A Comparison of Evolutionary Algorithms for System-Level Diagnosis

Bogdan Tomoyuki Nassu
Department of Informatics
Federal University of Paraná
Curitiba — PR — Brazil
bogdan@inf.ufpr.br

Elias Procópio Duarte Jr.
Department of Informatics
Federal University of Paraná
Curitiba — PR — Brazil
elias@inf.ufpr.br

Aurora T. Ramirez Pozo
Department of Informatics
Federal University of Paraná
Curitiba — PR — Brazil
aurora@inf.ufpr.br

ABSTRACT

The size and complexity of systems based on multiple processing units demand techniques for the automatic diagnosis of their state. System-level diagnosis consists in determining which units of a system are faulty and which are fault-free. Elhadef and Ayeb have proposed a specialized genetic algorithm (GA) that can be used to accomplish diagnosis. This work extends their approach, describing and comparing several evolutionary algorithms for system-level diagnosis. Implemented algorithms include a simple genetic algorithm, a specialized GA both with and without crossover and specialized versions of the compact GA and Population-Based Incremental Learning both with and without negative examples. These algorithms had their performance evaluated using four metrics: the average number of generations needed to find the solution, the average fitness after up to 500 generations, the percentage of tests that found the optimal solution and the average time until the solution was found. An analysis of experimental results shows that more sophisticated algorithms converge faster to the optimal solution.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: *Fault Tolerance*

General Terms

Algorithms, Reliability, Experimentation

Keywords

System-Level Diagnosis, Dependability, Evolutionary Algorithms

1. INTRODUCTION

Computer systems that rely on multiple processors to achieve their goals are increasingly popular. Examples include both local and wide-area computer networks, clusters, parallel architectures, and multi-processor boards. It is well-known that given a large enough time interval, processors will fail. A large system consisting of several processors must employ fault-tolerance techniques in order to be able to deliver its expected service even when some processors become faulty. As the size and complexity of such systems increase, techniques for the automatic detection of faults become necessary. System-level diagnosis consists in determining which units in a system are faulty and which are fault-free [10]. Based on this information, actions such as the isolation or replacement of faulty units can be taken.

The classical model for system-level diagnosis is the PMC model, first presented by Preparata, Metze and Chien in [9]. In this model, a system is defined as a collection of n heterogeneous units, represented by a set $U = \{u_1, \ldots, u_n\}$. Each unit in the system can be in one of two states: faulty or fault-free. Given a fault situation, the subset $F \subset U$ of permanently faulty units is called the fault set.

The diagnosis is solved based on the results of tests performed by the units in the system. It is assumed that a fault-free unit can determine and report the state of tested units correctly. However, the result of a test performed by a faulty unit is undefined, and a faulty unit can "lie" about tests executed on other units, i.e. a faulty unit may be reported as being fault-free or vice versa. The diagnosis is solved by a central unit to which all testers send the results of their tests. This central unit does not belong to U and does not perform any test. It is assumed that it never fails. It is also assumed that the state of the units does not change during diagnosis.

A testing graph is employed to represent the tests executed by the units. It is defined as a directed graph $G(U, E)$ in which each vertex represents a unit $u_i \in U$, and each edge (u_i, u_j) is in E if and only if u_i tests u_j. Two sets are associated to each unit $u_i \in U$:

$$\Gamma(u_i) = \{u_j | (u_i, u_j) \in E\}$$

$$\Gamma^{-1}(u_i) = \{u_j | (u_j, u_i) \in E\}$$

The first set contains the units u_j tested by u_i, and the second contains the units u_j that test u_i. These sets have two values associated with them: $d_{out}(u_i) = |\Gamma(u_i)|$ and $d_{in}(u_i) = |\Gamma^{-1}(u_i)|$. These values represent, respectively, the number of units tested by u_i and the number of units that test u_i.

Each edge $(u_i, u_j) \in E$ has a test result $S(u_i, u_j)$ asso-

GECCO'05, June 25–29, 2005, Washington, DC, USA.

ciated to it. If u_i tests u_j as faulty, the value of $S(u_i, u_j)$ is 1, otherwise its value is 0. Given the PMC model's assumptions, $S(u_i, u_j)$ is a reliable result if and only if u_i is fault-free. The set containing all the results from all the tests is called the system's syndrome, denoted by S^*. Therefore, the diagnosis is solved by the central unit based on S^*.

