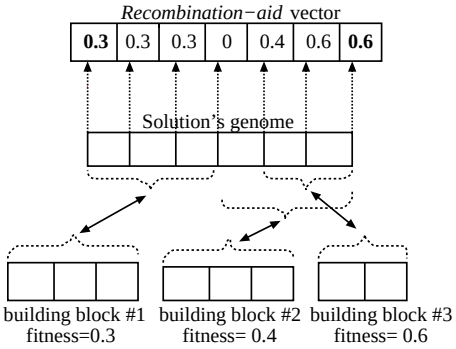
Recently, de Jong [5] and de Jong and Oates [5, 6] presented what they called the *DevRev* algorithm, wherein modules are developed (coevolved)



(a)



(b)



(c)

Figure 5.2: The *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) borders updating is done by duplicating values of neighboring cells.

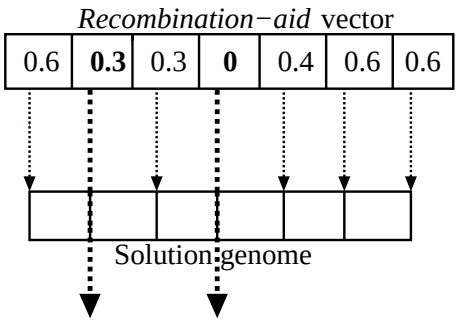


Figure 5.3: Choosing recombination loci using the *recombination-aid* vector. The two loci chosen are those that are least destructive, in that good (higher fitness) building blocks are preserved.

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 several disadvantages (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 is more flexible in that the inter-population interactions and the evolutionary operators allow for more complex evolutionary dynamics, resulting in better problem-solving capabilities. The choice of sample problem also reflects the difference in motivation between our work

```
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
  do { 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 :
       { 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
```

Figure 5.4: 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 Figure 4.1. Figure 5.2 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.

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.

5.2 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 4.3) for later.

The Puzzle Algorithm's performance was compared with the standard GA (Section 4.2) and GREEDY (Section 2.2) on the same 50-block and 80-block sets from Section 4.4. The only difference between the two series of experiments is in the input-generation process, specifically, in the length of the initial random 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 3.4. 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 Figure 5.5 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 be compared (as discussed above).

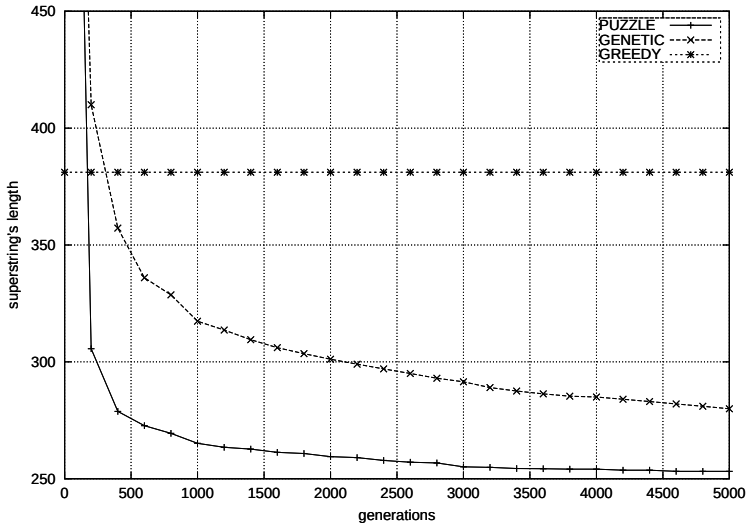


Figure 5.5: Experiment III: 50 blocks. Best superstring as a function of time. Shown are results for three algorithms: Puzzle (`PUZZLE`), GA (`GENETIC`), and GREEDY (`GREEDY`). Interpretation of graphs is as in Figure 4.3.

The results presented in Figure 5.6 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 the next section we will show how to "beat" the cooperative coevolutionary algorithm using an enhanced Puzzle Algorithm.

