Chapter 6

Experiments with synthetic examples

'Learning is not attained by chance, it must be sought for with ardor and attended to with diligence. '

Abigail Adams

6.1 Introduction

This chapter introduces three different studies leading to demonstrate the validity of the EDA approach and its behavior for the application to inexact graph matching problems. Different aspects already considered but not shown with experimental tests are described in these studies.

In order to avoid at the maximum the influence of a particular real problem on the behavior of EDAs and other algorithms, random or synthetic attributed graphs have been created and will be used as a starting point in all the experiments in this chapter. The aim of the different sections is as follows: Study 1 analyzes and compares the performance of EDAs and other evolutionary computation techniques such as GAs. Study 2 is an illustration of what happens inside complex EDAs and tries to make clearer the way of approaching better solutions. For this, the best structure that the probabilistic graphical models adopts for better representing the selected individuals of each generation is shown. Finally, in Study 3 the parallelization process of a complex algorithm such as EBNA is carried out, and an easily adaptable method for other similar complex approaches such as EGNA is introduced and justified with experimental data showing the considerable improvement in execution time that the proposed method obtains.

6.2 Study 1: measurement of the performance

6.2.1 Design of the experiment

Three different synthetic examples of graphs have been created randomly using different sizes of model graphs –G_M– and data graphs –G_D. The sizes of these graphs are as follows from the smallest to the biggest: in the first example the model graph G_M contains 10 vertices and 15 edges, and the data graph G_D has 30 vertices and 39 edges. For the second example the graph G_M contains 30 vertices and 39 edges, and the graph G_D 100 vertices and 247

edges. Finally, in the third example G_M contains 50 vertices and 88 edges, and the graph G_D has 250 vertices and 1681 edges. The sizes of these graphs have been chosen carefully. Firstly, the size for the graphs of the first –small– example has been selected based on the explanations given for a real problem in [Boeres et al., 1999, Boeres, 2002], which shows a reduced example of the inexact graph matching problem of graphs extracted from healthy human brain images introduced in [Perchant et al., 1999] and [Perchant and Bloch, 1999]. The size of the graphs for the third example are similar to the ones introduced in [Perchant et al., 1999], in which a model graph G_M that contains 43 vertices and 336 edges is matched against another graph G_D that contains 245 vertices and 1451 edges. Finally, the graphs for the second example are half way between the first and the third examples' ones. In what follows, we will call 10 & 30 example, 30 & 100 example, and 50 & 250 example to the small, second, and big graph matching cases respectively.

The number of edges chosen for all these graphs were selected knowing that the value returned by this fitness function does not depend on $|E_M|$ and $|E_D|$. Following the classification of graphs between sparse and dense introduced in [Larrañaga et al., 1997], the number of edges have been chosen to be the median of the sparse graphs of that size.

As in every optimization problem, a fitness function has to be defined in order to evaluate the goodness of any of the possible solutions. The fitness function $f_2(h)$ introduced in Section 3.4 (Equation 3.2) was selected for the experiments in this study just as an example because of its previous use in real graph matching problems [Boeres, 2002, Perchant and Bloch, 1999, Perchant et al., 1999].

For all the three cases, both the model and data graphs G_M and G_D have been generated randomly from scratch. The fact that no image processing is performed in the graph construction avoids the dependence of the image processing techniques in the final result. Obviously, as these graphs are generated randomly, they do not represent neither any knowledge nor any common segments, and we do not have previous knowledge about which is the optimum matching between vertices of G_M and G_D. As well as both graphs, $c_N(a_D, a_M)$ and $c_E(e_D, e_M)$ were also generated randomly, and α is assigned a value of 0.8 because the best results are obtained with this value in [Boeres et al., 1999] for their particular application. This value is taken as an example for these experiments too.

As the aim of the experiments with these three synthetic examples is to test the performance of EDAs in general, we have selected three discrete EDAs introduced in Section 4.5.1, as well as four continuous EDAs introduced in Section 4.5.2. Because the main difference between EDAs in both domains is the number of dependencies between variables that they can take into account, the fact of having graphs with different sizes will influence parameters such as the best solution obtained after a number of generations, the time to compute the algorithm, and the evolution of the algorithm itself through the search. This section describes the experiments and the results obtained[1]. The three discrete EDA algorithms are also compared to three broadly known GAs: canonic basic (cGA) [Holland, 1975], elitist (eGA) [Michalewicz, 1992] and steady state (ssGA) [Whitley and Kauth, 1988]. The two first GAs evolve from a population to another by applying crossover operations to some individuals in the population, and the difference between them is that the eGA always includes in the new population the best individual of the previous one, whereas cGA does not. The ssGA approach is somehow different, as it only generates one individual at each iteration, replacing only the worst individual of the population when its fitness value is worse than the

[1] A review of this work focusing only on the discrete domain can also be found in [Bengoetxea et al., 2001a, 2002a].

one of the new individual.

EDAs and GAs were implemented in ANSI C++ language, and the experiments were executed on a two processor Silicon Graphics machine SGI-Origin200 under IRIX OS version 64-Release 6.5 with 500 Mbytes of RAM.

In the discrete case, all the programs were designed to finish the search when the whole population is formed by the same repeated individual or when a maximum of 100 generations was reached. GAs were programmed to generate the same number of individuals as with discrete EDAs, and therefore 100 generations were executed for all of them. The ssGA is a special case because of generating a single individual at each iteration, but it was also programmed in order to generate the same number of individuals by allowing more iterations. In the continuous case, the algorithms were designed to finish when the 150th generation was reached.

The initial population for all the discrete algorithms was generated using the same random generation procedure based on a uniform distribution for all the possible values, and in the case where the correction of the individuals applies, both discrete EDAs and GAs were programmed using the same correction procedures. In the same way, the fitness function used in all the algorithms is exactly the same. In the continuous case, the generation of the first generation was also done following a similar procedure for generating continuous values.