A given syndrome S is said to be compatible with a fault set F if, for every edge $(u_i, u_j) \in E$ such as u_i is fault-free, $S(u_i, u_j) = 1$ if and only if u_j is faulty. The system's syndrome, S^*, is always compatible with the system's fault set. The subsets $S(u_i)$ and $S^{-1}(u_i)$ of S are also defined. These are, respectively, the sets containing the results of the tests performed by u_i and the tests performed on u_i.

A system is said to be t-diagnosable if the number of faulty units is not greater than t, and the central unit can identify them correctly. It is proven [4] that, if no two units test each other, a system is t-diagnosable if $n \geq 2t + 1$ and, for every unit $u_i \in U$, $d_{in}(u_i) \geq t$, that is, if every unit is tested by at least t others. The diagnosability t of a given system must be determined in order to diagnose that system; otherwise the result obtained may be incorrect.

System-level diagnosis is a well known problem, for which several approaches have been proposed [10]. Practical applications recently reported include network management [7]. Elhadef and Ayeb have proposed a specialized genetic algorithm (GA) for diagnosis [3]. In this work, their approach is extended, with the application of several evolutionary algorithms to diagnosis being described and compared. The first one is a simple genetic algorithm, developed for comparison purposes only. The second solution is a specialized genetic algorithm, very similar to the one proposed in [3], with a slightly different method being used to generate the initial population; versions both with and without crossover were implemented. The other algorithms are specialized implementations of estimation of distribution algorithms (EDAs). The algorithms are the Population-Based Incremental Learning — PBIL [1, 2] — and the compact GA [6]. Both were optimized specifically for system-level diagnosis, and PBIL was implemented with and without negative examples.

Experimental results allowed the comparison of these approaches using four metrics: the average number of generations needed to find the solution, the average fitness after up to 500 generations, the percentage of tests that found the optimal solution and the average time until the solution was found. An analysis of experimental results shows that more sophisticated algorithms converge faster to the optimal solution.

The rest of this paper is organized as follows. Section 2 presents the specialized GA proposed by Elhadef and Ayeb in [3]. In section 3, the evolutionary algorithms we propose are described: the simple GA and the specialized implementations of PBIL and the compact GA. The changes made to the specialized GA are also highlighted. Section 4 describes and discusses experimental results. Section 5 concludes the paper.

2. A SPECIALIZED GA BY ELHADEF AND AYEB

This section presents the specialized GA proposed by Elhadef and Ayeb in [3]. Its purpose is to discover, based on a given syndrome S^*, the system's fault situation.

In this specialized GA, a chromosome represents a possible fault state of the system. Each chromosome v is represented as a bit string, with each gene corresponding to the state of a unit in the system — faulty or fault-free. If the gene's value is one, the corresponding unit is faulty, otherwise it is fault-free. Therefore, a system with n units is represented by a chromosome of length n. For example, in a system with $n = 8$, the fault set $F = \{u_1, u_4, u_8\}$ is coded as (10010001). As system-level diagnosis assumes a t-diagnosable system in a fault situation, a chromosome that has no faulty units or has more than t faulty units may be deemed illegal. For example, if $t = 4$ and $n = 9$, the chromosomes (000000000) and (111001110) are not legal.

Given the proposed string representation, a good measure of a chromosome's fitness is how well it is represented by the tests performed by the units. This measure can be obtained by generating an S syndrome, compatible with the fault set represented by chromosome v. This S syndrome is then compared to the "real" system syndrome, S^*, that contains the results of all the tests performed by the units. The S syndrome must be generated so it is identical to S^* if and only if the fault set represented in v is the system's real fault set. In this case, v is the optimal solution.

Let $F(v)$ be the set containing the faulty units in chromosome v (the genes that have a value of 1); $v[i]$ be the i-th gene in v and $S(u_i, u_j)$ the result of test (u_i, u_j) in a S syndrome. The S syndrome can be generated by two rules. The first one states that for every $u_i \in F(v)$ and every $u_j \in \Gamma(u_i)$; $S(u_i, u_j) = S^*(u_i, u_j)$. In other words, if a unit is faulty in v, in S its tests have the same results found in the "real" system syndrome, S^*. The second rule says that for every $u_i \in U - F(v)$ and every $u_j \in \Gamma(u_i)$; $S(u_i, u_j) = v[j]$. This means that, if a unit u_i is not faulty in v, in S the test result $S(u_i, u_j)$ is 1 if u_j is faulty in v, or 0 otherwise. Thus, if $F(v)$ equals the "real" fault set F, S and S^* are identical and chromosome v represents the system's real state.

