A Memetic Algorithm Instantiated with *Selection Sort* Consistently Finds Global Optima for the Error-Correcting Graph Isomorphism

Rodolfo Torres-Velázquez and Vladimir Estivill-Castro

School of Electrical Engineering and Computer Science,
University of Newcastle, University Drive, Callaghan, NSW 2308 Australia

Abstract - We study several tested cases of the Error-Correcting Graph Isomorphism Problem. The set S_n of $n!$ permutations on n items is the search space for this optimization problem. We apply *MA-sorting*. This is a Memetic Algorithm with domain-independent mutation operators based on classical sorting. Each sorting algorithm works on results to a comparison predicate and defines a path in S_n . In *MA-sorting* the mutation operator is a local-search that evaluates permutations suggested by the sorting algorithm. When such evaluation results in an improvement, the mutation is accepted and the comparison operator of the sorting algorithm evaluates to true. In contrast with previous proposals, our *MA-sorting* instantiated with *Selection Sort* finds optimal solutions for this case study.

I. Introduction

Memetic Algorithms (MAs) combine Genetic Algorithms (GAs) and Local Search (LS). Empirical evidence shows [8, 14, 4, 15, 11, 12, 19, 16, 5, 6] that MAs successfully attack hard combinatorial optimization problems. Previously [5, 6, 18], we proposed to integrate classical sorting in GAs towards improved permutation search, in particular we applied *Insertion Sort* and Quicksort. The experiment here extends our previous work in the following aspects. We now instantiate *MA-sorting* with *Selection Sort* to demonstrate convincingly that our approach is general and flexible. Second, we implement ideas suggested earlier [5] to produce three variants of mutation. We identify commonalities with previously proposed operators [8, 14, 4, 15, 11, 12, 19, 16]. We show that they can be considered as particular cases of our more general approach.

We obtain improved results for a benchmark of experiments of the *Error-Correcting Graph Isomorphism*. We implemented our proposal to solve this problem and the results of our experiments show that our *MA-sorting* instantiated with *Selection Sort* finds global optima for the tested samples of the case study.

II. The Error-Correcting Graph Isomorphism Problem

A benchmark problem arises 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. Remarkably, in the pattern-matching arena, modeling with graphs has been fruitfully used to match objects [20, 21, 13]. Thus, the interest in finding efficient algorithms to deal with the Graph Isomorphism (GI) problem, although its precise computational complexity remains unknown [13]. It requires to find a bijection 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. However, 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) [20]. To define this problem we first need the notion of attributed graph [20].

Definition 1 (AG) An *attributed graph* is a 4-tuple $G_a = (V, E, \alpha, \beta)$ where $V \neq \emptyset$ is a finite set of vertices; $E \subset V \times V$, is a set of distinct ordered pairs (edges) of distinct elements in V ; $\alpha : V \rightarrow \mathbb{R}$, is a function called vertex interpreter; and $\beta : E \rightarrow \mathbb{R}$, is a function called edge interpreter.

Definition 2 (ECGI) Given two AGs $G_a = (V(G_a), E(G_a), \alpha_G, \beta_G)$ and $H_a = (V(H_a), E(H_a), \alpha_H, \beta_H)$, with $|V(G)| = |V(H)|$, the *Error-Correcting Graph Isomorphism* problem is to find a permutation $\pi : V(G_a) \rightarrow V(H_a)$ so that some metric of total dissimilarity between the graph $G'_a = (\pi[V(G_a)], \pi[E(G_a)], \alpha_{G'}, \beta_{G'})$ and the graph H_a is minimized.

In order to explain our algorithm we introduce first our local-search model.