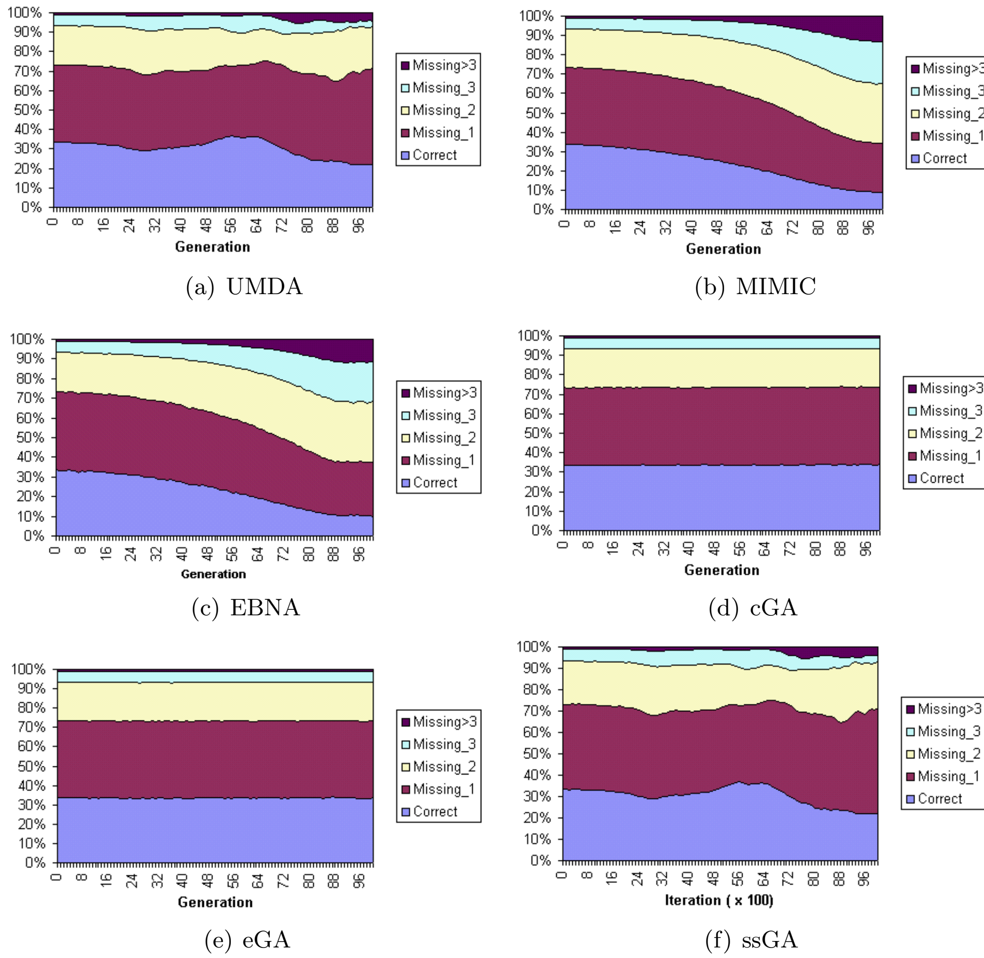
In EDAs of both domains, the following parameters were used: a population of 2000 individuals ($R = 2000$), from which the best 1000 are selected ($N = 1000$) to estimate the probability, and the elitist approach was selected (that is, always the best individual is included for the next population and 1999 individuals are simulated). In GAs a population of 2000 individuals was also selected, with a mutation probability of $1.0/|V_D|$ and a crossover probability of 1.

6.2.2 The need to obtain correct individuals

As one of the aims of this study is to analyze the behavior of EDAs in ordinary problems, we decided to add extra constraints to the problem. For this, three conditions have been introduced in Section 3.3.3 that need to be satisfied for any solution. It is important to realize that both in GAs and discrete EDAs the individuals generated every generation could contain an incorrect solution (that is, the solution might not satisfy the three conditions introduced in Section 3.3.3). That is why some techniques to avoid the presence of incorrect individuals have been introduced in Section 4.5.1. Continuous EDAs do not have such a problem, as the individual representation chosen avoids it completely.

Nevertheless, as using the techniques to obtain correct individuals introduced in Section 4.5.1 implies a computational cost as well as a direct and permanent manipulation on the population itself, it is important to check whether the percentage of incorrect individuals for the different algorithms is high enough to justify such a correction. We will take our 30 & 100 case as an example to confirm whether the additional manipulation process is required or not.

Figure 6.1 shows the percentages of correct and incorrect individuals during the search process for the three discrete Estimation of Distribution Algorithms (UMDA, MIMIC and EBNA), as well as for the cGA, eGA and ssGA in the 30 & 100 example without applying any technique to correct the wrong individuals (PLS only). Similarly, Figure 6.2 shows the results of applying penalization under the same conditions. These graphics illustrate the mean results of 20 executions for each of the algorithms. For each graphic the x axis represents the generation number, and the y axis the percentage of individuals that are



(a) UMDA (b) MIMIC

(c) EBNA (d) cGA

(e) eGA (f) ssGA

Figure 6.1: Figures for the correctness of the UMDA, MIMIC and EBNA (discrete EDAs) as well as for the cGA, eGA and ssGA (GAs), applied to the second example of 30 & 100 vertices without correction.

correct (i.e. all the vertices in V_M have been assigned in the matching), the ones where a value is missing (i.e. when only one vertex of V_M has not been assigned), the ones where two values are missing (i.e. when two vertices of V_M have not been assigned), when three values are still to be assigned, and finally the individuals in which more than three vertices have not been assigned to a data vertex. These graphics illustrate that at the final generation of all the algorithms practically none of the individuals is acceptable, but the percentage of correct individuals decreases sooner when increasing the size of the graphs. It is important to note that in this 30 & 100 example the individuals have a total size of 100 variables or genes, and each of them have to be assigned to a value between 1 and 30 ($|V_M| = 30$ and $|V_D| = 100$) has to be assigned to each of them, thus it is more probable that at least one of the vertices in V_M has not been assigned in the variables or genes than in the 10 & 30 example.

From the results we can conclude that for graphs of complexity similar or higher than in this 30 & 100 example, the number of correct individuals gradually decreases every generation for all the algorithms. Furthermore, the percentage of individuals that have one vertex of V_M not matched in the last generation appears to be very high for all the algorithms. These

