Classical-Sorting Embedded in Genetic Algorithms for Improved Permutation Search

Vladimir Estivill-Castro
Department of Computer Science and Software Engineering
University of Newcastle
University Drive, Callaghan
NSW 2308 Australia
vlad@cs.newcastle.edu.au

Rodolfo Torres-Velázquez
Department of Computer Science and Software Engineering
University of Newcastle
University Drive, Callaghan
NSW 2308 Australia
rodolfo@cs.newcastle.edu.au

Abstract- A sorting algorithm defines a path in the search space of $n!$ permutations based on the information provided by a comparison predicate. Our generic mutation operator for hybridization, blends a hill-climber and follows the path traced by any sorting algorithm. Our proposal adds search (exploitation) capability to the mutation operator. Mutation requests swaps to construct and test new permutations, while the sorting algorithm supplies suggestions for swapping pairs as comparison to perform. The need to compare pairs of items in sorting is fulfilled by evaluating a guiding function. This new *HGA-sorting*, hybrid instantiated with Insertion Sort, dramatically improves previous results for a benchmark of experiments of the Error-Correcting Graph Isomorphism.

1 Introduction

In combinatorial optimization, simple Genetic Algorithms are not usually considered as good as domain specific methods, in terms of efficiency and quality of solutions [Grefenstette et al., 1985, Davis, 1991, Ulder et al., 1994]. However plenty of experimental evidence has shown that combining Genetic Algorithms (GAs) with other mechanisms, mainly with local-search methods, results in more effective optimization [Grefenstette et al., 1985, Liepins and Hilliard, 1987, Ulder et al., 1994, Estivill-Castro, 2000].

Analytical and experimental attempts have been made to explain those results [Whitley, 1995, Yamada and Reeves, 1997, Goldberg and Voessner, 1999]. For example, based on small benchmark problems, Yamada et al. [Yamada and Reeves, 1997] sustain that local optima occur in clusters on the fitness landscape. Thus, an Hybrid GA with local search, which explores around local optima, is able to find better optima. Nevertheless, in general terms, there are no guarantees, and the exact nature of the fitness landscape is highly dependent, in a dynamic way, on every factor involved in the mechanism applied to solve the problem [Culberson and Lichtner, 1997]. Several theoretical models are actually extensions of *simple* GA models and they share with those their virtues and limitations. It is widely recognized that, in this respect, fundamental questions remain open [Mitchell, 1996].

In this paper we obtain more effective genetic algorithms for optimization problems involving permutations. We achieve this through hybridization with classical sorting algorithms. The search for the permutation that optimizes some criteria is a genetic search. However, we dramatically improve this search with a hill-climber in the mutation operator. The sorting algorithms directs the hill-climber, (rather than a discrete gradient). We obtain improved results for a benchmark of experiments of the *Error-Correcting Graph Isomorphism*. This problem seems so hard that in this case, genetic algorithms do perform better than several other optimization strategies [Wang et al., 1997]. We implemented our proposal to solve this problem and the results of our experiments show a marked improvement over previous optimization with GAs. Since our proposal is more generic that the case study, we present a discussion of the rationale behind it, and its effectiveness.

2 Sorting Guides the Hill-Climber

Including local-search in Genetic Algorithms raises the issue of where to insert the local-search mechanism. The solutions proposed belong to three categories. The first group of proposals extends the capabilities of the crossover operator by means of local searching [Grefenstette et al., 1985, Liepins and Hilliard, 1987, Yamada and Reeves, 1997]. The second one, applies local-searching to some, or all, of the chromosomes in the population, as a separated and independent step of the GA mechanism [Ulder et al., 1994, Whitley, 1995]. The last one, uses mutation as a local search operator [Davis, 1991, Estivill-Castro and Torres-Velázquez, 1999, Estivill-Castro, 2000]. Our proposal here belongs to the last group.

2.1 Classical-Sorting as a Local-Search Mechanism

A classical and well understood mechanism to deal with a class of permutation problems is the family of sorting algorithms [Knuth, 1973].

Although it is an unusual point of view, classical sorting algorithms can be conceptualized as local-search mechanisms, where the local information provided by each pair of elements in the string is enough to decide, in a deterministic way, the search direction. For example consider the predicate

$<: X \times X \rightarrow \{True, False\}$ defined by

$$< (x_1, x_2) = \begin{cases} True & \text{if } x_1 < x_2 \\ False & \text{otherwise} \end{cases}$$

(where the set X is a total order). Then, for a given sequence $\langle x_1, \ldots, x_n \rangle$ of n items in X, classical sorting corresponds to the following problem [**sorting inversions**]:

$$\text{minimize } Inv(\pi) = Inv(\langle x_{\pi(1)}, \ldots, x_{\pi(n)} \rangle).$$

where $Inv(\pi)$ is a measure of disorder [Knuth, 1973, Estivill-Castro et al., 1993] that counts the total number of pairs in the wrong order (that is, the number of *inversions*, where an inversion is a pair $i < j$ where $< (x_{\pi(i)}, x_{\pi(j)})$ is $False$).

It has long been known that, this inversion-minimization problem can be solved optimally in $\Theta(n \log n)$ time in the worst case by *Heapsort* and *Merge sort* and in $\Theta(n \log n)$ time in the average case by *Quicksort*. However, if sorting operations are restricted to swaps of adjacent elements (i.e. the sequence of items is stored in a double-linked list), then the best algorithm is Insertion Sort, requiring $n + Inv(\langle x_1, \ldots, x_n \rangle)$ comparison and swaps.

Thus, $Inv(\pi)$ is a function that monotonically reflects the disorder (the distance from the optimum) of a given permutation with respect to swaps of adjacent elements. Of course this works in the case that we can apply simple operations to permutations and we have a function that can assess if that local change is a step in the right direction. For an operator ϕ and minimization problem Q, a function f will be called a monotonic function if, for any given permutation π, $f(\pi) > f(\phi(\pi))$ if and only if, $\phi(\pi)$ is better solution than π.

We illustrate this once more with our running example in which we are using sorting as an optimization problem. Consider the following version [**sorting runs**]:

$$\text{minimize } Runs(\pi) = 1 + \|\{(i, i+1) \mid 1 \leq i < n \wedge \neg < (x_{\pi(i)}, x_{\pi(i+1)})\}\|.$$

In this version, sorting corresponds to minimizing the number of ascending runs (the optimal value is $Runs = 1$ when the sequence is sorted). It is not hard to see that Inv is not a monotonic function for swaps of adjacent items in the sorting runs version. For example, consider the permutation $\pi = \langle 1, 4, 5, 2, 3 \rangle$, and let the operator ϕ be the first swap performed by Insertion Sort. Then, the next permutation in the search space visited by Insertion Sort is $\phi(\pi) = \langle 1, 4, 2, 5, 3 \rangle$. We see that this swap has increased the number of runs. However, consider as operators the path in the search space by Natural Merge Sort [Knuth, 1973]. This algorithm iterates a pass over the input sequence where it merges the first and second run, then the 3rd and 4th run, and so on. Each pass reduces the number of runs by half (almost). When a pass produces only one run, Natural Merge Sort terminates. Clearly, the guiding function of Natural Merge Sort is $Runs$. More interestingly, for each pass by Natural Merge Sort over the sequence, $Runs$ is monotonic with respect to sorting inversions. That is, monotonically reducing $Runs$ with Natural Merge Sort monotonically reduces Inv.