III. Local Search

LS has been also shown successful to deal with hard combinatorial optimization problems [1]. In such problems the search space is the set S of feasible solutions and the purpose is to find a solution $i^* \in S$ such that given a cost function $f : S \rightarrow \mathbb{R}$, $f(i^*)$ is globally optimal. A LS algorithm iteratively searches for a better solutions in the local neighborhood of a current solution i . The neighborhood $N \subseteq S$ of a LS algorithm is the set of solutions that can be reached from the current solution i by the application of a neighborhood operator $\nu : S \rightarrow 2^S$. LS mechanisms are approximation algorithms with the common feature of an underlying neighborhood operator ν is used to guide the search.

LS comprises four basic steps: *Initialization*, *Nomination*, *Comparison* and *Termination*. An outline of our LS model is displayed in Algorithm 1.

Algorithm 1 Local Search

```

1: Initialization: Select an starting solution  $i \in S$ ;
    $current\_best \leftarrow i$ ;
2: repeat
3:   repeat
4:     Nomination: Select a solution  $j \in \nu(i)$ 
5:     Comparison: if  $f(j)$  is better than
                    $f(current\_best)$   $current\_best \leftarrow j$ 
6:   until Termination condition is satisfied
7:    $i \leftarrow current\_best$ 
8: until no better neighbors

```

The starting solution selected during the *Initialization* step may impact on the quality of the solution obtained. For that reason, it is common to execute LS several times using different starting solutions. When the starting solution is randomly generated the approach is called *multi-start local search*. If the starting solution is the result of the perturbation of a previously obtained local optimum, the approach is called *iterated local search*. The latter has been recognized as the most effective approximate approach to the Traveling Salesman Problem [9].

In some cases the criterion used in the *Comparison* step is more elaborated. For example, simulated annealing uses randomized thresholds as the comparison criterion. This allows for a solution that is not the current best to be adopted as the current solution to continue the search.

During the *Nomination* step, the neighborhood operator ν may be more complex, particularly if we are performing a systematic nomination of neighbors. In this case, we maintain either a record of some of the previous nominations, as for example taboo search does, or the

state of the last nomination, if the itinerary of nominations is deterministic (nomination becomes an enumeration). Therefore, ν determines a neighborhood structure of the search space S and may establish an order for nominating solutions. Finding efficient and effective neighborhood operators which lead to high-quality solutions is one of the challenges of LS [1].

The *Termination* condition may be, for example: stop as soon as a better neighbor solution is found (*first improvement*); stop after all the neighbors of the current solution i has been evaluated (*best improvement*); or stop after performing a previously defined number of nominations. Observe that the order of nominations prescribed by ν is relevant, for example, when *first improvement* have been selected as the *Termination* condition.

IV. Selection Sort as a Local Search Mechanism that Directs Mutation

A well understood mechanism to deal with a class of permutation problems is the family of sorting algorithms. The sorting problem deals with sequences of items from a total order, and usually consists of re-arranging any given sequence so that the items are in ascending or descending order. Although it is unusual, classical sorting algorithms can be conceptualized as LS mechanisms, where the local information provided by each pair of elements in the array is enough to decide the search direction in a deterministic way. In fact, the algorithm is searching for the permutation that re-arranges the sequence into ascending (or descending) order. Using measures of presortedness it is possible to formalize sorting as an optimization problem. For example, let Exc be the minimum number of exchanges (arbitrary swaps) required to sort the sequence. The input sequence $X = \langle x_1, \dots, x_n \rangle$ for a sorting problem defines a permutation $\pi(X)$ on $\{1, \dots, |X|\}$ as $\pi(X)[i]$ equals the final position of x_i when X is sorted. More than 150 years ago, Cayley realized that $Exc(X) = |X| - \text{the number of cycles in } \pi(X)$. Since the identity permutation has $|X|$ cycles, in this case Exc is zero, the minimum (the sequence was already sorted). Sorting is now formalized as follows: Minimize $Exc(X)$. We model *Selection Sort* as a LS mechanism.

One of the simplest sorting algorithms is *Selection Sort*. It works as follows: first find the smallest element in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position. Continue in this way until the entire array of elements is sorted [17, p 96]. *Selection Sort* moves from left to right, placing elements in their final position without looking back. The computation cost of *Selection Sort* is $\Theta(n^2)$.