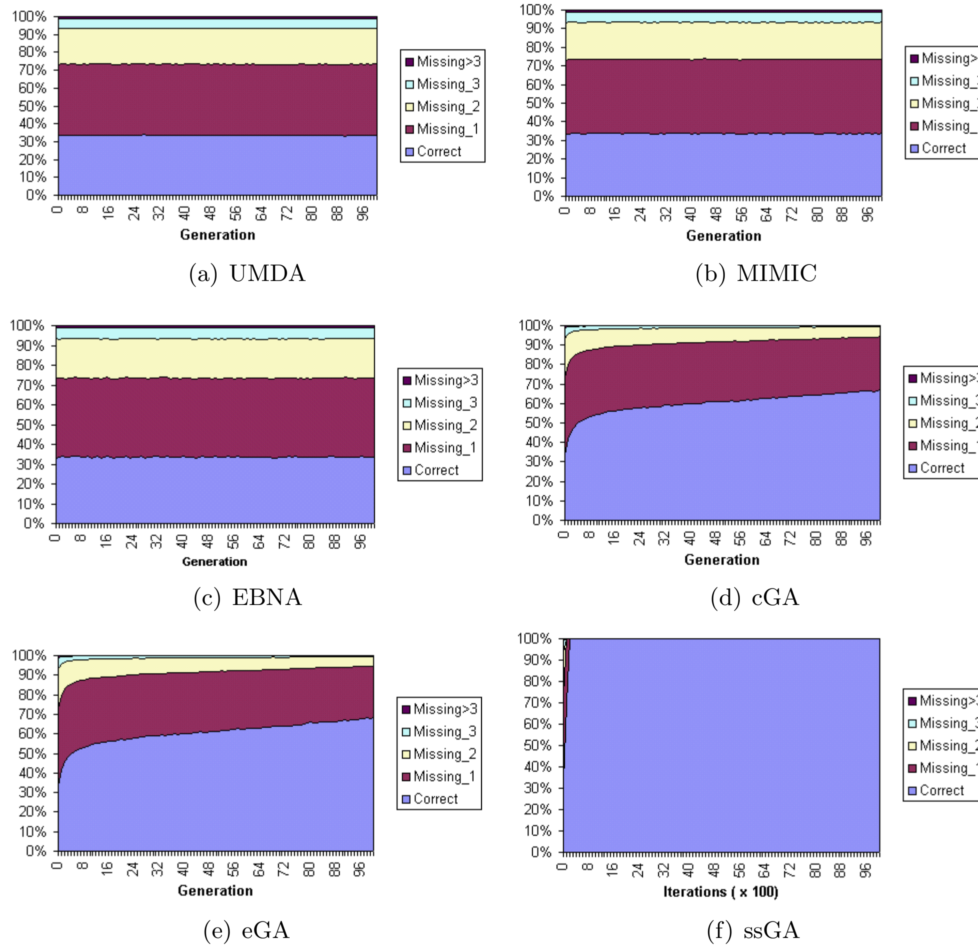


Figure 6.2: Figures for the correctness of the UMDA, MIMIC and EBNA discrete EDAs as well as for the cGA, eGA and ssGA GAs, applied to the second example of 30 & 100 vertices and using penalization.

graphs show clearly the behavior of the different algorithms, as well as the nature of the fitness function selected. As a result, we can conclude that without any mechanism to correct or guide the generation of individuals the percentage of correct ones will decay quite fast. This decay can be appreciated in both discrete EDAs and GAs, but for the case of EDAs the problem appears to be much more important, as when applying PLS simulation alone none of the three EDA contains any correct individuals at the last generation of the search. On the other hand, the three types of GAs do also contain a high proportion of incorrect individuals at the last generation, and therefore this does not ensure that the final solution returned by the algorithm will be a correct one.

Another conclusion about the penalization procedure can also be obtained from these results: the penalization of the incorrect individuals is the only correction method that does not manipulate the learning and simulation steps (the others are LTM and ATM as explained in Section 4.5.1), but whichever the penalization weight to the incorrect individuals there will always be the possibility of finding incorrect individuals in the final population. In fact, the proportion of incorrect individuals for penalization in UMDA, MIMIC, EBNA, cGA, eGA and ssGA were of 33.66%, 33.69%, 33.69%, 66.81%, 68.32% and 100% respectively. A

stronger penalization is required to be applied to individuals if this method is to be selected for graphs with as many vertices as in these three examples, but it would never ensure 100% of correct individuals in the population.

6.2.3 Discrete domain

Experimental results by combining correction methods and algorithms

Once proved the need to control the generation of the individuals in every population, the three methods described in Section 4.5.1 for discrete domains were combined with the three discrete EDA algorithms. In the case of the GAs, the last two methods described in the same section were used for cGA eGA and ssGA, as the ones based on the modification of the probability in the simulation step do not apply in GAs which do not perform such a step.

The results obtained from the different executions of the algorithms are shown in this section. For each algorithm and example the mean values of the fitness value of the best individual at the last generation, the number of generations to reach the final solution, and the computation time are shown.

The computation time presented in these experimental results is actually the CPU time of the process from the beginning to the end, and therefore it is not dependent on the variations on the multiprogramming level during the execution time. This computation time is presented as a measure to illustrate the different computation complexity of all the algorithms. It is important also to note that all the operations for the estimation of the distribution, the simulation, and the evaluation of the new individuals are carried out through memory operations.

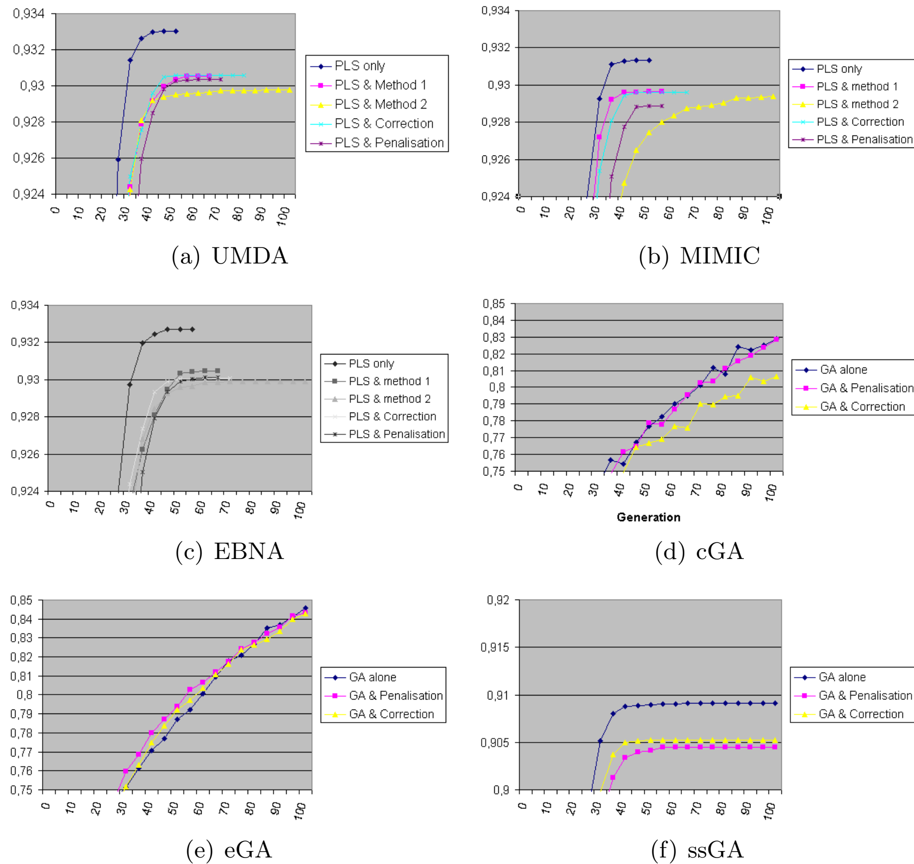
The null hypothesis of the same distribution densities was tested for each of the different algorithms and for each of the correction methods to control the generation of new individuals. The non-parametric test of Kruskal-Wallis was used [Kruskal and Wallis, 1952][2]. This task was carried out with the statistical package S.P.S.S. release 9.00.