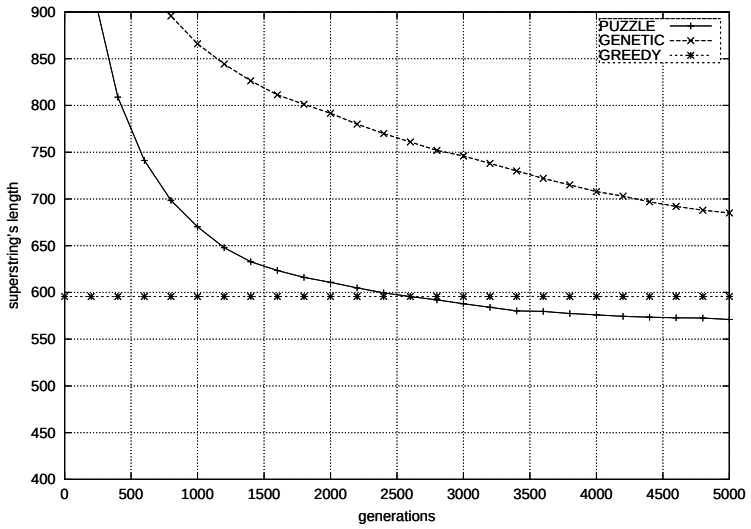


Figure 5.6: Experiment IV: 80 blocks. Best superstring as a function of time. Shown are results for three algorithms: puzzle algorithm (`PUZZLE`), genetic algorithm (`GENETIC`), and GREEDY (`GREEDY`). Interpretation of graphs is as in Figure 4.3.

5.3 Adding Cooperative Coevolution

When comparing results obtained by the Puzzle Algorithm with the cooperative coevolutionary algorithm presented in Section 4.4 we see that on small (50-block) problem instances the Puzzle Algorithm is best, while on larger (80-block) problem instances the cooperative coevolutionary 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 4.3) with one difference: a Puzzle Algorithm evolves the prefixes and suffixes populations. Thus, there are 4 populations in toto: (1) prefixes and (2) suffixes (as in the cooperative algorithm), along with (3) prefixes building blocks, and (4) suffixes building blocks, the latter two guiding the selection of recombination loci (detailed in Section 5.1). See Figure 5.7 for the formal pseudocode.

5.4 Experimental Results: Cooperative Coevolution Versus Co-Puzzle

The Co-Puzzle algorithm's performance was compared with the cooperative coevolutionary algorithm (Section 4.3) and GREEDY (Section 2.2) on the same 50-block and 80-block sets from Section 4.4. The cooperative coevolutionary algorithm's setup is detailed in Section 3.4. The setup for the