Lets describe more formally how *Selection Sort* remem-

bers its state of execution in two pointers $(k, l) \in K \times L$ where $K = \{1, \dots, k, \dots, n\}$, $L = \{k + 1, \dots, l, \dots, n\}$ (note $k < l$). A state (k, l) means that

- the sequence X is already sorted in its initial segment $\langle x_1, \dots, x_{k-1} \rangle$,
- the initial segment $\langle x_1, \dots, x_{k-1} \rangle$ has the $(k - 1)$ smallest elements in the sequence,
- we are attempting to place the k -th smallest element in the k -th position, and
- the element in the k -th position is the smallest amongst $\langle x_k, \dots, x_{l-1} \rangle$.

Computation from such a state proceeds as follows. We compare the elements in the k -th and l -th position. If the element in the l -th position is smaller, it is swapped with the one in the k -th position. Then, the next state is $(k, l + 1)$ (unless $l = n$, in which case we have extended the initial segment and the new state is $(k + 1, k + 2)$).

Note that a very similar encoding of the state occurs for *Insertion Sort* [5], but the state only means that the initial segment is sorted (not that it contains the $(k - 1)$ smallest elements of the sequence). Also, the path of states increments differently in *Insertion Sort* than in *Selection Sort*.

We illustrate now the execution of *Selection Sort*, in the context of Algorithm 1. This sorting mechanism induces the function $\nu : S \times K \times L \rightarrow 2^S$ where $K = \{1, \dots, k, \dots, n\}$, $L = \{1, \dots, l, \dots, n\}$ and $k \neq l$. Assume the *Initialization* step produces the permutation $\pi_i = \langle 1, 4, 5, 2, 3 \rangle$, and the current state of the *Nomination* step is represented by $k = 2$ and $l = 4$ (pointers to elements of π). The *Nomination* step produces $\pi_j = \nu(\pi_i, k, l) = \langle 1, 2, 5, 4, 3 \rangle$. The comparison predicate $<(\pi_i, \pi_j)$ guides the *Comparison* step. If the *Termination* step was defined as *first improvement* and the result of the *Comparison* step is *True*, the LS mechanism ends with π_j . Otherwise, the *Nomination* step produces $\pi_j = \nu(\pi_j, 2, 5) = \langle 1, 3, 5, 2, 4 \rangle$ and the *Comparison* step tests $<(\pi_i, \pi_j)$ continuing the cycle.

In our integration of the MA algorithm, a mutation is the result of a swap suggested by *Selection Sort*. The acceptance or not of the swap is performed by evaluating the objective function in the entire permutation. It is important to notice that we do not compare the individual elements that are being swapped, but the fitness of the entire permutations as a whole, before and after the swapping. However, if the resulting permutation represents an improvement (with respect to the objective function), the message relayed to the concurrent *Selection Sort* is that the comparison returned *True*, as if the two elements were swapped (from the perspective of *Selection Sort* we compared the individuals). Symmetri-

cally, if the resulting permutation does not correspond to an improvement with respect to the objective function, the message relayed back to *Selection Sort* is that the comparison resulted in *False* (as if the individual items were not swapped).

The swap mutation operator used by Wang et al. [21] chooses a uniform randomly pair of positions in the chromosome and swap the contents of the selected positions. This operation often produces a chromosome with worse performance. This is not considered a problem when a 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 is a hill-climber. Other mutation operators which perform LS has been proposed in literature. Many of them [8, 4, 19, 16] are based on variants of the 2-opt algorithm proposed by Croes [3].

In our approach [5, 6], classical sorting 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 , *Selection Sort* defines which mutation operation to perform. We refer to our mutation operator as *SORT*.

The only explicit knowledge in our *MA-sorting* algorithm is that the problem involves permutations.