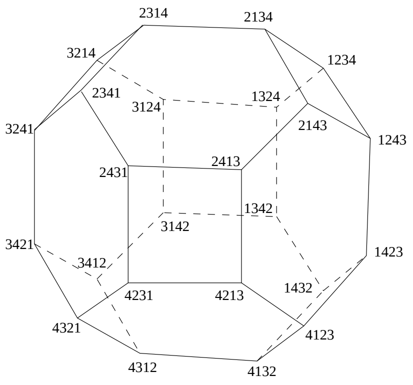

Figure 1: The search space of 4! permutations arranged into a graph by swaps of adjacent items.

Finding such monotonic functions is usually not an easy task, specially for other harder combinatorial optimization problems that search for an optimal permutation among the $n! = \Theta(n^n)$ possible alternatives. In our illustration example, with sorting items from a doubly linked list, Insertion Sort monotonically reduces Inv to zero, traveling a direct descent and shortest path in this space of $n!$ permutations (by contrast *Heapsort* will initially introduces more inversions on an already sorted sequence when constructing a heap). Figure 1 displays the space of $n!$ permutations in the case of $n = 4$ and where swaps of adjacent items in the sequence are the operators.

Thus, a sorting algorithm defines paths in the search space of n! permutations, from any permutation to the goal permutation. For instance, Figure 2 shows the search tree (the set of possible paths) for 4! permutations, build by the Insertion Sort mechanism. Testing the value of the predicate $< (x_{\pi(i)}, x_{\pi(j)})$ defines the path. For any permutation π, the Insertion Sort algorithm visits permutations in the left branch from the node for π, all the way down to a leaf. We use the embed search direction of the implicit paths into a hybrid genetic algorithm.

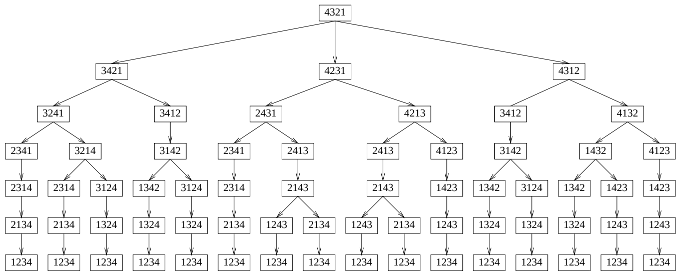

Figure 2: The search tree for 4! permutations built by the Insertion Sort mechanism

2.2 Classical-Sorting as Director of Mutation

Although the precise role of crossover and mutation in GAs has been debated [Spears, 1993], it is not our purpose to be part of this debate. Our aim, instead, is to facilitate the complex interaction between mutation and crossover by means of classical-sorting.

In GAs, the concept of searching for permutations has been present since 1985 [Davis, 1985, Goldberg and Lingle, 1985]. Naively crossing or mutating the chromosomes of permutations can produce invalid representations, so *compatible order-based* operators are designed to assure that the offspring is a valid representation of a permutation. However, there are several proposals for these order-based operators. In the case of crossover, variants try to preserve ordering properties (adjacency, absolute position, or relative order of elements) of the parents to be passed to the offsprings [Starkweather et al., 1991, Escalada-Imaz and Torres-Velázquez, 1997]. It has been empirically shown that the effectiveness of these operators is highly dependent on the nature of the problem.

For mutation, also several compatible order-based operators that preserve the valid representation of a permutation have been proposed [Davis, 1991]. For example, the swap mutation operator, used by Wang et al. [Wang et al., 1997], chooses a uniform randomly pair of positions in the chromosomes and swap the contents of the selected positions. However, this operation often produces a chromosome with worse performance. This is not considered a problem when mutation operator is conceived as a diversity generator. By contrast, we apply the results of the mutation operator only if we achieve better performance. In this sense our mutation operator can be seen as a hill-climber.

In our approach, the sorting algorithms works as a map for the search space of $n!$ permutations. Thus, when a mutation is to be applied with given mutation probability p_m, the genetic algorithms will be told what mutation to apply. However, the sorting algorithm determines its path as the result of comparing items. This feedback to the sorting algorithms comes, hopefully, from a monotonic function with respect to the operators of the sorting algorithm and the minimization problem at hand.

Note that, in simple genetic algorithms with binary representation, a random swap is applied to each position with probability p_m. This operator samples all the search space, but with much higher probability near the chromosome that is being mutated. For example, if we consider a binary chromosome of length l, with probability

$$\left(\begin{array}{c} h \\ l \end{array} \right) p_m^h (1-p_m)^{l-h},$$

we obtain a chromosome at Hamming distance h from the original chromosome. This mutation operator is uniform anywhere in the search space and it blind to what was the result of previous mutations. It is fine as a operator to preserve diversity, and in order not to radically disrupt the current solutions

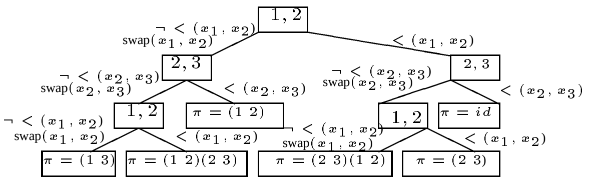

Figure 3: Insertion Sort as an algorithm for searching the permutation π such that $\pi(\langle x_1, x_2, x_3 \rangle)$ is sorted.

and upset convergence, it has a bias to produce chromosomes near the originator (specially, since p_m is much closer to 0 that to 1 and $1-p_m$ is much closer to 1). It does not require any memory and it is oblivious to where in the search space the chromosomes are. Some of this blindness is corrected by approaches that change the mutation probability as a function of the number of generations, similarly to a learning rate in neural networks or a temperature argument in simulated annealing.

Our approach is to add search (exploitation) capability to the mutation operator. We consider a sorting algorithm as a director for which mutation operation to perform. Because of this, we refer to our mutation operator as *SORT*.

All sorting algorithms define a decision tree. This is a binary tree where nodes are labeled by the two indices of the items to be compared. In a sorting algorithm, after a comparison, two possible outcomes define edges where the computation continues. The edges may involve some operation, like a swap, where the algorithm encodes the permutation. Figure 3 shows a comparison tree for Insertion Sort on sequences of length 3. Figure 3 also shows the actions (the swaps), but formally the decision tree is just the schema of comparisons. Leaves indicate the permutation in the original sequence. The essence of a sorting algorithms is the decision tree. It specifies the comparisons that follow as a result of previous comparisons in order to determine the permutation of the original input. In fact, this is how the $\Omega(n \log n)$ lower bound on the number of comparisons required for sorting is proved.

Note that, if a sorting algorithm were to operate in parallel on an array a where initially $a[i] = i$ (for $i = 1, \ldots, n$), then, on completion of the sorting, the array a encodes the permutation of the input. For example, on input sequence $\langle 10, 15, 3 \rangle$, Insertion Sort will first swap the last two items. The new sequence is $\langle 10, 3, 15 \rangle$, and array a is $[1, 3, 2]$. Insertion Sort will complete the sorting when it swaps the second and first item, and thus, array $a = [3, 1, 2]$. This encodes the original permutation, as $a[i] = j$ means the i-th smallest is in the j-th position of the input (for example, the smallest is in the 3rd position of $\langle 10, 15, 3 \rangle$).