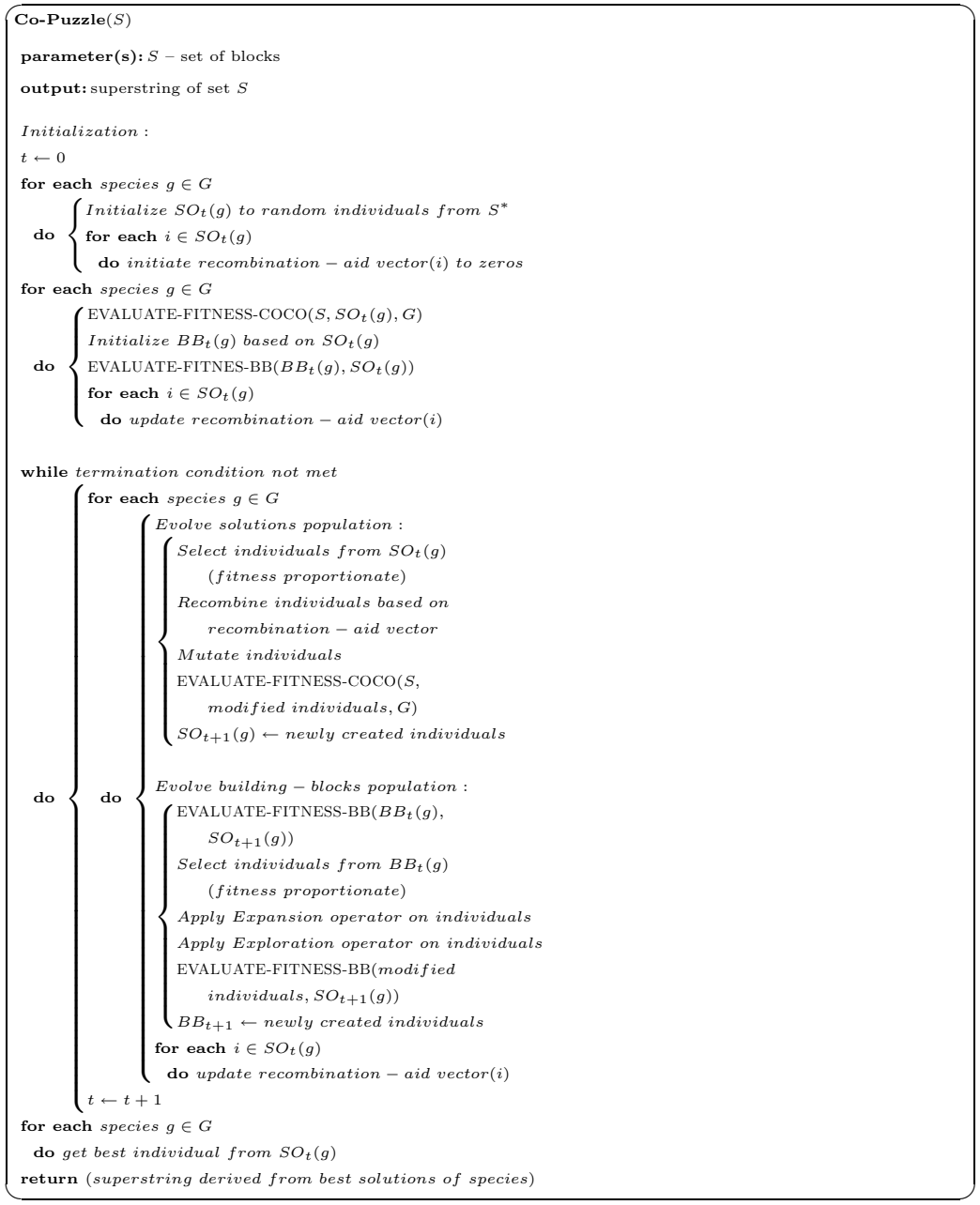


Figure 5.7: Pseudocode of Co-Puzzle. G, SO, and BB are as defined in Figures 4.2 and 5.4. EVALUATE-FITNESS-COCO is define is Figure 4.2. EVALUATE-FITNESS-BB is define in Figure 5.4

building-blocks population is detailed in Section 5.2.

The results presented in Figure 5.8 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 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 5.5.

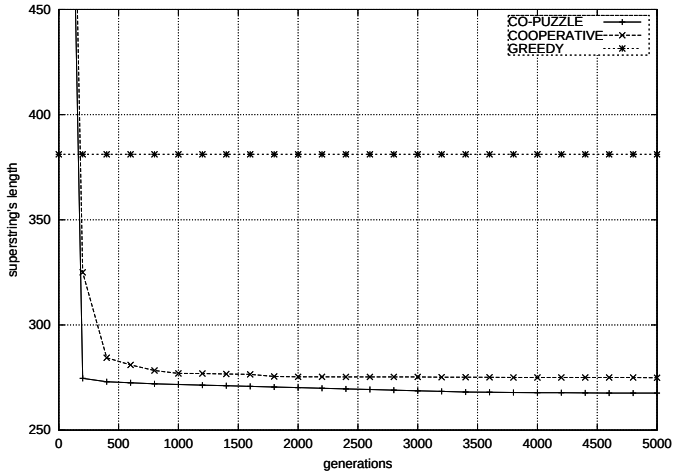


Figure 5.8: Experiment V: 50 blocks. Best superstring as a function of time. Shown are results for three algorithms: combination of the puzzle algorithm and cooperative coevolution(`CO-PUZZLE`), cooperative coevolutionary algorithm (`COOPERATIVE`), and GREEDY (`GREEDY`). Interpretation of graphs is as in Figure 4.3.