Discrete domain: best individual, number of generations required, and computation time

The results of the 10 & 30 example are shown in Figure 6.3 and Tables 6.1, 6.2, and 6.3. Figure 6.3 is done with the mean of 20 executions for all the algorithms, showing their different behaviors depending on the correction method employed. The reader is reminded that the *PLS only* and *Penalization* methods do not ensure a population of only correct individuals. Shorter lines indicate that the algorithm finishes requiring less generations. Tables 6.1, 6.2, and 6.3 show also the mean values as well as the results of applying the Kruskal-Wallis test to the different parameters (fitness value, number of generations required, and execution time required).

In an analogous way, results for the 30 & 100 example are shown in Figure 6.4 and Tables 6.4, and 6.5. Results for the 50 & 250 example are found in Figure 6.5 and Tables 6.6, and 6.7. These tables show also the mean values showing their behavior when changing the correction method employed as well as the results of applying the Kruskal-Wallis test to the different parameters (metric and execution time required). The number of generations reached for the 10 & 30 and 30 & 100 examples was of 100 for all the algorithms.

[2]The interested reader is referred to [Siegel, 1956] for a deep explanation on non-parametric tests.



(a) UMDA (b) MIMIC

(c) EBNA (d) cGA

(e) eGA (f) ssGA

Figure 6.3: Graphs showing the best individual at each generation of the searching process for the algorithms UMDA, MIMIC, EBNA, cGA, eGA, and ssGA for the case of the 10 & 30 vertex graphs. Note the different scales between discrete EDAs and GAs.

At the light of the results obtained in the fitness values, we can conclude that from the three GAs used, ssGA appears clearly as the one that obtains the best results. Furthermore, the computation time to generate the final solution is also less than the one required by the other two GAs.

There is little difference in the best individual obtained by the different discrete EDAs: even if with some correction methods EBNA obtains the best results, in some cases such as in LTM and penalization, UMDA returned the best results. As explained before, EBNA was expected to return the best result due to its ability to estimate more accurately the probability, in spite of the higher computational cost. Nevertheless, as the differences do not appear to be significant, we could not conclude that any algorithm is superior to the rest simply based on these experimental results. Even if in EBNA no restrictions are set to the structure to learn, the results obtained could indicate that the most appropriate structure for this problem could be a structure with at most pairwise dependencies. It is also important to note that both graphs G_M and G_D have been created at random and that they do not show any knowledge, which makes the matching process even more dependent on the fitness function used.

Regarding the difference between discrete EDAs and GAs, it seems clear that EDAs obtain better results for any of the correction methods applied to the individuals. It is

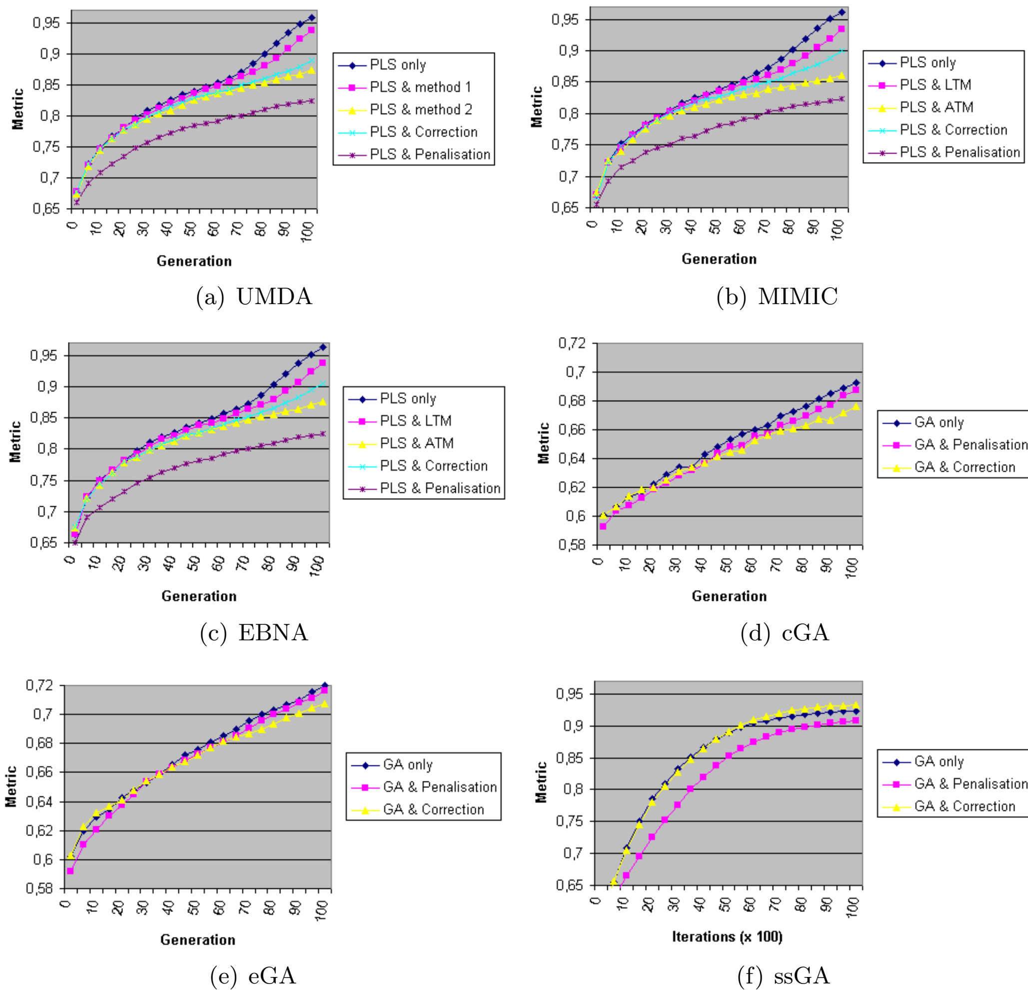


Figure 6.4: Graphs showing the best individual at each generation of the searching process for the algorithms UMDA, MIMIC, EBNA, cGA, eGA, and ssGA for the case of the 30 & 100 vertex graphs. Note the different scales between discrete EDAs and GAs.

important therefore to note that ssGA obtains similar results as EDAs.