For example, suppose a system with $n = 3$ and $t = 1$ and in which u_1 tests u_2 as faulty, u_2 tests u_3 as non-faulty and u_3 tests u_1 as faulty. So, $S^*(u_1, u_2) = 1$, $S^*(u_2, u_3) = 0$ and $S^*(u_3, u_1) = 1$. Suppose the syndrome S, generated by chromosome $v = (001)$. As u_1 is not faulty in v, the value of $S(u_1, u_2)$ is the value of $v[2]$. Then, $S(u_1, u_2) = 0$. The value of $S(u_2, u_3)$ is also defined that way, and $S(u_2, u_3) = 1$. But, as u_3 is faulty in v, the test result $S(u_3, u_1)$ is retrieved from $S^*(u_3, u_1)$, and $S(u_3, u_1) = 1$. Now, suppose the chromosome $v = (100)$. In this case, the S syndrome has $S(u_1, u_2) = 1$, $S(u_2, u_3) = 0$ and $S(u_3, u_1) = 1$. So, S and S^* are identical, and $v = (100)$ is the optimal solution.

If the chromosome is illegal, its fitness is 0. Otherwise, the fitness is calculated based on the generated syndrome, S. The fitness function used in this work considers each gene has its own fitness. A chromosome's fitness is the normalized sum of its genes' fitness values.

The f function that calculates the fitness for $v[i]$, the i-th gene of v, is defined as:

$$f(v[i]) = \frac{f_{in}(v[i]) + f_{out}(v[i])}{2}, \text{ where}$$

$$f_{in}(v[i]) = \frac{|S^{-1}(u_i) \cap S^{*-1}(u_i)|}{d_{in}(u_i)}, \text{ and}$$

$$f_{out}(v[i]) = \begin{cases} 1 & \text{if } d_{out}(u_i) = 0, \\ \frac{|S(u_i) \cap S^*(u_i)|}{d_{out}(u_i)} & \text{otherwise} \end{cases}$$

Function $f_{in}(v[i])$ computes the normalized number of tests performed on unit u_i that have the same result in the generated syndrome, S, and in the system's syndrome, S^*. In a similar fashion, $f_{out}(v[i])$ computes the normalized number of tests performed by unit u_i with equal results in S and S^*. If u_i does not perform any tests, $f_{out}(v[i]) = 1$. Therefore, the i-th bit's fitness considers u_i both as a testing and as a tested unit. The value of $f(v[i])$ can be seen as the likeness of the state of u_i in v being correct.

The fitness FT for a chromosome v is then the normalized sum of v's genes fitness values, or:

$$FT(v) = \frac{\sum_{i=1}^{n} f(v[i])}{n}$$

Considering the example presented earlier, if $v = (001)$, $f(v[1]) = 0.5$, $f(v[2]) = 0$ and $f(v[3]) = 0.5$. Then, $FT(v) = 0.33$. Now, if $v = (100)$, all the genes will have a fitness of 1, and then $FT(v) = 1$.

If v is the optimal solution, that is, if v represents the real fault state in the system, then $FT(v) = 1$. So, the algorithm's end condition corresponds to achieving a chromosome with a fitness of 1.

In each generation of the evolution loop of the GA, given a population of size p, p chromosomes are selected to generate the new population. Roulette-wheel selection is employed. As the selection is always made over the whole current population, each chromosome may be selected more than once. Some of the selected chromosomes are then reproduced without alterations, and some are used in the crossover operation. The crossover rate does not change during the algorithm's execution. At last, each bit in the chromosome has a chance of being mutated.

The crossover operation used is a simple one-point crossover that generates two new chromosomes for each pair of parents. The generation of illegal chromosomes is solved in a very simple way: if the crossover generates an illegal chromosome, the operation is ignored and the parents are simply reproduced in the new population.

The specialized GA uses an adaptive strategy to define which bits in a chromosome are mutated. This strategy is based on the fitness value of each bit in the chromosome. The specialized GA uses this value to determine which bits are mutated: if the bit's fitness is less than the mutation rate, the bit is flipped. If there are no bits to mutate by this criterion, the one with the smallest fitness has a chance equal to the mutation rate of being mutated.

To avoid the generation of illegal chromosomes by the mutation operation, two strategies are used. Let v be a chromosome and i the bit that will be mutated. If $|F(v)| = 1$ and $v[i] = 1$, i is the only faulty unit in v, and if $v[i]$ is mutated, v will have no faulty units. If this happens, the bit corresponding to the tester of i with the smallest fitness is also flipped. If $|F(v)| = t$ and $v[i] = 0$, the mutation will generate an illegal chromosome with more than t faulty units. In this case, the bit corresponding to the faulty unit with the smallest fitness is flipped. Using this scheme, the generation of illegal chromosomes is avoided.