The results presented in Figure 5.9 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.

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.

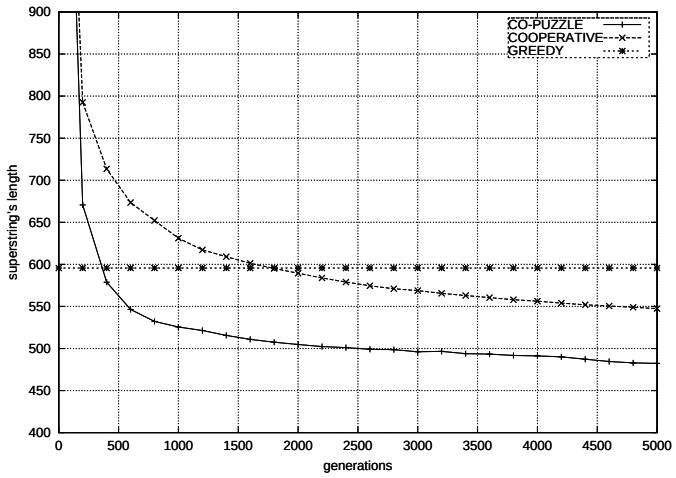


Figure 5.9: Experiment VI: 80 blocks. Best superstring as a function of time. Shown are results for three algorithms: combination of the puzzle algorithm and cooperative coevolution(`CO-PUZZLE`), cooperative coevolutionary algorithm (`COOPERATIVE`), and GREEDY (`GREEDY`). Interpretation of graphs is as in Figure 4.3.

5.5 Discussion

Figure 5.10 shows the 4 genetic algorithms presented in this dissertation, along with the relations between them. In [41] (Chapter 4) we presented the transformation from a simple GA to a cooperative GA (bottom-left arrow in Figure 5.10). In [40] (the current chapter) we added the other three arrows (transformations). These relations are general, in the sense that the same transformational diagram of Figure 5.10 applies not only to the SCS problem but to a variety of different problem domains.

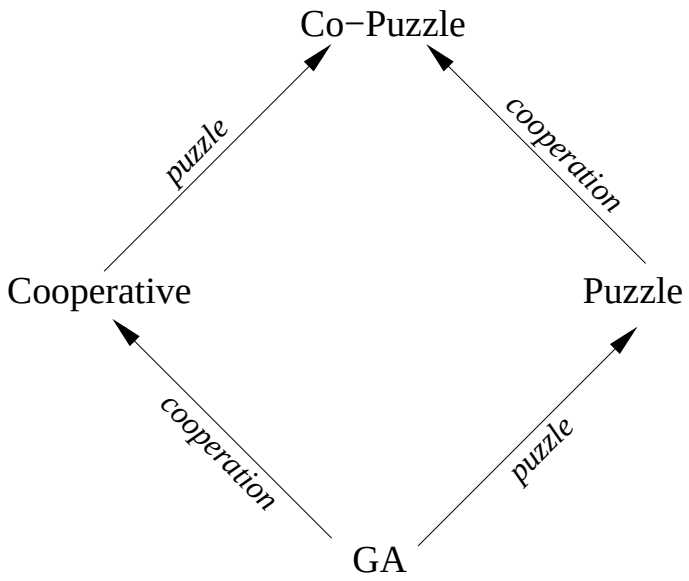


Figure 5.10: "Moving" between the algorithms presented in this paper.