Additional results. Additionally, the Kruskal-Wallis test was also applied to the correction methods between discrete EDAs only, and the non-parametric test of Mann-Whitney [Mann and Whitney, 1947] was carried out for GAs only. The results are shown in Table 6.8.

Another important aspect to remember is the control of the correctness of the individuals. As the LTM and ATM imply the manual modification of the learned model by changing the probabilities, the learning itself is somehow manipulated. The correction of individuals does also modifies at random some of the individuals of the population. The penalization of the incorrect individuals is the only correction method that does not manipulate the learning and simulation steps, but whichever the penalization weight to the incorrect individuals there will always be the possibility of containing incorrect individuals in the final population. A stronger penalization could improve these values, but it would never ensure 100% of correct individuals in the population.

Regarding the fitness values of the best individuals obtained at the end of the search

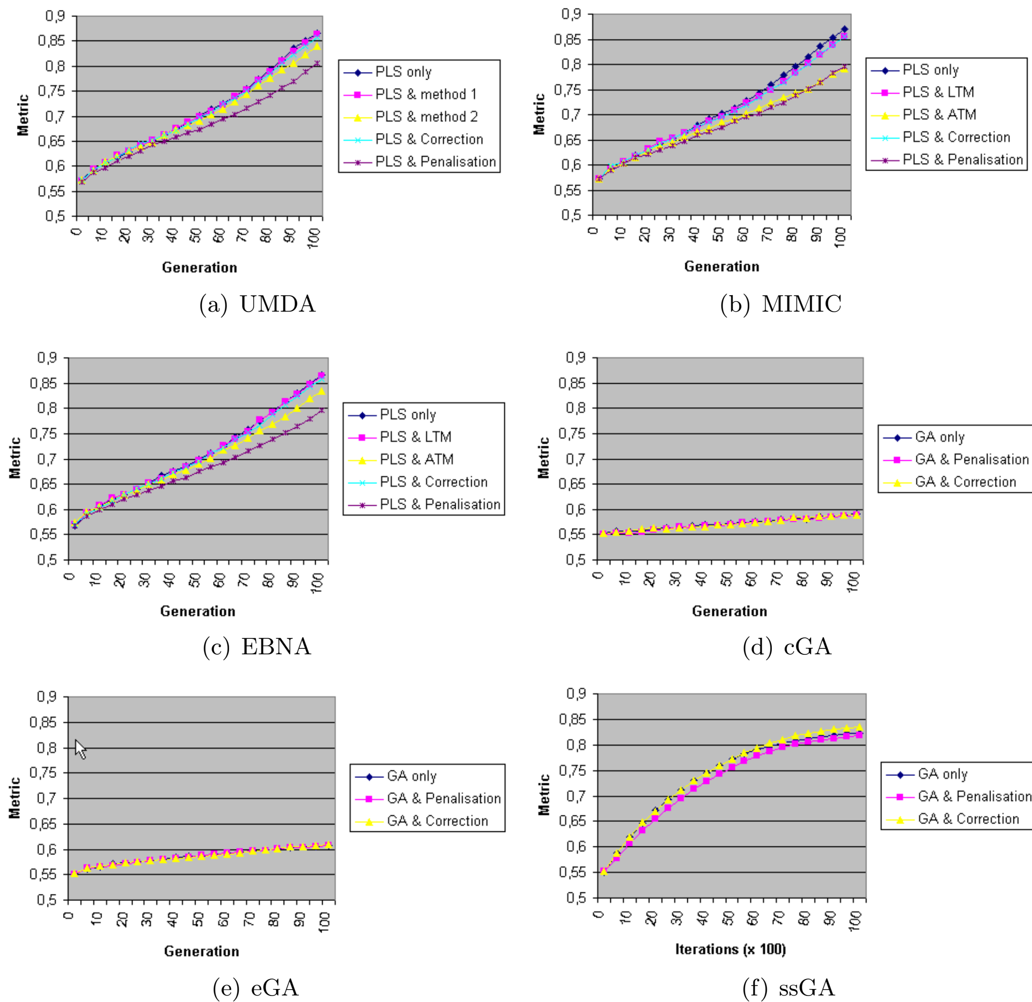


Figure 6.5: Graphs showing the best individual at each generation of the searching process for the algorithms UMDA, MIMIC, EBNA, cGA, eGA, and ssGA for the case of the 50 & 250 vertex graphs. Note the different scales between discrete EDAs and GAs.

processes, we can conclude that from the three GAs used, ssGA appears clearly as the one that obtains the best results. Furthermore, the computation time to generate the final solution is also less than the one required by the other two GAs. The best individuals obtained using the different EDAs are very similar: even if with some correction methods EBNA obtains the best results, in some cases such as in LTM and penalization UMDA performs better. As explained before, EBNA is expected to return better results due to its ability to estimate more accurately the probability distribution every generation, in spite of a higher computational cost. Nevertheless, the small differences between EDAs do not appear to be significant for this example regarding Table 6.8. This effect can be explained by the fact that both graphs have been created at random and that they should not reflect any dependence between variables, and as a result EBNA cannot find more dependencies than other simpler EDAs. On the other hand, when comparing EDAs and GAs, it appears clearly that EDAs obtain better results using any of the correction methods applied to the individuals. It is important to note however that only ssGA obtains nearly as good results as EDAs.

	Method 1	Method 2	Correction	Penalization	Statistical Significance
UMDA	0.9305	0.9297	0.9305	0.9303	$p = 0.016$
MIMIC	0.9296	0.9293	0.9296	0.9289	$p = 0.242$
EBNA	0.9304	0.9299	0.9301	0.9301	$p = 0.320$
cGA	–	–	0.8065	0.8285	$p < 0.001$
eGA	–	–	0.8428	0.8431	$p = 0.766$
ssGA	–	–	0.9053	0.9045	$p = 0.636$
Statistical Significance	$p < 0.001$	$p = 0.283$	$p < 0.001$	$p < 0.001$	

Table 6.1: Best fitness values for the 10 & 30 example (mean results of 20 runs).