Our mutation operator requests direction from a sorting algorithm. That is, our mutation operator does not randomly swap items of a chromosome encoding a permutation. That would be the spirit of a memoryless mutation which works obliviously to what has been learned in the past. Our mutation actually request from the sorting algorithm which permutation to construct from the current permutation. The sorting algorithm will suggest operators ϕ to mutate the current permutation π into $\phi(\pi)$.

Conceptually, the sorting algorithm and the mutation operator (in the genetic algorithm) interact. The mutation operator will indicate success to the sorting algorithm when $f(\pi) \geq f(\phi(\pi))$ (for a minimization problem). For example, if ϕ were a swap operator of the items in position i and j, then the search learns that the item in position j goes before the item in position i, in very much the same way as a sorting algorithm learns the relative order of two values when performing a comparison. If $f(\pi) < f(\phi(\pi))$, then feedback is also supplied, indicating that the operator ϕ proposed by the sorting algorithm was unsuccessful.

Any comparison-based sorting algorithm can participate in this hybridization, simply, when about to compare the current values at the i-th with j-th positions, the proposed operation is the swap of items in these positions. An improved optimization is to be interpreted by the sorting algorithm as the comparison returning *True*, and otherwise as the comparison returning *False*.

We call our hybridization strategy *HGA-sorting*. The impact of the sorting algorithm into the mutation, and thus into *HGA-sorting* is to be regulated in two dimensions. The first dimension is the amount of progress in the state of the sorting algorithm per mutation. The second dimension is the proportion of sorting algorithm states to chromosomes in the population. We now explain these two dimensions.

Note that the sorting algorithm maintains its state, namely, in which node of its decision tree is currently operating. So, when a mutation is to be applied to a chromosome, the sorting algorithm resumes from its current state. On *minimal progress mode*, the sorting algorithm would propose an operator ϕ. If $f(\pi) \geq f(\phi(\pi))$, then the mutation is accepted and the sorting algorithm receives a success signal as if the corresponding comparison had resulted in *True*. For the purposes of the sorting algorithm, it now chooses the branch of operation of the successful comparison but stops there. If $f(\pi) < f(\phi(\pi))$, then the mutation is rejected (it is not applied). The sorting algorithm receives a failure signal as if the corresponding comparison returned *False*. It now chooses the alternative branch of operation and halts again. In this mode of operation the sorting algorithm advances at most one level in the depth of its decision tree.

However, for this same dimension, the sorting algorithm could be on *maximal progress mode*. Namely, if $f(\pi) \geq f(\phi(\pi))$, then the mutation is accepted and another operator is requested. A sequence of operators is applied to progressively mutate and improve more the current operator until $f(\pi) < f(\phi(\pi))$. At this point the mutation for that chromosome is terminated with the version that most improved the function f. The sorting algorithm advances its state, to the deepest node until a comparison produces no improvement.

Note that these two modes represent extremes. In minimal progress mode, only one hill-climber step in the direction suggested by the sorting algorithm is taken. In maximal progress mode, steps in the direction suggested by the sorting are taken until such suggestion fails. Naturally, it is possible to mediate between these extremes. For example, allowing an argument to determine a (deterministic or random) amount of progress in the sorting algorithm per mutation.

With respect to the number of states in the sorting algorithm, we can regard the sorting algorithm as multi-treaded, with one thread per chromosome, or as a single thread for the entire population. In the first extreme, each chromosome will carry within its encoding information regarding the state of its thread, and where mutation is to be applied to it, the sorting algorithm resumes accordingly. In the second extreme, there is only one state and although another chromosome is being mutated, the interaction with the sorting algorithm resumes from where the last mutation left.

Our idea allows sorting algorithms to provide with their decision tree, a map of the $n!$ space of permutations, and indicate what operations are to be made to move from anywhere in this space to the desired goal permutation (typically in subquadratic time!). Our hybrid GA does not search for permutations as blindly as a simple GA. Our hybridization allows it to determine facts like item i should be before item j. That is, it adopts how sorting algorithms learn the permutation to sort the input.

3 Harder Problems than Sorting

The design of our hybrid GA intends to apply it to harder problems than sorting, and in particular, in minimization problems searching for permutations. A benchmark problem emerges from the use of graphs in pattern matching.

Graphs are combinatorial objects that have been widely used in applications where structured objects emerge in a natural way. In what follows G is a graph, $V(G)$ denotes the vertices of G and $E(G)$ the edges of G. Remarkably, in the pattern-matching arena, modeling with graphs has been fruitfully used to match objects [Tsai and Fu, 1979, Shapiro and Haralick, 1981, Messmer and Bunke, 1999]. Thus, the interest in finding efficient algorithms to deal with the Graph Isomorphism problem.

Definition 1 (GI) *An* isomorphism *[West, 1996] from G to H is a bijection $F : V(G) \rightarrow V(H)$ such that $uv \in E(G)$ if and only if $F(u)F(v) \in E(H)$. We say "G is* isomorphic *to H", written $G \cong H$, if there is an* isomorphism *from G to H.*

Graph Isomorphism (GI) [Read and Corneil, 1977] is an important problem in Graph Theory whose precise computational complexity remains unknown [Kobler et al., 1993]. It requires to find a bijection F of the vertices so that the edge structure is the same. Labeling the vertices from the same set corresponds to finding a permutation π. However, it is not a minimization problem. The interesting aspect is that, in practice a close variant is a hard minimization problem. In real world applications of pattern matching, the existence of noise, distortion, uncertainty or measurement errors, together with weights associated to nodes and edges, translates the GI problem into its inexact version: the inexact Graph Isomorphism (iGI) or Error-Correcting Graph Isomorphism (ECGI). In order to define this problem we first need the notion of attributed graph.

Definition 2 (AG) *An* attributed graph *[Tsai and Fu, 1979] is a 4-tuple*

$$G_a = (V, E, \alpha, \beta)$$

where

- $V = G_a(V)$*, is a finite nonempty set of vertices;*
- $E \subset V \times V$*, is a set of distinct ordered pairs of distinct elements in* V *called edges;*
- $\alpha : V \to \Re$*, is a function called vertex interpreter;*
- $\beta : E \to \Re$*, is a function called edge interpreter.*

Definition 3 (ECGI) *The* Error-Correcting Graph Isomorphism *problem is that given two attributed graphs* ***AGs*** $G_a = (V(G), E(G), \alpha_G, \beta_G)$ *and* $H_a = (V(H), E(H), \alpha_H, \beta_H)$ *with* $V(G) = V(H)$*, we must find a permutation* $\pi : V(G) \to V(H)$ *so that some metric of total dissimilarity between the graph* $\pi(G_a)$ *and the graph* H_a *is minimized.*

To illustrate the virtues of our hybrid genetic algorithm we will describe its application to this harder combinatorial minimization problem. This is also a benchmark because GAs have previously shown to be effective [Wang et al., 1997].