A straightforward method that can be used to generate the initial population is a uniform random method, in which every bit in every chromosome is defined randomly. The specialized GA can generate the initial population in a more efficient way, reducing the total number of generations needed so that the optimal solution can be found. For each chromosome, a unit is chosen randomly, and is defined as being fault-free. The state of other units can then be inferred from the system's syndrome, S^*. In this work, this method was further developed, as shown in section 3.2.

3. PROPOSED EVOLUTIONARY ALGORITHMS FOR DIAGNOSIS

This section describes several evolutionary algorithms for system-level diagnosis that were implemented and compared. The first one is a simple GA, designed only for comparison purposes. The second one is a specialized GA, very similar to the one presented in the previous section. Versions of the Population-Based Incremental Learning - PBIL [1, 2] - and the compact GA [6] are also presented. These algorithms were optimized for the diagnosis problem.

All algorithms will use the same string representation and fitness function proposed by Elhadef and Ayeb, presented in the previous section.

3.1 A Simple GA

A simple GA was developed, for comparison purposes only. It is a more general approach than the specialized GA. The initial population is generated using a uniform random method. The crossover operation is a simple one-point crossover, and the mutation rate is the same for all the genes in a chromosome. Illegal chromosomes may be generated in the initial population or by the crossover and mutation operations. These chromosomes will have a fitness of 0.

3.2 A Specialized GA

The implemented specialized GA is very similar to the one presented in the previous section. Two changes were made.

One of them is that the algorithm can be run with no crossover at all. This was done because the tests reported in [3], attained the best results when the smallest crossover rates were used. That way, the importance of crossover in this specialized GA must be evaluated.

The other change was in the method used to generate the initial population. This work considers a more detailed approach than the one proposed by Elhadef and Ayeb. The following algorithm is used to generate each chromosome v in the initial population. It must be noted that it avoids the generation of illegal chromosomes in the initial population.

1. A unit u_i is chosen randomly, and is defined as being fault-free in v.

2. Units tested by u_i have their state defined in v by the results of the tests performed by u_i. These results are taken from the system's syndrome S^*.

3. Units that test u_i in S^* as faulty are defined as being faulty themselves in v.

4. If the total number of faulty units in v is greater than t, or if all the units were defined as being fault-free, u_i cannot be fault-free. The chromosome is discarded and

u_i is not selected anymore in the first step to generate other chromosomes.

5. If the state was not defined for all the units and there is at least one faulty unit in v, the remaining units are assumed to be fault-free. Otherwise, one of the units with undefined state is chosen (randomly) and defined as faulty, while the remaining units are considered fault-free.

3.3 PBIL — Population-Based Incremental Learning

In a genetic algorithm, the population stores information about the points already visited by the algorithm in the search space. Operators such as selection and crossover can be seen as a way to use this information. Understanding the role of the population and genetic operators in a GA made the creation of another class of algorithms possible. These algorithms replace operations such as crossover and selection with another technique: the estimation of the distribution of genes. The evolutionary algorithm presented in this section, PBIL (Population-Based Incremental Learning) [1, 2], is based on this concept.

In PBIL, evolution is guided by a vector containing real values, the probability vector. Each position in this vector represents the probability of a gene in a chromosome taking a value equal to 1. These probabilities are initialized with 0.5, or 50%. This means each gene has an equal chance of taking a value of 0 or 1. This value is updated through the generations, so it represents individuals with high fitness. At each generation, a new population is created from the probability vector.

PBIL is characterized by three parameters. The first one is the size of the population generated at each iteration. The second one is the learning rate, which tells how much each bit of the probability vector is moved in the direction of the best solutions in each generation. The third one defines how many individuals are used to update the probability vector in each generation. The selection of these individuals can be also extended, so the algorithm can also learn from negative examples. Thus, besides being updated in the direction of the x best examples, the probability vector can be updated in the opposite direction of the y worst examples. The probability vector can also suffer mutations, which lead to small variations of its values.

When PBIL is used for system-level diagnosis, the probability vector can be generated in an optimized way, excluding right from start some illegal solutions. This can be done by the following algorithm, executed for each position i in the probability vector.

1. Unit u_i is assumed to be fault-free.

2. The state of the units tested by u_i is defined by the results of the tests made by u_i. These results are obtained from the system's syndrome, S^*.