The impact of the sorting algorithm into the mutation, and thus into *MA-sorting* is to be regulated in two dimensions. The first dimension is the amount of progress in the sorting process at each invocation of mutation; this corresponds to the *Termination* step in our model of LS. The second dimension is the proportion of sorting algorithm states to chromosome in the population (recall the *Nomination* step may maintain a record of previous nominations). More details about these two dimensions can be found in our previous work [5].

Our *SORT* mutation operator determines facts like item i should be before item j , replicating how sorting algorithms learn the permutation to sort the input.

V. Experimental Setting

We reproduced the experimental setting of Wang et al. [21]. Their construction of graphs for the ECGI problem constitutes a benchmark where to test the effectiveness of GAs. A graph G_a with n nodes is randomly generated as well as a permutation π^* is uniformly constructed (each permutation has probability $1/n!$). The graph H'_a is the result of the application of π^* and noise to G_a (see [5, 18]). The parameter $\varepsilon \in [0, 20]$ is used in the experiment to regulate the approximate mismatch in the pair of graphs.

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. In Definition 2 we left open the precise definition of a criterion of dissimilarity between attributed graphs. This is based on their edge-weight matrix (a type of adjacency matrix).

Definition 3: Let $n = |G_a(V)|$ be the number of vertices of G_a and $M(G_a) = [m_{i,j}]$ be the $n \times n$ edge-weight matrix of the attributed undirected graph G_a given by $m_{i,i} = \alpha(v_i)$ and $m_{i,j} = \beta(v_i, v_j)$.

Note that two graphs are isomorphic only if their edge-weight adjacency matrices differ by a permutation 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). 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(π)) 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 finding a permutation π that maximizes $Mx_ATE(\pi)$ given by $Mx_ATE(\pi) = C'_{max} - ATE(\pi)$, where C'_{max} is the maximum possible value of $ATE(\pi)$.

We implemented *MA-sorting* instantiated with *Selection Sort* 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.

TABLE I
MA-sorting PARAMETERS.

Parameter	MA-sorting
Encoding Scheme	Integer Permutation
Population Size	30
Selection	Inhibitive
Termination	Upon Thresh Convergence
Scaling	Ranking Scaling
Crossover	PMX
Mutation	SORT
Crossover Probability	0.9
Mutation Probability	.1
Elitism	True

The operators and parameters of the GAs used in our

experiments are summarized in Table I. Operators like PMX [7] are well known in the GA community when applied to optimization problems searching for permutations. We will skip PMX details here, but the use of this conventional crossover operator (as suggested by Grefenstette [8]) avoids the danger of premature convergence [10]. Similarly, we skip description of the Inhibitive Selection [21], Terminate Upon Thresh Convergence [21] and details of Ranking Scaling [2]. These can be found in the references.

In our experiments, the mutation operator is the result of using *Selection Sort* each time the mutation operator is requested. We implemented three different modes of *SORT* based on *Selection Sort*. The first mode (SSF) uses a single global encoding (k, l) independent of the size of the population. So, the *Selection Sort* may resume on a different chromosome because it resumes when a mutation is requested. This mode assumes a collective memory of the chromosome for facts like *i is before j*. It can be considered a single thread for *Selection Sort*. This mode halts the *Selection Sort* with *first improvement* as the *Termination* criterion. The second mode (SRF) randomly generates the state of the sorting algorithm and uses *first improvement* as the *Termination* criterion. We have found that SRF coincides with the operator proposed by some authors [14, 15, 11, 12] as a variant of the 2-opt operator [3]. The third mode (SRB) is similar to the second mode but uses *best improvement* as the *Termination* criterion.

VI. 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 $Mx_ATE(\pi_{found})$ and the optimum value $Mx_ATE(\pi^*)$ (recall permutation π^* was used to construct H_a from G_a). This ratio is called *correctness* and is given by

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

Applying the methodology described in Section V, we constructed 50 pairs of graphs for each integer value of $\varepsilon \in [0, 20]$, and each size $n \in \{10, 15, 20\}$. For each graph pair we executed three times the GA and selected the best solution found. So, we generated 3150 test cases, and executed 9450 times each Genetic Algorithm.