A summary of the results we obtained is given in Table 5.1. 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

Table 5.1: Best average results (along with bitwise distance from optimum in parenthesis) obtained by five algorithms presented in this dissertation: GREEDY, GA, cooperative coevolution, Puzzle, and Co-Puzzle. For each of the fifty 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)**

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 2.3, using the setup from Section 3.4. The Co-Puzzle algorithm's setup is as detailed in Subsection 5.2.

The results, presented in Table 5.2, 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).

The Puzzle approach undoubtedly improves a simple GA (at least for the SCS problem). There are numerous other relative-ordering problems (some

Table 5.2: 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)**

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 the SCS problem. 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 of 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 [10, 11].

Chapter 6

Proposal: The Messy Puzzle Algorithm

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

6.1 A Brief Introduction to Messy GAs

In this section, we briefly review the original messy GA. Readers interested in more details should consult other sources (Messy GAs [11], Fast Messy GAs [10]). 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, Korb, and Deb [11] to eliminate some major problems of the standard GA. In [10], 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, in-

cluding permutation-based problems [16].

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* 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 $< allele\_locus, allele\_value >$. Thus, a "messy" genome may be *over-specified* when more than one gene corresponds to the same allele locus, or *under-specified* when certain genes are missing. To evaluate over-specified genomes only the first appearance of a gene is considered. When evaluating under-specified 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
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 down side of having all these building blocks is that each must be evaluated. Since the number of these building blocks is huge it forms a bottle-neck for the mGA. In [10], 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 sub-functions. 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.

6.2 Messy GAs 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 non-consecutive 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 should solve some of the deficiencies of the mGA:

- 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 need 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.

Chapter 7

Concluding Remarks and Future Research

In this dissertation we presented a number of novel coevolutionary algorithms designed for the SCS problem and empirically compared their performance on instances inspired by DNA sequencing. Moreover, we discussed the benefits of using our methods in the general field of evolutionary algorithms.

Our conclusions are as follows:

- Cooperative coevolution might prove deleterious when too many species are used. 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. When the problem is "easy" for a cooperative coevolutionary algorithm that uses less species, adding more species only proves harmful.

- When a suitable number of species are used, cooperative coevolution improves performance. We believe the main reason behind this suc-

cess 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 (or the Puzzle algorithm).

- The Puzzle approach undoubtedly improves a simple GA. The crucial idea of adding a building-blocks population to a standard GA in order to aid in the selection of recombination loci improves drastically the performance of a standard GA on the SCS problem and may well prove beneficial for a whole slew of problems. 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.

- Cooperation between cooperative coevolution and Puzzle ultimately improves global performance. Using the Puzzle algorithm instead of a standard GA to evolve the species in a cooperative coevolutionary algorithm increases the quality of the solutions found for each subproblem, thus increasing the quality of the global solution.

Our work opens up several avenues for future research:

- Scaling analysis of cooperative coevolution. How exactly a cooperative coevolutionary algorithm performs as a 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. Our conjecture is that the behavior will turn out to be as in Figure 7.1.

- Tackling larger problem instances using the Co-Puzzle algorithm with a mutable number of species. Toward this end, finding the relation