3. Units that test u_i in S^* as being faulty are defined as faulty themselves.

4. If there were more than t units whose state was defined as faulty, or if all the units were defined as fault-free, unit u_i can not be fault-free - it is surely faulty. Then, the value in the i-th position in the probability vector is defined as being 1. Otherwise the value is set to 0.5.

To avoid the generation of illegal solutions from a probability vector, the following strategy is used: if the chromosome does not have any faulty units, the unit with the smallest fitness is defined as being faulty. If there are more than t faulty units, the ones with the smallest fitness are considered fault-free, until the number of faulty units is no greater than t.

3.4 Compact GA

The compact genetic algorithm, or cGA [6], is another algorithm that, like PBIL, replaces the genetic operators of selection and crossover with the probabilistic distribution of genes. This algorithm reduces significantly memory requirements because, unlike the other algorithms considered until now, it does not need to maintain a population of solutions.

Like PBIL, the cGA uses a probability vector with real values that starts with all positions having the value 0.5, or 50%. This vector is used to probabilistically generate 2 solutions. These solutions are then compared, and the probability vector is updated in the direction of the "winner" — the one with the highest fitness. Each bit is updated independently: if the winner has a bit in 1 and the looser in 0, the corresponding bit is incremented in the probability vector. In the same way, if the winner has a bit in 0 and the looser in 1, the bit is decremented in the probability vector. If a bit has the same value in both the winner and the looser, its value remains unchanged in the probability vector. The values in the probability vector are incremented or decremented by an amount defined by a learning rate parameter. There are studies that show that a cGA with a learning rate of $1/n$ behaves like a simple GA with a population of size n [6, 5].

The cGA can be used to perform system-level diagnosis in a way that is similar to PBIL. The same procedures for initializing the probability vector and to avoid the generation of illegal solutions can be used without changes.

4. EXPERIMENTS

The algorithms presented in section 3 were implemented to solve system-level diagnosis so that their performance could be compared. Some variations on the basic algorithms were evaluated, resulting in a total of six different configurations: simple GA, specialized GA with and without crossover, PBIL with and without learning from negative examples, and compact GA.

4.1 Description

The experiments were executed on testing graphs for t-diagnosable systems with size n, using different values of t and n. These testing graphs were generated with each unit $u_i \in U$ testing units $u_{(i+1) \bmod n}$ to $u_{(i+t) \bmod n}$. As $n \geq 2t + 1$, each unit will be tested by t other units and no two units will test each other. Figure 1 shows a testing graph with $n = 8$ and $t = 3$.

For each test, a fault set was defined by randomly selecting n_f faulty units, so that $n_f \leq t$. From the testing graph and the fault set, the system's syndrome S^* was generated, with the results of the tests performed by the faulty units being defined randomly.

To improve the algorithms' performance, the fitness function was modified so that no divisions were made, i.e. the normalization was removed. That way, the algorithms may

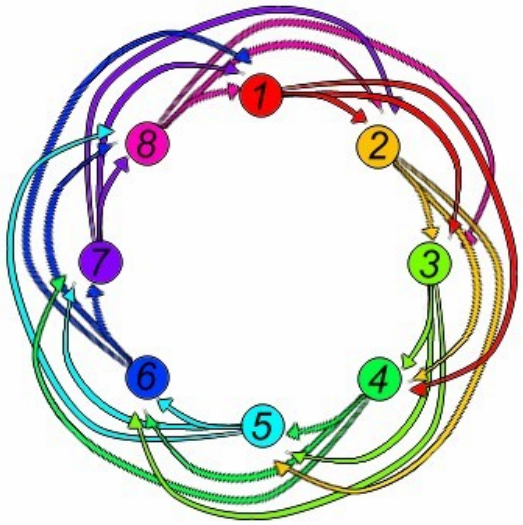

Figure 1: A testing graph with $n = 8$ and $t = 3$.

work with integer values only, and the fitness range will be from 0 to $2nt$, instead of 0 to 1.

To define the parameters for each algorithm, they were run 10 times using several different sets of parameters and testing graphs with $n = 81$ and $t = 40$ (the biggest values considered in these experiments). The set of parameters with the best performance was selected for each algorithm. The chosen configurations were:

- Simple GA: population = 10, crossover rate = 0.1 and mutation rate = 0.01.
- Specialized GA with crossover: population = 10, crossover rate = 0.05 and mutation rate = 0.01.
- Specialized GA without crossover: population = 10 and mutation rate = 0.01.
- PBIL without negative examples: population = 10, learning rate = 0.2, mutation rate = 0.08 and solutions used for update = 1.
- PBIL with negative examples: population = 10, positive learning rate = 0.2, negative learning rate = 0.1, mutation rate = 0.08 and solutions used for update = 1.
- Compact GA: learning rate = 0.15 and mutation rate = 0.08.