	Method 1	Method 2	Correction	Penalization	Statistical Significance
UMDA	50.85	67.00	49.90	53.65	$p = 0.023$
MIMIC	42.00	92.70	46.90	47.40	$p < 0.001$
EBNA	52.00	63.40	51.20	54.55	$p = 0.088$
cGA	–	–	100	100	$p = 1.000$
eGA	–	–	100	100	$p = 1.000$
ssGA	–	–	100	100	$p = 1.000$
Statistical Significance	$p < 0.001$	$p < 0.001$	$p < 0.001$	$p < 0.001$	

Table 6.2: Number of required generations for the 10 & 30 example (mean results of 20 runs).

	Method 1	Method 2	Correction	Penalization	Statistical Significance
UMDA	00:58	01:25	01:01	01:32	$p < 0.001$
MIMIC	00:56	02:39	01:02	01:32	$p < 0.001$
EBNA	03:37	04:25	03:38	04:34	$p < 0.001$
cGA	–	–	01:00	01:00	$p = 0.100$
eGA	–	–	01:01	01:01	$p = 0.100$
ssGA	–	–	01:09	01:09	$p = 0.747$
Statistical Significance	$p < 0.001$	$p < 0.001$	$p < 0.001$	$p < 0.001$	

Table 6.3: Time to compute for the 10 & 30 example (mean results of 20 runs, in mm:ss format).

6.2.4 Continuous domain

In an analogous way as in the discrete domain, continuous EDAs were also tested in order to check their performance in graph matching problems. The same three examples were taken and were executed 20 times each. The results of the experiment are shown in Figures 6.6, 6.7 and 6.8, and Table 6.9.

This table shows that the differences between the algorithms in the discrete and continu-

	Method 1	Method 2	Correction	Penalization	Statistical Significance
UMDA	0.940502	0.874684	0.892806	0.825850	$p < 0.001$
MIMIC	0.936400	0.859538	0.898960	0.824063	$p < 0.001$
EBNA	0.936739	0.875429	0.905114	0.823836	$p < 0.001$
cGA	–	–	0.674490	0.687297	$p = 0.004$
eGA	–	–	0.706609	0.712994	$p = 0.160$
ssGA	–	–	0.932318	0.911038	$p < 0.001$
Statistical Significance	$p = 0.773$	$p < 0.001$	$p < 0.001$	$p < 0.001$	

Table 6.4: Best fitness values for the 30 & 100 example (mean results of 20 runs).

	Method 1	Method 2	Correction	Penalization	Statistical Significance
UMDA	00:11:50	00:12:56	00:11:59	00:13:05	$p < 0.001$
MIMIC	00:17:19	00:18:24	00:17:29	00:18:50	$p < 0.001$
EBNA	03:18:06	03:19:06	03:18:13	03:19:16	$p < 0.001$
cGA	–	–	00:09:07	00:09:08	$p < 0.001$
eGA	–	–	00:09:07	00:09:07	$p = 1.000$
ssGA	–	–	00:09:03	00:09:04	$p = 0.317$
Statistical Significance	$p < 0.001$	$p < 0.001$	$p < 0.001$	$p < 0.001$	

Table 6.5: Time to compute for the 30 & 100 example (mean results of 20 runs, in hh:mm:ss format).

	Method 1	Method 2	Correction	Penalization	Statistical Significance
UMDA	0.863936	0.840546	0.856377	0.805633	$p < 0.001$
MIMIC	0.854939	0.792449	0.855580	0.796515	$p < 0.001$
EBNA	0.863677	0.833811	0.858611	0.795378	$p < 0.001$
cGA	–	–	0.587868	0.588509	$p = 0.725$
eGA	–	–	0.608552	0.607220	$p = 0.725$
ssGA	–	–	0.835702	0.818191	$p < 0.001$
Statistical Significance	$p = 0.050$	$p < 0.001$	$p < 0.001$	$p < 0.001$	

Table 6.6: Best fitness values for the 50 & 250 example (mean results of 20 runs).

	Method 1	Method 2	Correction	Penalization	Statistical Significance
UMDA	01:47:34	01:51:16	01:47:54	01:49:26	$p < 0.001$
MIMIC	02:45:43	02:50:07	02:45:45	02:47:26	$p < 0.001$
EBNA	53:01:35	53:08:04	53:03:35	52:59:49	$p = 0.001$
cGA	–	–	01:40:23	01:40:15	$p = 0.297$
eGA	–	–	01:40:35	01:40:34	$p = 0.925$
ssGA	–	–	01:40:50	01:40:39	$p = 0.180$
Statistical Significance	$p < 0.001$	$p < 0.001$	$p < 0.001$	$p < 0.001$	

Table 6.7: Time to compute for the 50 & 250 example (mean results of 20 runs, in hh:mm:ss format).

First Experiment(10 & 30)

	between EDAs only	between GAs only
Correction	Fitness value: $p = 0.139$ Generations: $p < 0.001$ Time: $p < 0.001$	Fitness value: $p < 0.001$ Generations: $p = 1.000$ Time: $p < 0.001$
Penalization	Fitness value: $p < 0.001$ Generations: $p < 0.001$ Time: $p < 0.001$	Fitness value: $p < 0.001$ Generations: $p = 1.000$ Time:$p < 0.001$

Second Experiment(30 & 100)

	between EDAs only	between GAs only
Correction	Fitness value: $p = 0.164$ Time: $p < 0.001$	Fitness value: $p < 0.001$ Time: $p < 0.001$
Penalization	Fitness value: $p = 0.471$ Time: $p < 0.001$	Fitness value: $p < 0.001$ Time:$p < 0.001$

Third Experiment(50 & 250)

	between EDAs only	between GAs only
Correction	Fitness value: $p = 0.886$ Time: $p < 0.001$	Fitness value: $p < 0.001$ Time: $p = 0.012$
Penalization	Fitness value: $p = 0.787$ Time: $p < 0.001$	Fitness value: $p < 0.001$ Time:$p = 0.025$

Table 6.8: Particular non-parametric tests for the 10 & 30, 30 & 100 and 50 & 250 examples. The cases where the generations in GAs are $p = 1.000$ indicate that all GAs executed during 100 generations. These are the mean results of 20 runs for each algorithm.