Plots of the results are displayed in Figure 1. 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 ε values. The y -axis is a correctness scale, from 70% correct to 100% correct. The plotted

lines are average *correctness*, taken over the 50 problems. The dotted lines correspond to results for our *MA-sorting* instantiated with *Selection Sort* (labeled in the figure as SSF, SRF and SRB), and to previous results [5] for our *MA-sorting* instantiated with *Insertion Sort* (labeled HGA). The solid lines correspond to Wang et al. algorithm (labeled GAB). This figure illustrates that the three variants of our *MA-sorting* with *Selection Sort* performs much better than the rest when the size of the problem is 15 and 20. Moreover, both SSF and SRB are superior to SRF for all the tested cases. Finally, SRB is able to find the global optimum for all the tested cases.

Fig. 1. Correctness for *MA-sorting* instantiated with *Selection Sort*. Plots for 3 sizes. Average for 50 graph pairs.

Table II shows average CPU time, standard deviation

and maximum overhead, for different size graphs when *MA-sorting* is instantiated with *Selection Sort* (SRB) in comparison with GAB. It is clear that hybridization adds overhead, but the table shows that it is constant for the range of ϵ values, and that it also remains acceptable in the dimension of n values. Thus, the *MA-sorting* instantiated with *Selection Sort* is effective, obtaining optimal solutions for reasonable requirements of CPU time.

TABLE II
CPU TIME REQUIREMENTS. OVERHEAD FOR DIFFERENT SIZE GRAPHS USING SRB.

Graph Size	μ (sec)	σ (sec)	Maximum (sec)
10	0.0656	0.0136	0.0862
15	0.9802	0.0191	0.9936
20	4.6671	0.0681	4.7486

VII. Discussion

The first remarkable point derived from this work is the outstanding performance of our *MA-sorting* instantiated with *Selection Sort*. When SRB guides the mutation operator we find optimal solutions for the tested cases of the Error-Correcting Graph Isomorphism.

We conjecture that the main reason for this optimal performance is that SRB, in contrast with HGA, is not restricted to only exploiting the swapping of adjacent items. This makes us suspect that this test-bed may be easy, and not representative of the computational complexity of the ECGI problem. However, if other algorithms do poorly on this test-bed, they are less likely to be effective on harder instances.

Although for the case study the neighborhood operator ν suggested by SRB is clearly superior to that suggested by HGA, further research is needed to contrast the results here with Quicksort [18]. This research must develop theory that can explain why randomly re-starting the state of *Selection Sort* results in better performance.

The goal of our research was to develop optimization methods for a general class of problems characterized as follows. It should be clear that a permutation similar in sorted order with respect to the optimal solution has also a good value for the objective function. Our hybridization with sorting algorithms was inspired on minimizing domain knowledge but accepting that the optimization problem would offer some regularity of the landscape with respect to a measure of presortedness. Thus, it makes sense for the optimizer to learn heuristically; that is, record information “ i before j ” suggested by the sorting algorithm. For the case study of the ECGI problem, a permutation that is a close order to the optimal

permutation is a reasonable match between the two given graphs to reduce the error in the match. We conjecture that the appropriate combination of a measure of presortedness with a suitable operator ν induces good quality solutions. Although we can not say which of the many measures of presortedness and operators ν best suit the ECGI problem. Again, this indicates that more detailed theoretical research is needed to explain the contrast with other instantiations, like using Quicksort [18].