3.1 A Benchmark of Graphs

We reproduce the experimental setting of Wang et al. [Wang et al., 1997]. Their construction of graphs for the ECGI problem constitutes a benchmark where to test the effectiveness of GAs. Thus, we encode an attributed undirected graph as an adjacency matrix. Let $n = |G_a(V)|$ be the number of vertices of G_a and $M(G_a) = [m_{i,j}]$ be the $n \times n$ adjacency matrix of an attributed undirected graph G_a; where $m_{i,j} = \alpha(v_i)$ if $i = j$, and $m_{i,j} = \beta(v_i, v_j)$, otherwise. Note that two graphs are isomorphic only if their adjacency matrix differ by permutations of rows and columns, that is, if $M(H_a) = P \cdot M(G_a) \cdot P^T$, where P and P^T are the matrix representation of a permutation π and its transpose, respectively (i.e. P is a permutation matrix).

The construction of Wang et al. assigns integer values in the interval $[0, 100]$ to each cell $m_{i,j}$ of the matrix $M(G_a)$, independently and uniformly (each integer entry $m_{i,j}$ is chosen with uniform probability $1/101$ and independently of other entries). In order to build the graph H_a (the isomorphic graph), a random permutation π is uniformly constructed (each permutation has probability $1/n!$). To do this, we use the well know procedure where we start with an array a initialized to $a[i] = i$, for $i = 1, \ldots, n$. Repeatedly, from $j = n$ down to 2, an index i is chosen uniformly in the integer interval $[1, j]$ and the values in $a[i]$ and $a[j]$ are swapped. We read π from the array a and construct the permutation matrix P by applying π to the columns of the identity matrix.

The graph H'_a has adjacency matrix $M(H'_a) = P \cdot M(G_a) \cdot P^T$. In order for the matching to be inexact, we add noise. For each entry in the upper triangle of $M(H'_a)$ an integer is selected uniformly in the interval $[-\nu, +\nu]$ (that is, with probability $1/(2\nu + 1)$), where $\nu \in [0, 20]$ is a parameter of the experiment regulating the approximate mismatch in the pair of graphs. In this process, noise is only added to the upper triangle and changes are reflected to keep an undirected graph. The resulting matrix is the adjacency matrix for the attributed graph H_a. The minimization algorithms will receive as input the pair G_a, H_a and we hope that they will recover π. Measuring to what extent the algorithms recover π allows to evaluate the quality of the optimization.

3.1.1 Algorithms for ECGI

Wang et al. show that GAs perform much better than branch-and-bound and other optimization approaches for the ECGI. They use an heuristic named *status matching* (a detailed description of this heuristic can be found in their paper [Wang et al., 1997]). While they left open how well this heuristic performs on its own, they do insert one chromosome obtained with status matching into the initial randomly-generated population of their GA. We have found that the heuristic gives a reasonable good approximated solution; better when noise levels are low (i.e. ν is close to 0). Wang et al. also left open how does their GA perform without this initial seeding of the population.

3.1.2 Fitness Function

In Definition 3, we left open the precise definition of a criterion of similarity between attributed graphs. Wang et al. conducted experiments with at least two fitness functions, and concluded that the absolute total error (**ATE**) in the entries of the adjacency matrices is more accurate.

Definition 4 ($ATE(\pi)$**)** *Given two attributed graphs* G_a *and* H_a*, the* Absolute Total Error *of a permutation* π *is the 1-norm of the matrix* $M(G_a) - P \cdot M(H_a) \cdot P^T$*, where* P *is the permutation matrix of* π*. The explicit form of the Absolute Total Error is given by*

$$ATE(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} |m_{i,j} - m_{\pi(i),\pi(j)}|.$$

Wang et al. re-write this as a maximization problem as finding a permutation π that maximizes $Max\_ATE(\pi)$ given by

$$Max\_ATE(\pi) = C'_{max} - ATE(\pi),$$

where C'_{max} is the maximum possible value of $ATE(\pi)$.

3.1.3 Our implementation of Hybrid-GA

We implemented our hybrid GA and left fixed most of the common aspects with Wang et al. to demonstrate that hybridization as proposed here is the source of the much-improved optimization. That is, our experiments are designed to illustrate that our mutation operator SORT achieves an hybridization with sorting methods that speeds up convergence to much better solutions.

The operators and parameters of the GAs used in our experiments are summarized in Table 1. Operators like *PMX* [Goldberg and Lingle, 1985] are well known in the GA community when applied to optimization problems searching for permutations. We will skip PMX details here. Similarly, we skip description of the Inhibitive Selection and the details of Ranking Scaling [Wang et al., 1997].

Parameter	*HGA-sorting*
Encoding Scheme	Integer Permutation
Population Size	30
Selection	Inhibitive
Scaling	Ranking Scaling
Crossover	PMX
Mutation	SORT
Crossover Probability	0.9
Mutation Probability	0.1

Table 1: GA Parameters

In our experiments, the mutation operator is the result of using Insertion Sort each time the mutation operator is requested. We use a single thread of the sorting algorithm, since this is the simplest implementation. The idea here is to illustrate that our proposed hybridization results in significant improvements even with its simplest variants.

For the amount of progress that the sorting algorithm is allowed to do on each mutation, we use an intermediate point between minimal progress and maximal progress. Recall that the j-th iteration of Insertion Sort inserts the $(j+1)$-th item into an already sorted subsequence in positions 1 to j (for $j = 1, \ldots, n-1$). We allow the sorting algorithm to progress one iteration per mutation. Thus, to recall the state of the one thread of Insertion Sort, the hybridization only needs one global variable j. When a mutation is to be applied to a chromosome encoding a permutation π, a sequence of new permutations is generated form π. This sequence of permutations is analogous to the path of permutations visited by Insertion Sort when scanning back for a place to insert the current $(j+1)$-th in the already sorted subsequence. The first permutation in this sequence is a permutation $\phi(\pi)$ that results from swapping the $(j+1)$-th item with the j-th item. The next permutation tested is constructed by inserting the $(j+1)$-th item in π in the $(j-1)$-th position. This is equivalent to swapping the $(j-1)$-th and j-th positions in $\phi(\pi)$. We call this new permutation $\phi(\phi(\pi))$. Naturally, $\phi(\phi(\phi(\pi)))$ is the result of swapping the $(j-2)$-th and $(j-1)$-th positions in $\phi(\phi(\pi))$. And so on, obviously, stopping when the original $(j+1)$-th item reaches the first position, or a comparison indicates the place for the $(j+1)$-th item is found. Then, the insertion is complete and the global variable j increases by one.

Now, recall that in our hybrid, the sorting algorithm does not compare but queries if $f(\phi(\pi))$ is a better value that $f(\pi)$. Thus, in our instantiation of *HGA-sorting* with Insertion Sort, for a mutation of a chromosome π evaluates $f(\phi(\pi))$, $f(\phi(\phi(\pi)))$ and so on, while this improves. If the sequence of values stops improving, or moving the original $(j+1)$-th item reaches to the front of the permutation π, the mutation is completed. On completion, the new chromosome is a new member of the population and the global variable j is increased by one. Note that, this mutation may not change π if in the first swap the value of $f(\phi(\pi))$ is worse than the value of $f(\pi)$.

The design of our experiment is based on swaps for the following reasons.

- *Spirit of Insertion Sort.* We expect that if we are testing i-th and $(j+1)$-th items and find and improvement, this means the current $(j+1)$-th item is actually to precede the i-th. But because we have been working with Insertion Sort, we want to preserve the relative order of the i-th element with all elements between the i-th and j-th position. Insertion Sort assumes it has items in the first position to j-th already sorted.

- *Efficiency.* It is actually efficient to perform the swaps in the representation of the current permutation. What we want to do is heuristically test the order of the i-th and $(j+1)$-th entry in the current permutation.