Let (n, t) be a testing graph for a t-diagnosable system with n units. The following testing graphs were defined: $(81, 5)$, $(81, 10)$, $(81, 40)$, $(41, 5)$, $(41, 10)$, $(41, 20)$, $(21, 5)$, $(21, 10)$, $(11, 5)$ and $(8, 3)$. Each algorithm was executed 100 times for each of these testing graphs, using the parameters defined above, and their average performance was computed.

4.2 Results

Tables 1 to 4 show the results obtained in the experiments. Table 1 shows the average fitness of the best solution after up to 500 generations. Table 2 shows the percentage of tests that found the optimal solution. Table 3 shows the average number of generations needed for the algorithms to find the optimal solution. The maximum number of generations was defined as being 500. Table 4 shows the average time needed for the algorithms to find the optimal solution, or until the number of generations reached 500.

From the obtained results, it is possible to come up with several conclusions about the algorithms' performances. The simple GA shows a very efficient execution time, but overall results were the worst among all the algorithms. When there is a big difference between the values of n and t, the algorithm was, in most cases, unable to find even a solution that came close to the optimal solution. This probably occurs because in those cases the number of illegal chromosomes generated is too high, and this makes the convergence of results very hard.

The compact GA had the worst execution times. For the largest networks, the number of generations needed to find the solution was very high, but unlike the simple GA, the average fitness of the solutions was frequently close to the optimal one. Therefore, it can be concluded that the compact GA had a good convergence, although it hardly found the optimal solution itself.

PBIL found the optimal solution in almost all cases. Its performance was surpassed only by the specialized GA. It can be inferred that, for this problem, using the negative examples was harmful, because in most cases there was an increase in the execution time, not compensated by a faster convergence. Indeed, in some cases, the convergence was slower when negative examples were used.

The specialized GA, implemented with the changes proposed in this work had the best overall performance among the tested algorithms. Besides having found the optimal solution in almost all cases, the specialized GA did it in a very small number of generations. Indeed, in several cases, the optimal solution was found in the initial population. This shows how important is the approach used to generate the initial population. It is possible to conclude that the specialized GA without crossover had a similar performance to the one with crossover. In the largest networks, the number of generations needed to find the optimal solution was very similar in both approaches, and removing the crossover made the average execution time for the algorithm to reduce slightly. This could indicate that the crossover may not be responsible for the convergence in this problem. However, as in most cases the initial population already had the optimal solution, or at least one solution with a very high fitness, the number of generations in the tests was very small. So, it is not possible to analyze in a conclusive manner the role of crossover in the convergence for this problem.

5. CONCLUSIONS AND FUTURE WORK

This work presented several evolutionary algorithms that can be used to solve system-level diagnosis, as well as optimizations for these algorithms. The proposed algorithms include a simple GA and specialized versions of the compact GA and PBIL. Also presented is a specialized GA based on the one presented in [3], with a with a different method being used to generate the initial population.

Six variations of the proposed algorithms were implemented and tested, so their performance could be compared. Besides the Simple GA and Compact GA, versions of PBIL both with and without negative examples were evaluated, as well as versions of the specialized GA both with and without

Table 1: Average fitness after up to 500 generations.

	Max. Fitness	Simple GA	Spec. GA w/ X-over	Spec. GA w/o X-over	PBIL w/o neg. samples	PBIL w/ neg. samples	cGA
n = 8, t = 3	48	47.88	48	48	48	48	48
n = 11, t = 5	110	110	110	110	110	110	110
n = 21, t = 5	210	131.92	210	210	210	210	201.4
n = 21, t = 10	420	420	420	420	420	420	420
n = 41, t = 5	410	4.1	410	410	410	410	398.9
n = 41, t = 10	820	89.72	820	820	820	820	769.46
n = 41, t = 20	1640	1640	1640	1640	1640	1640	1629.92
n = 81, t = 5	810	0	810	810	810	810	797.86
n = 81, t = 10	1620	0	1620	1620	1620	1620	1573
n = 81, t = 20	3240	0	3240	3240	3240	3240	3030.89
n = 81, t = 40	6480	6457.18	6480	6480	6480	6480	6132.8

Table 2: Percentage of the tests that found the optimal solution.