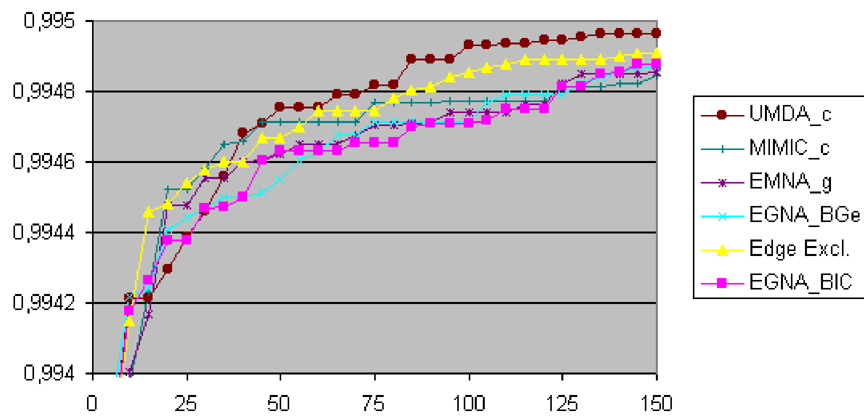


Figure 6.6: Graphs showing the best individual of the 10 & 30 case at each generation of the searching process for the continuous EDAs UMDA$_c$, MIMIC$_c$, EGNA$_{BGe}$, EGNA$_{BIC}$, EGNA$_{ee}$, and EMNA$_{global}$.

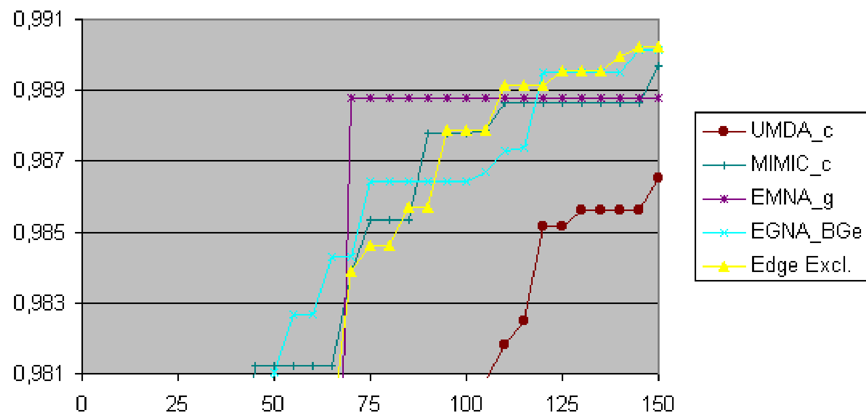


Figure 6.7: Graphs showing the best individual of the 30 & 100 case at each generation of the searching process for the continuous EDAs UMDA$_c$, MIMIC$_c$, EGNA$_{BGe}$, EGNA$_{ee}$, and EMNA$_{global}$.

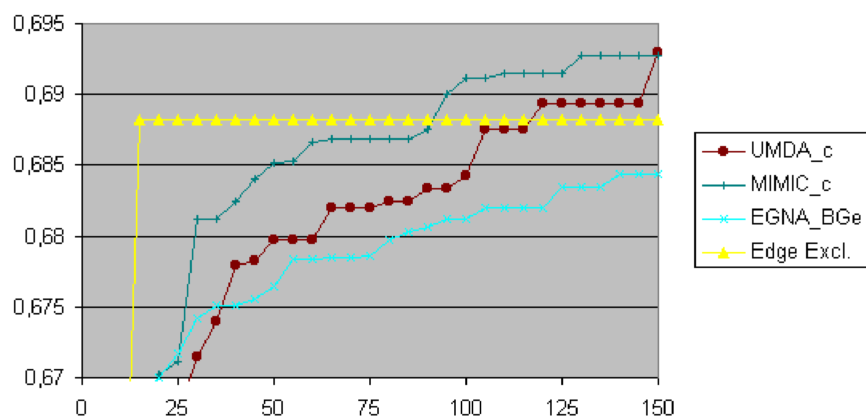


Figure 6.8: Graphs showing the best individual of the 50 & 250 case at each generation of the searching process for UMDA$_c$, MIMIC$_c$, EGNA$_{BGe}$, and EGNA$_{ee}$.

	10 & 30 best	10 & 30 time	30 & 100 best	30 & 100 time	50 & 250 best	50 & 250 time
$UMDA_c$	0.994964	00:10:36	0.986516	01:33:29	0.693013	10:13:14
$MIMIC_c$	0.994844	00:11:09	0.989696	01:35:29	0.692722	10:18:56
$EGNA_{ee}$	0.994909	00:14:29	0.990219	07:26:56	0.688189	67:08:11
$EGNA_{BGe}$	0.994887	00:13:23	0.990141	03:09:57	0.684401	461:59:00
$EGNA_{BIC}$	0.994878	02:29:41	–	–	–	–
$EMNA_{global}$	0.994855	00:32:09	0.988799	97:45:25	–	–

Table 6.9: Figures of the 3 cases of Study 1 for the continuous EDAs, obtained as the mean values after 20 executions of the continuous EDAs. The *best* column corresponds to the best fitness value obtained through the search.

ous domains are significant for all the algorithms analyzed. The null hypothesis of the same distribution densities was tested (non-parametric tests of Kruskal-Wallis and Mann-Whitney) for each of them with the statistical package S.P.S.S. release 9.00. These tests confirmed the significance of the differences in the results regarding the value of the best solution obtained. It is important to remember that all the solutions obtained by the continuous representation are correct, and therefore these results can be compared directly to any of the correction methods described for the discrete case. In many continuous versions of the EDA algorithms fitter results were obtained at the end of the search than their respective discrete versions, and it was only on the 50 & 250 example, where the results obtained by continuous EDAs are worse than the discrete EDAs. The main drawback of continuous EDAs is the longer execution time they require, which is extremely larger for the case of more complex continuous EDAs such as $EGNA_{BGe}$. In $EGNA_{BIC}$ and $EMNA_{global}$ the execution time was so high that after 500 hours of execution time the processes were aborted. These results show clearly that the behavior of selecting a discrete learning algorithm or its equivalent in the continuous domain is very different regarding all the parameters analyzed.