- *Non-monotonicity*. This gives an environment in which asses the effectiveness of our ideas with non-monotonic functions with respect to the operator ϕ.

- *Behavior with other sorting algorithms.* When hybridizing with any other sorting algorithm, the next comparison suggests a swap (maybe of non-adjacent items), and so the strategy generalizes trivially to other sorting algorithms.

In order to carry out our experiments, we used the software *GAlib* genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology.

3.2 Experimental Results

We use the fitness value of the best chromosome π_{found} in the final population to assess the quality of solutions. We compute the ratio between $Max\_ATE(\pi_{found})$ and the optimum value $Max\_ATE(\pi^*)$ (the permutation π^* is the original permutation used to construct H_a from G_a). This ratio is called *correctness* as is given by

$$correctness = 1 - \frac{|Max\_ATE(\pi_{found}) - Max\_ATE(\pi^*)|}{Max\_ATE(\pi^*)}.$$

Applying the methodology described in Section 3.1, we constructed 50 pairs of graphs for each integer value of $\nu \in [0, 20]$, and each size value $n \in \{10, 15, 20\}$. For each graph pair we executed three times the GA and selected the best solution found. So, we generated 3150 test cases, and executed

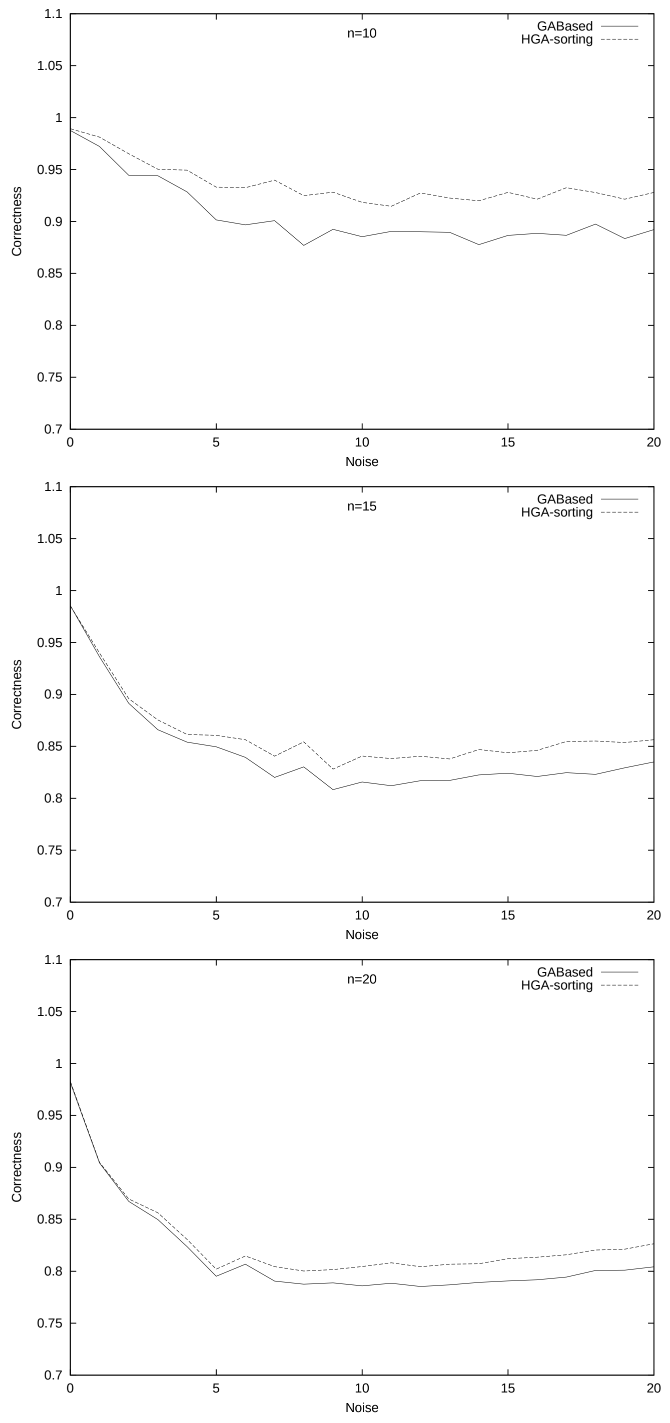

Figure 4: Correctness without elitism. Plots for 3 graph sizes. Average for 50 graph pairs.

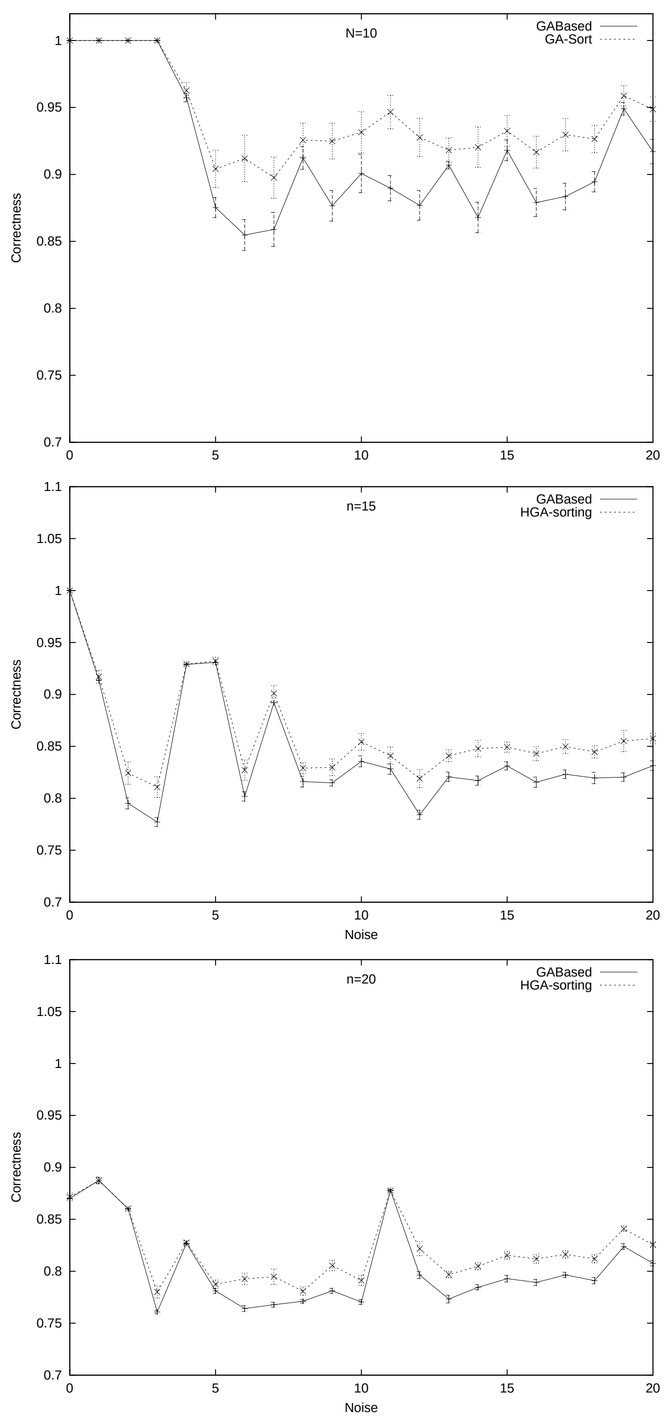

Figure 5: Confidence intervals of 99%. Average of 50 runs, using one problem instance.