	Simple GA	Spec. GA w/ X-over	Spec. GA w/o X-over	PBIL w/o neg. samples	PBIL w/ neg. samples	cGA
n = 8, t = 3	99%	100%	100%	100%	100%	100%
n = 11, t = 5	100%	100%	100%	100%	100%	100%
n = 21, t = 5	61%	100%	100%	100%	100%	38%
n = 21, t = 10	100%	100%	100%	100%	100%	100%
n = 41, t = 5	1%	100%	100%	100%	100%	20%
n = 41, t = 10	9%	100%	100%	100%	100%	11%
n = 41, t = 20	100%	100%	100%	100%	100%	85%
n = 81, t = 5	0%	100%	100%	100%	100%	22%
n = 81, t = 10	0%	100%	100%	100%	100%	10%
n = 81, t = 20	0%	100%	100%	100%	100%	11%
n = 81, t = 40	62%	100%	100%	100%	100%	0%

Table 3: Number of generations required to find the solution.

	Simple GA	Spec. GA w/ X-over	Spec. GA w/o X-over	PBIL w/o neg. samples	PBIL w/ neg. samples	cGA
n = 8, t = 3	37.3	2.33	0.94	6.99	12.66	49.46
n = 11, t = 5	41.38	1.1	0.8	16.03	21.62	69.41
n = 21, t = 5	277.11	9.11	9.96	35.36	30.23	378.27
n = 21, t = 10	68.33	2.63	3.53	41.01	45	147.13
n = 41, t = 5	496.72	3.28	2.86	34.25	40.75	400.04
n = 41, t = 10	472.53	5.82	5.36	56.06	67.55	445
n = 41, t = 20	166.71	6.73	8.07	72.1	71.89	305.77
n = 81, t = 5	500	0.87	0.84	65.05	62.85	390
n = 81, t = 10	500	0.93	0.83	64.05	104.29	450
n = 81, t = 20	500	0.88	0.93	126.38	124.02	444
n = 81, t = 40	421.62	0.83	0.76	164.76	142.2	500

Table 4: Average time until the solution was found.

	Simple GA	Spec. GA w/ X-over	Spec. GA w/o X-over	PBIL w/o neg. samples	PBIL w/ neg. samples	cGA
n = 8, t = 3	0.01	0.00	0.00	0.00	0.01	0.01
n = 11, t = 5	0.02	0.00	0.00	0.01	0.02	0.03
n = 21, t = 5	0.07	0.01	0.00	0.09	0.09	0.62
n = 21, t = 10	0.11	0.01	0.00	0.28	0.25	0.22
n = 41, t = 5	0.06	0.03	0.02	0.60	0.86	5.52
n = 41, t = 10	0.15	0.07	0.08	0.12	0.29	6.76
n = 41, t = 20	0.71	0.09	0.05	0.96	0.93	2.23
n = 81, t = 5	0.08	0.10	0.10	0.76	2.06	43.58
n = 81, t = 10	0.09	0.33	0.31	2.48	4.87	81.05
n = 81, t = 20	0.11	0.90	1.02	10.15	12.45	116.2
n = 81, t = 40	26.31	1.40	1.23	16.3	15.74	50.16

crossover. The test results show that the more sophisticated algorithms converge faster to the optimal solution. In the largest networks, this becomes very evident, with the simpler approaches (simple GA, compact GA) not finding the optimal solution or even a good solution, while the most sophisticated solution (specialized GA) finds it very quickly. It can also be noted that the use of negative examples in PBIL was harmful for this problem: in most cases there was an increase in the execution time, not compensated by a faster convergence.

Future work includes employing other evolutionary algorithms, such as the extended compact GA [5] and the BOA [8] for diagnosis. The application of evolutionary algorithms to distributed diagnosis, where all the units in the system solve the diagnosis independently, is also under investigation.

6. REFERENCES

[1] S. Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA, 1994.

[2] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In A. Prieditis and S. Russel, editors, *The International Conference on Machine Learning 1995*, pages 38–46, San Mateo, CA, 1995. Morgan Kaufmann Publishers.

[3] M. Elhadef and B. el Ayeb. An evolutionary algorithm for identifying faults in t-diagnosable systems. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 74–83, 2000.

[4] S. L. Hakimi and A. T. Amin. Characterization of connection assignment of diagnosable systems. *IEEE Transactions on Computers*, 23:86–88, 1974.