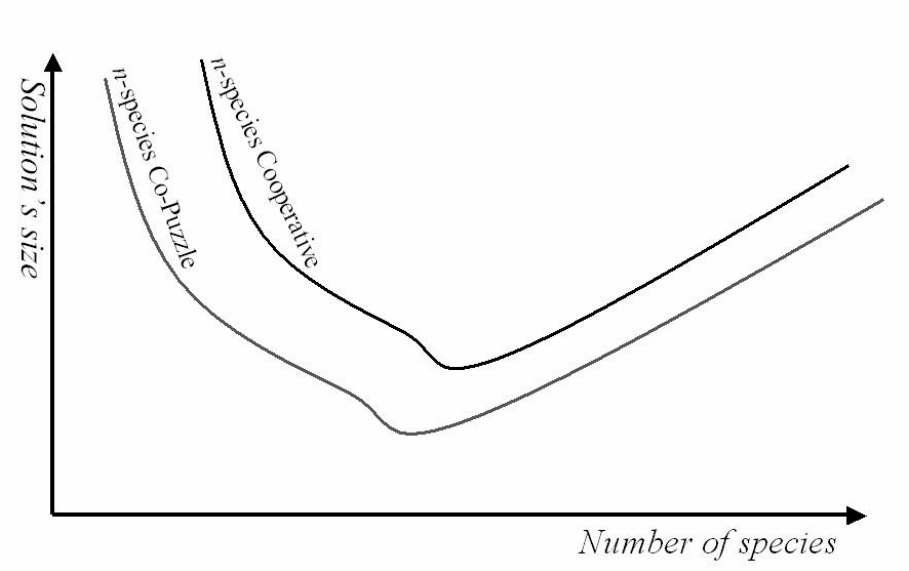


Figure 7.1: Conjectured scaling analysis of cooperative coevolution. The graph shows the expected solution length as a function of the number of species used by a cooperative coevolutionary algorithm (with and without using the Puzzle approach) on a fixed problem size. As can be seen we expect there to be an optimal number of species for each algorithm; moreover, we conjecture that Co-Puzzle will be less "species-demanding" and will find better solutions.

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 [33]).

- 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.

- The Messy Puzzle Algorithm (Chapter 6).

Bibliography

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

[2] J. Branke, M. Middendorf, and F. Schneider. Improved heuristics and a genetic algorithm for finding short supersequences. *OR Spektrum*, 20:39–45, 1998.

[3] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J. J. Grefenstette, editor, *Proceedings of Internatinal Conference on Genetic Algorithms and their Applications*, pages 183–187. Lawrence Erlbaum Associates, 1985.

[4] C. Darwin. *On the Origin of Species by Means of Natural Selection; or, the Preservation of Favoured Races in the Struggle for Life.* John Murray, London, 1859.

[5] E. D. de Jong. Representation development from Pareto-Coevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*, pages 265–276. Springer-Verlag, 2003.

[6] E. D. de Jong and T. Oates. A coevolutionary aproach to representation development. In E. D. de Jong and T. Oates, editors, *Workshop on*

Development of Representations – ICML-2002, Sydney NSW 2052, 2002. Online proceedings: http://www.demo.cs.brandeis.edu/icml02ws.

[7] R. Eriksson and B. Olsson. Cooperative coevolution in inventory control optimisation. In G. D. Smith, N. C. Steele, and R. F. Albrecht, editors, *In Proceedings of 3rd International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA 1997)*, Norwich, UK, 1997. Springer.

[8] S. B. Gabriel *et al.* The structure of haplotype blocks in the human genome. *Science*, 296(5576):2225–2229, 2002.

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

[10] D. E. Goldberg, K. Deb, H. Kargupta, and G. Harik. Rapid accurate optimization of difficult problems using fast messy genetic algorithms. In S. Forrest, editor, *Proc. of the Fifth Int. Conf. on Genetic Algorithms*, pages 56–64, San Mateo, CA, USA, 1993. Morgan Kaufmann.

[11] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1990.

[12] F. Hants, editor. *Cybernetic solution path of an experimental problem*, August 1965. English translation of lecture given at the Annual Conference of the WGLR at Berlin in September, 1964.

[13] L. Helmuth. Genome research: Map of the human genome 3.0. *Science*, 293(5530):583–585, 2001.

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

[15] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan, 1975.

[16] D. Knjazew and D. E. Golberg. OMEGA — Ordering messy GA: Solving permutation problems with the fast messy genetic algorithm and random keys. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 181–188, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[17] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Eleventh International Joint Conference on Artificial Intelligence*, pages 768–774. Morgan Kaufmann, 1989.

[18] A. Lesk, editor. *Computational Molecular Biology, Sources and Methods for Sequence Analysis.* Oxford University Press, Oxford, 1988.