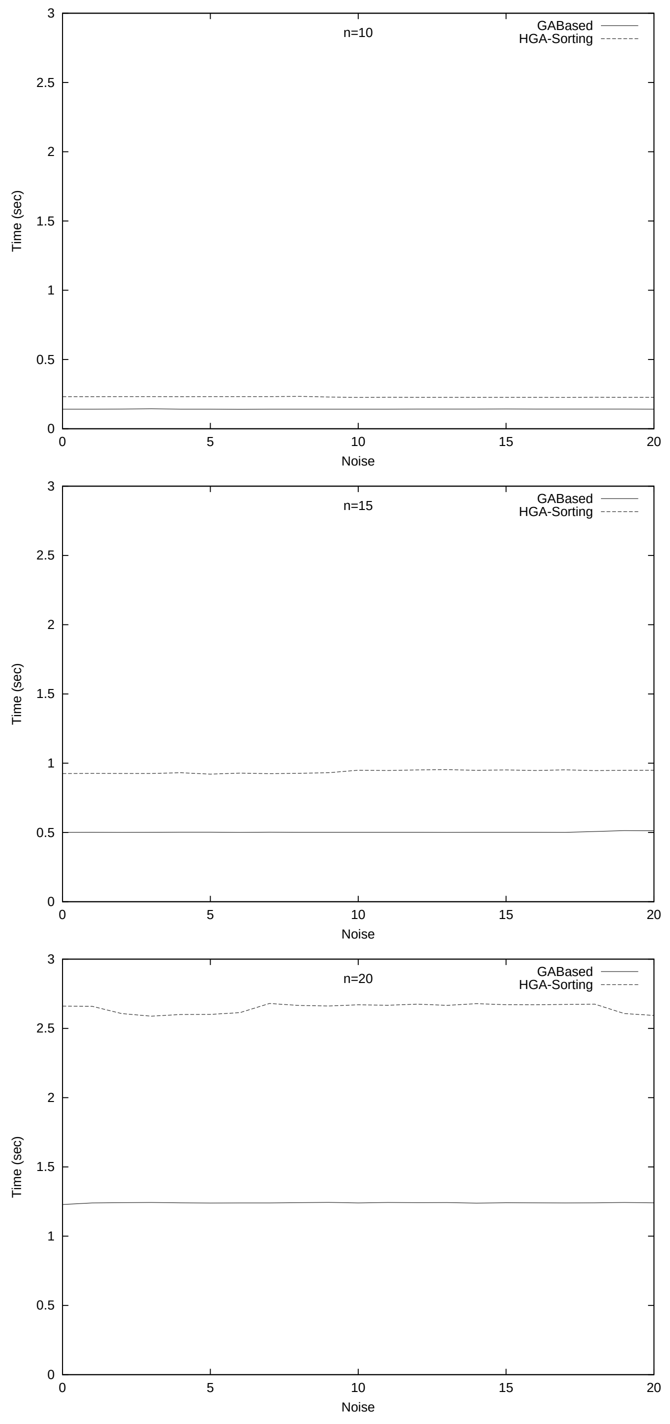

Figure 6: Execution time.

9450 times each Genetic Algorithm. Plots of the results are displayed in three plots in Figure 4. The plots correspond, from top to bottom, to the three values for the size n of the graph. In the plot, the x-axis shows the range of ν values. The y-axes is a correctness scale, from, 70% correct to 100% correct. The plotted lines are average *correctness*, taken over the 50 problems. The dotted line corresponds to results for our *HGA-sorting* instantiated with Insertion Sort (labeled in the figure as *HGA-sorting*), while the solid lines corresponds to Wang et al. algorithm (labeled GABased). This figure illustrates that our *HGA-sorting* with Insertion Sort performs much better. The difference in statistically significant for all graph sizes and for all $\nu > 5$. Moreover, for graph size $n = 10$, the improvement is statistically significant across the entire range of noise values ν. Therefore, we conclude that our method is consistently better for a broad spectrum of test cases, for different size graphs, and for medium and high noise levels.

Figure 5 illustrates the statistical significance of our results. Again, plots from top to bottom correspond to the three values for the size n, while the x-axis correspond to the range of noise values ν. However, what we see here are results for only one of the 50 graphs generated. For this problem instance, each version of the genetic algorithm is executed 150 times. The algorithms compared are *HGA-sorting* instantiated with Insertion Sort (labeled as *HGA-sorting*), and the GA of Wang et al. (GABased). For each three consecutive executions, the best solution found is adopted as the solution of the method. The lines represent the average of the 50 solutions produced by each method. The intervals plotted are 99% confidence interval, and show that that *HGA-sorting* is better for $\nu > 5$ with very high statistical significance (the intervals do not overlap, which is even higher than 99% significant).

The status matching heuristic produces remarkable results of noise values $\nu < 5$. Thus, it is natural that hybridization as proposed here offers little improvement. We already pointed at that Wang et al. abstained to evaluate this heuristic alone, but our experiments reveal that for $\nu < 5$, this heuristic does remarkably well. In fact, for $\nu = 0$ (non noise), the heuristic produces correctness above 98%. This means that in 100 problem instances with graphs of size 10, this seeding heuristic obtains the required permutation 98 times. So, in this case, neither of the genetic algorithms can do much improvement after the application of status matching.

Figure 6 shows the comparison of CPU time requirements. It is clear that hybridization adds overhead, but the plots show that, this overhead is constant for the range of ν values, and that it also remains constant and acceptable in the dimension of n values. Thus, the *HGA-sorting* instantiated with Insertion Sort is effective, obtaining better quality solutions for comparable requirements of CPU time.

3.3 Elitism

The previous set of experiments adhere in as much as described by Wang el at [Wang et al., 1997]. The only difference being our proposed hybridization with sorting.

However, it is reasonable to explore also if hybridization is also effective in other conditions, in particular, Wang et al. use a series of sophistication but do not use elitism. The plots in Figure 7 are the corresponding average performance plots when both algorithms use elitism. Once again, *HGA-sorting* instantiated with Insertion Sort (labeled in the figure as *HGA-sorting*), comes as a clear winner. Similarly, Figure 8 replicates the experiment to assess statistical confidence when both use elitism. Thus, with very strong statistical confidence, beyond 99%, our results show *HGA-sorting* instantiated with Insertion Sort is better. Finally, Figure 9 confirms again that the CPU time overhead of hybridization is well worth it, since it is just a small constant.

4 Discussion

The first remarkable point derived from this work is the competitive performance obtained when classical-sorting guides the mutation operator, especially for medium and high noise levels.

Secondly, this work opens the possibility of using *HGA-sorting* with functions with different levels of monotonicity, with the clear purpose of gaining in understanding about the complex relationship between mutation and crossover in GAs.

Finally, we conjecture that the main reason for the better performance, is that this mechanism is spreading and regulating the use of mutation. Mutation is not a blind random operator, but in problems that search for permutations, we can balance exploitation and exploration more wisely. The sorting mutation proposed here is directing the exploration on the pattern of the sorting algorithm and avoiding the inefficiency of randomly exploring swaps or small changes that may have already shown to be non-productive.