[5] G. R. Harik. Linkage learning via probabilistic modeling in the ecga. Technical Report 99010, University of Illinois at Urbana-Champaign., 1999.

[6] G. R. Harik, F. G. Lobo, and D. E. Goldberg. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 3(4):287–297, November 1998.

[7] E. P. D. Jr. and L. C. E. D. Bona. A dependable snmp-based tool for distributed network management. In *IEEE DSN/DCC Symposium*, pages 279–284, 2002.

[8] M. Pelikan, D. E. Goldberg, and E. Cant-Paz. BOA: The Bayesian optimization algorithm. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, volume I, pages 525–532, Orlando, FL, 13-17 1999. Morgan Kaufmann Publishers, San Fransisco, CA.

[9] F. Preparata, G. Metze, and R. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, EC-16(6):848–854, 1967.

[10] A. Subbiah and D. M. Blough. Distributed diagnosis in dynamic fault environments. *IEEE Transactions on Parallel Distributed Systems*, 15(5):453–467, 2004.

APPENDIX

A. PBIL PSEUDOCODE

```
PBIL (n, t, S*)
  prob_vector = array of n positions
  initialize (prob_vector, n, t, S*)
  pop = array of POP_SIZE solutions
  for i = 1 to POP_SIZE
    pop [i] = generateSolution (prob_vector)
  best_solutions = array of #PSAMPLES positions
  best_solutions = bestSolutions (pop)
  generations = 0

  while fitness (best_solutions [1], n, t, S*) < 1
        and generations < MAX_GENERATIONS
  {
    for i = 1 to #PSAMPLES
      for j = 1 to n
        if best_solutions [i][j] = 1
          prob_vector [j] = prob_vector [j] + LRATE
        else
          prob_vector [j] = prob_vector [j] - LRATE

     mutate (prob_vector, MUT_CHANCE, MUT_RATE)

     for i = 1 to POP_SIZE
       pop [i] = generateSolution (prob_vector)
     best_solutions = bestSolutions (pop)
     generations = generations + 1
  }
  return (best_solutions [1])
```

B. PBIL WITH NEGATIVE EXAMPLES PSEUDOCODE

```
PBIL (n, t, S*)
  prob_vector = array of n positions
  initialize (prob_vector, n, t, S*)
  pop = array of POP_SIZE solutions
  for i = 1 to POP_SIZE
    pop [i] = generateSolution (prob_vector)
  best_solutions = array of #PSAMPLES positions
  best_solutions = bestSolutions (pop)
  worst_solutions = array of #NSAMPLES positions
  worst_solutions = worstSolutions (pop)
  generations = 0

  while fitness (best_solutions [1], n, t, S*) < 1
        and generations < MAX_GENERATIONS
  {
    for i = 1 to #PSAMPLES
      for j = 1 to n
        if best_solutions [i][j] = 1
          prob_vector [j] = prob_vector [j] + LRATE
        else
          prob_vector [j] = prob_vector [j] - LRATE

     for i = 1 to #NSAMPLES
      for j = 1 to n
        if worst_solutions [i][j] = 1
          prob_vector [j] = prob_vector [j] - LRATE
        else
```

```
        prob_vector [j] = prob_vector [j] + LRATE

   mutate (prob_vector, MUT_CHANCE, MUT_RATE)

   for i = 1 to POP_SIZE
     pop [i] = generateSolution (prob_vector)
   best_solutions = bestSolutions (pop)
   generations = generations + 1
}
return (best_solutions [1])
```

C. COMPACT GA PSEUDOCODE

```
prob_vector = array of n positions
initialize (prob_vector, n, t, S*)
sol1 = generateSolution (prob_vector)
sol2 = generateSolution (prob_vector)
best = pickBestSolution (sol1, sol2)
worst = pickWorstSolution (sol1, sol2)
generations = 0

while fitness (best, n, t, S*) < 1
      and generations < MAX_GENERATIONS
{
   for i = 1 to n
      if best [i] = 1 and worst [i] = 0
        prob_vector [i] = prob_vector [i] + LRATE
      else if best [i] = 0 and worst [i] = 1
        prob_vector [i] = prob_vector [i] - LRATE

   mutate (prob_vector, MUT_CHANCE, MUT_RATE)

   sol1 = generateSolution (prob_vector)
   sol2 = generateSolution (prob_vector)
   best = pickBestSolution (sol1, sol2)
   worst = pickWorstSolution (sol1, sol2)
   generations = generations + 1
}
return (best)
```