Acknowledgment

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

References

- [1] E. Aarts and J. K. Lenstra. Introduction. E. Aarts et al eds., *Local Search in Combinatorial Optimization*, 1–17. Wiley, 1997.
- [2] J. E. Baker. Adaptive selection methods for genetic algorithms. J. J. Grefenstette, ed., *Int. Conf. on Genetic Algorithms*, 101–111, Carnegie Mellon U., 1985. Lawrence Erlbaum.
- [3] G. A. Croes. A method for solving traveling-salesman problems. *Op. Research*, 5:791–812, 1958.
- [4] L. Davis, ed. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
- [5] V. Estivill-Castro and R. Torres-Velázquez. Classical sorting embedded in genetic algorithms for improved permutation search. *CEC2001*, 941–948, Seoul, Korea, 2001. IEEE Press.
- [6] V. Estivill-Castro and R. Torres-Velázquez. How should feasibility be handled by genetic algorithms on constraint combinatorial optimization problems? the case of the *valued n-queens problem*. *2nd Workshop on Memetic Algorithms. WOMA II. GECCO-2001*, 146–151, 2001.
- [7] D. E. Goldberg and R. J. Lingle. Alleles, loci, and the traveling salesman problem. J. J. Grefenstette, ed., *Int. Conf. on Genetic Algorithms*, 154–159, Carnegie Mellon U., 1985. Lawrence Erlbaum.
- [8] J. J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. L. Davis, ed., *Genetic Algorithms and Simulated Annealing*, 42–60. Pitman, 1987.
- [9] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: a case study. E. Aarts et al eds., *Local Search in Combinatorial Optimization*, 215–310. Wiley, 1997.
- [10] G. E. Liepins and M. R. Hilliard. Greedy genetics. J. J. Grefenstette, ed., *2nd Int. Conf. on Genetic Algorithms*, 90–99. Lawrence Erlbaum, 1987.
- [11] P. Merz and B. Freisleben. A genetic local search approach to the quadratic assignment problem. T. Bäck, ed., *7th Int. Conf. on Genetic Algorithms*, 465–472, Michigan State U., East Lansing, 1997. Morgan Kaufmann.
- [12] P. Merz and B. Freisleben. Fitness landscape analysis and memetic algorithms for the quadratic assignment problem. *IEEE T. Evolutionary Computation*, 4(4):337–352, 2000.
- [13] B. T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 1979–1998, 1999.
- [14] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. Schaffer, ed., *3rd Int. Conf. on Genetic Algorithms*, 416–421, George Mason U., 1989. Morgan Kaufmann.
- [15] H. Mühlenbein. Evolution in time and space - the parallel genetic algorithm. G. J. E. Rawlins, ed., *Foundations of Genetic Algorithms*, 316–337, Indiana University, 1991. Morgan Kaufmann.
- [16] M. Rocha, R. Mendes, P. Cortez, and J. Neves. Sitting guests at a wedding party: Experiments on genetic and evolutionary constrained optimization. *Congress on Evolutionary Computation CEC2001*, 671–678, Seoul, Korea, 2001. IEEE Press.
- [17] R. Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing Company, 1992.
- [18] R. Torres-Velázquez and V. Estivill-Castro. A Memetic algorithm instantiated with Quicksort for the error-correcting graph isomorphism problem. *2nd European Workshop on Evolutionary Computation in Combinatorial Optimization*, in press, Kinsale, Ireland, 2002. Springer-Verlag Lecture Notes in Computer Science.
- [19] H.-K. Tsai, J.-M. Yang, and C.-Y. Kao. A genetic algorithm for traveling salesman problems. L. Spector, et al eds., *GECCO-2001. Genetic and Evolutionary Conference*, 687–693, San Francisco, CA, 2001. Morgan Kaufmann.
- [20] W.-H. Tsai and K.-S. Fu. Error-correcting isomorphisms of attributed relational graphs for pattern analysis. *IEEE T. Systems, Man and Cybernetics*, 9(12):757–768, 1979.
- [21] Y.-K. Wang, K.-C. Fan, and J.-T. Horng. Genetic-based search for error-correcting graph isomorphism. *IEEE T. Systems, Man and Cybernetics, Part B: Cybernetics*, 27(4):588–597, 1997.