While the main idea in this paper is generic to any sorting algorithm, we have also observed that the proposed hybridization with Insertion Sort has a regulatory effect on mutation. To describe this effect, we recall that the literature of GAs has recommended to adjust the probability of mutation during a run. Typically, the suggestion is to increase mutation probability as the generations progress. For later generations, the population is close to convergence and diversity is hard to obtain unless the mutation probability is increased. Thus, the intention is to maintain the exploration capability of the GA. This effect is automatically achieved by our hybridization since initially, the values of the global variable j, regulating the amount of progress the algorithm performs are small. Thus, the mutation performs small changes. As the value of j grow, a single call to the mutation operator allows more radical changes.

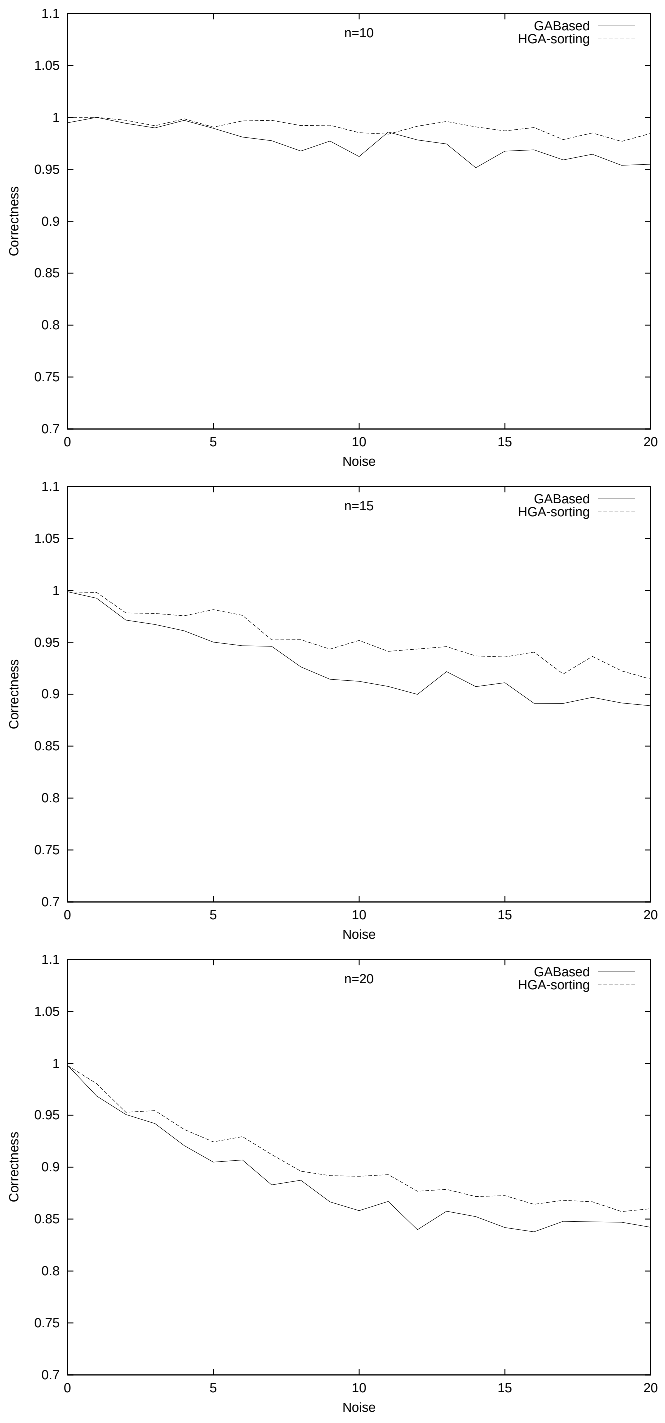

Figure 7: Correctness with elitism. Plots for 3 graph sizes. Average for 50 graph pairs.

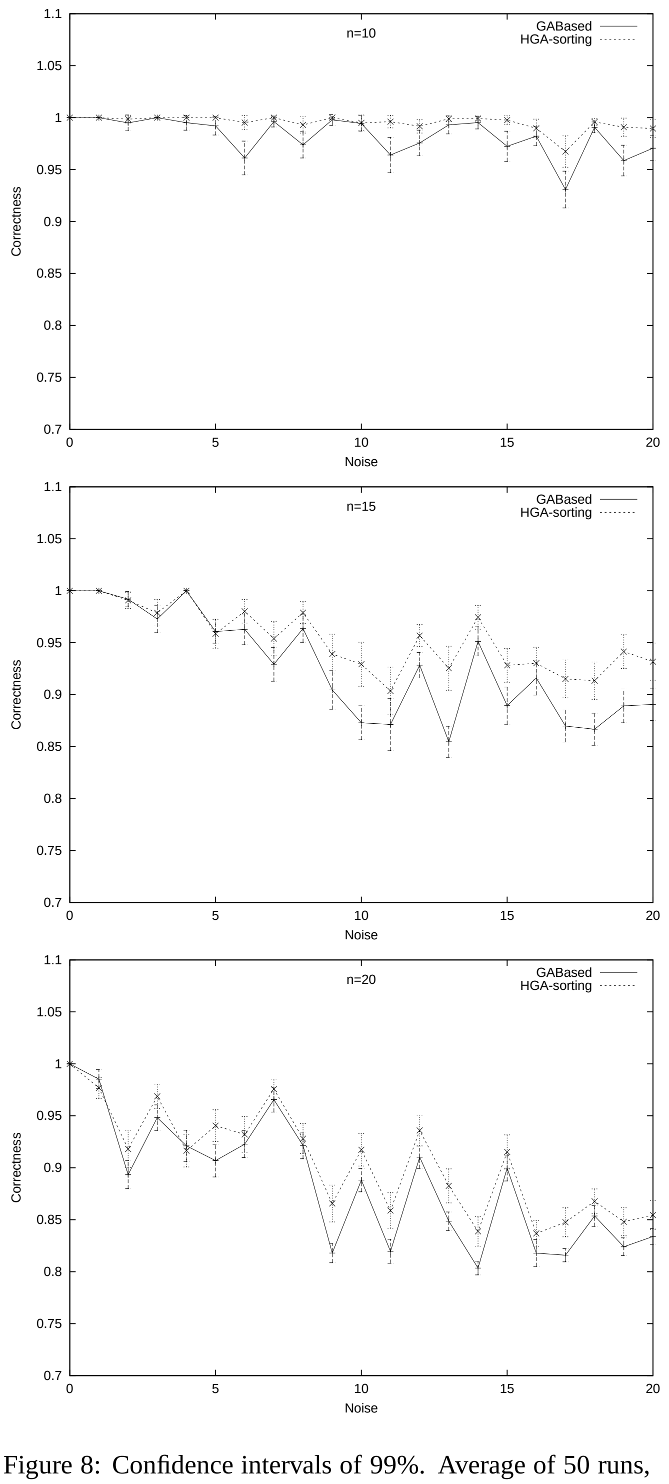

Figure 8: Confidence intervals of 99%. Average of 50 runs, using one problem instance (with elitism).