[19] R. Michel and M. Middendorf. An ACO algorithm for the shortest common supersequence problem. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 51–61. McGraw-Hill, London, 1999.

[20] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, 1996.

[21] D. E. Moriarty. *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks.* PhD thesis, University of Texas, Austin, 1997.

[22] D. E. Moriarty and R. Miikkulainen. Evolving obstacle avoidance behavior in a robot arm. In *Proceedings of the Forth International Conference on Simulation of Adaptive Behavior*, Cape Code, MA, 1996.

[23] J. Paredis. Coevolutionary constraint satisfaction. In Y. Davidor, H-P. Schwefel, and R. Manner, editors, *Proceedings of the Third Conference on Parallel Problem Solving from Nature, Lecture Notes in Computer Science*, volume 866, pages 46–55, Berlin, Heidelberg, 1994. Springer Verlag.

[24] J. Paredis. Steps towards coevolutionary classification neural networks. In R. Brooks and P. Maes, editors, *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems (Artificial Life IV)*, pages 102–108, Cambridge, MA, 1994. MIT Press/Bradford Books.

[25] J. Paredis. Coevolutionary computation. *Artificial Life*, 2(4):355–375, 1995.

[26] J. Paredis. The symbiotic evolution of solutions and their representations. In L. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 359–3650, San Mateo, CA, 1995. Morgan Kaufmann.

[27] J. Paredis. Coevolutionary process control. In G. D. Smith, editor, *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA97)*, Vienna, 1997. Springer.

[28] C.-A. Peña-Reyes and M. Sipper. Applying Fuzzy CoCo to breast cancer diagnosis. In *Proceedings of the 2000 Congress on Evolutionary Computation (CEC00)*, volume 2, pages 1168–1175. IEEE Press, Piscataway, NJ, 2000.

[29] C.-A. Peña-Reyes and M. Sipper. The flowering of Fuzzy CoCo: Evolving fuzzy iris classifiers. In V. Kurková, N. C. Steele, R. Neruda, and

M. Kárný, editors, *Proceedings of 5th International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA 2001)*, pages 304–307. Springer-Verlag, Vienna, 2001.

[30] C.-A. Peña-Reyes and M. Sipper. Fuzzy CoCo: A cooperative-coevolutionary approach to fuzzy modeling. *IEEE Transactions on Fuzzy Systems*, 9(5):727–737, October 2001.

[31] M. A. Potter. *The Design and Analysis of a Computational Model of Cooperative Coevolution*. PhD thesis, George Mason University, 1997.

[32] M. A. Potter and K. A. De Jong. A cooperative coevolutionary approach to function optimization. In *Proceedings of the Third Conference on Parallel Problem Solving from Nature (PPSN III)*, pages 249–257, Jerusalem, Israel, 1994. Springer-Verlag.

[33] M. A. Potter and K. A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, Spring 2000.

[34] C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

[35] M. Sipper. A success story or an old wives' tale? On judging experiments in evolutionary computation. *Complexity*, 5(4):31–33, March/April 2000.

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

[37] J. Storer. *Data Compression: Methods and Theory*. Computer Science Press, Rockville, Maryland, 1988.

[38] Z. Sweedyk. A 2.5-approximation algorithm for shortest superstring. *SIAM Journal of Computing*, 29(3):954–986, December 1999.

[39] J. Turner. Approximation algorithms for the shortest common superstring problem. *Information and Computation*, 83:1–20, 1989.

[40] A. Zaritsky and M. Sipper. The preservation of favoured building blocks in the struggle for fitness: The puzzle algorithm. submitted to *IEEE Transactions on Evolutionary Computation*, November 2003.

[41] A. Zaritsky and M. Sipper. Coevolving solutions to the shortest common superstring problem. *BioSystems*, 2003. (to appear; draft version available at http://www.cs.bgu.ac.il/∼assafza).