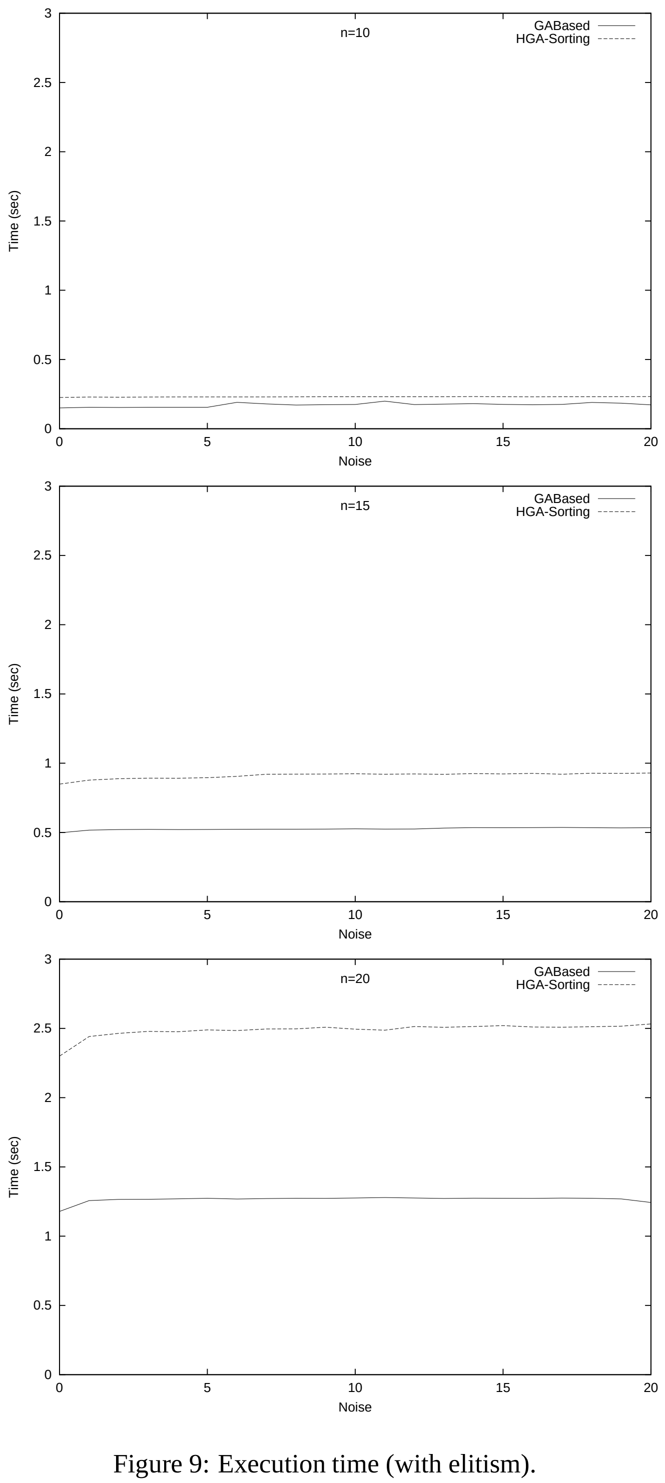

Figure 9: Execution time (with elitism).

Acknowledgment

The second author was supported by a fellowship from the National Council of Science and Technology (CONACYT, México), grant number 135385.

Bibliography

J. Culberson and J. Lichtner. On searching α-ary hypercubes and related graphs. In R. K. Belew and M. D. Vose, editors, *Foundations of Genetic Algorithms·4*, pages 263–290. Morgan Kaufmann Publishers, 1997.

L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 162–164, 1985.

L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

G. Escalada-Imaz and R. Torres-Velázquez. Algoritmos geneticos genericos y basados en orden: Conceptos fundamentale y mecanismo de base. *Inteligencia Artificial*, (3): 10–29, 1997. In spanish.

V. Estivill-Castro. Hybrid genetic algorithms are better for spatial clustering. In R. Mizoguchi and J. Slaney, editors, *PRICAI 2000 Topics in Artificial Intelligence. 6th Pacific Rim Internationa Conference on Artificial Intelligence*, pages 424–434. Springer Verlag LNAI 1886, 2000.

V. Estivill-Castro, H. Mannila, and D. Wood. Right invariant metrics and measures of presortedness. *Discrete Applied Mathematics*, 42:1–16, 1993.

V. Estivill-Castro and R. Torres-Velázquez. Hybrid genetic algorithm for solving the p-median problem. In X. Yao, R. I. McKay, C. S. Newton, and J. H. Kim, editors, *Proceedings of the Second Asia Pacific Conference on Simulated Evolutio and Learning (SEAL-98)*, pages 18–25. Springer Verlag LNAI 1585, 1999.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, 1989.

D. E. Goldberg and R. J. Lingle. Alleles, loci, and the traveling salesman problem. In J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Thei Applications*, pages 154–159. Lawrence Erlbaum Associates, 1985.

D. E. Goldberg and S. Voessner. Optimizing global-local search hybrids. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. G. an Vasant Honovar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO-99)*. Morgan Kaufmann Publishers, 1999.

J. Grefenstette, R. Gopal, B. Rosmaita, and D. V. Gucht. Genetic algorithms for the traveling salesman problem. In J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Thei Applications*, pages 160–168. Lawrence Erlbaum Associates, 1985.

D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, 1973.

J. Kobler, U. Schoning, and J. Toran. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhauser, 1993.

G. E. Liepins and M. R. Hilliard. Greedy genetics. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithm and Their Applications*, pages 90–99. Lawrence Erlbaum Associates, 1987.

B. T. Messmer and H. Bunke. A desicion tree approach to graph and subgrah isomorphism detection. *Pattern Recognition*, pages 1979–1998, 1999.

M. Mitchell. *An Introduction to Genetic Algorithms*. A Bradford book, The MIT Press, 1996.

R. C. Read and D. G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.

L. G. Shapiro and R. M. Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(5):504–519, September 1981.

W. M. Spears. Crossover or mutation. In L. D. Whitley, editor, *Foundations of Genetic Algoritms·2*, pages 221–237, San Mateo, California, 1993. Morgan Kaufmann Publishers.

T. Starkweather, S. McDaniel, K. Mathias, and D. Whitley. A comparison of genetic sequencing operators. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 69–76, San Mateo California, 1991. Morgan Kaufmann Publishers.

W.-H. Tsai and K.-S. Fu. Error-correcting isomorphisms of attributed relational graphs for patter analysis. *IEEE Transactions on Systems, Man and Cybernetics*, 9(12):757–768, December 1979.

N. L. J. Ulder, E. H. L. Aarts, H.-J. Bandelt, P. J. M. van Laarhoven, and E. Pesch. Genetic local search algorithms for the travelling salesman problem. In *First International Conference on Parallel Problem Solving from Nature*, pages 109–116, 1994.

Y.-K. Wang, K.-C. Fan, and J.-T. Horng. Genetic-based search for error-correcting graph isomorphism. *IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics*, 27(4):588–597, August 1997.

D. B. West. *Introduction to Graph Theory*. Prentice Hall, 1996.

D. Whitley. Modeling hybrid genetic algorithms. In G. Winter and P. Cuesta, editors, *Genetic Algorithms in Engineering and Computer Science*, pages 191–201. John Wiley, 1995.

T. Yamada and C. R. Reeves. Permutation flowshop scheduling by genetic local search. In *Genetic Algorithms in Engineering Systems: Innovations and Applications*, pages 232–238, 1997.