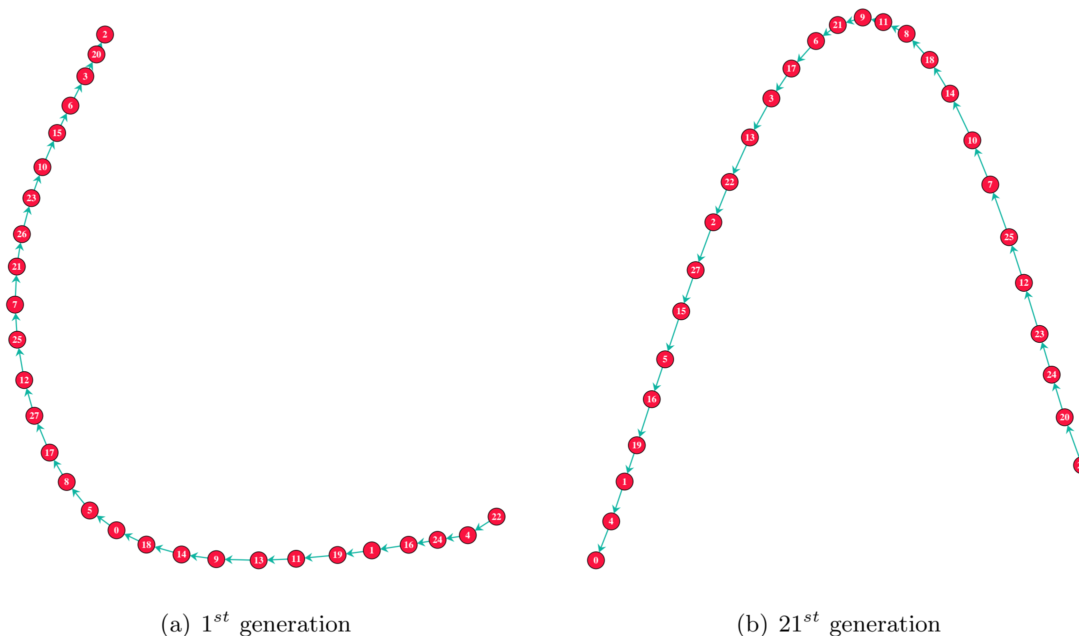
It is important to note that the number of evaluations was different as the ending criteria for the discrete and continuous domains have been set to be different. In all the cases, the continuous algorithms obtained a fitter individual, but the CPU time and number of individuals created was also bigger.

At the light of the results obtained in the fitness values, we can conclude the following: generally speaking, continuous algorithms perform better than discrete ones, either when comparing all of them in general or when only with algorithms of equivalent complexity.

6.3 Study 2: evolution of probabilistic graphical structures

The aim of this study is to analyze the evolution of the probabilistic graphical model complexity (the Bayesian network in the discrete case and the Gaussian network in the continuous one) so that the reader can have an idea of the complexity of the graph matching problem and the behavior of each of the different EDAs.

Due to the difficulty of visualizing a structure with as many nodes as shown in the graphs of the previous study, a smaller real example has been chosen. This example is taken from an image of human muscle cells, where the model graph G_M contains a vertex for each of the cells in the image. This image was over-segmented and the data graph G_D was obtained from it. In this particular example the graph G_M contains 14 vertices and 66



(a) 1st generation   (b) 21st generation

Figure 6.9: Figures showing the structures learned during the MIMIC search in discrete EDAs.

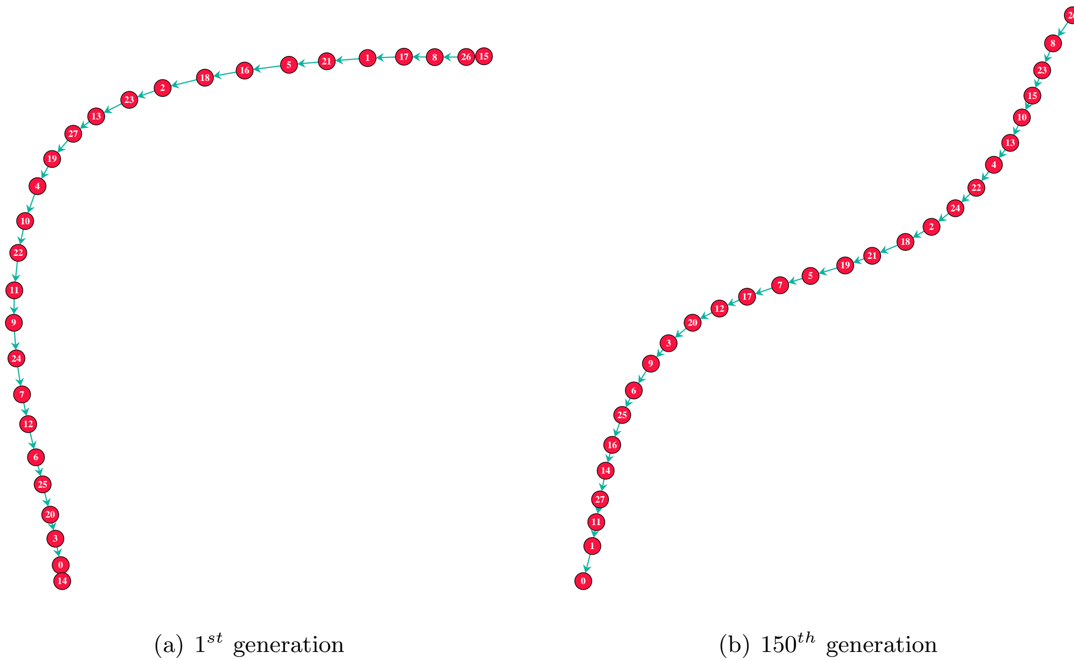
edges, and the data graph G_D contains 28 vertices and 124 edges. EDAs were applied to this example using individuals with a size of 28 variables ($|V_D| = 28$) in both discrete and continuous domains. Therefore, all the probabilistic graphical structures generated during the successive generation in the EDA approach using this representation of individuals is of 28 vertices. The number of edges of these structures symbolizes the number of dependencies between the different regions of the data image that the algorithm detects.

The discrete UMDA example is not shown in any figure, as it does consider all the variables as having no interdependencies. The assumed structure is the same as for UMDA$_c$, and it is shown in Figure 6.13a.

With discrete EDAs we obtain structures such as the ones illustrated in Figure 6.9 (for the MIMIC approach) and Figure 6.11 (for EBNA). These two examples show clearly that the algorithm is learning a structure according to the complexity we expected: MIMIC takes into account pairwise dependencies and generates a structure in the form of a chain in every generation, and only the order of the variables changes during the search. The EBNA algorithm imposes no restrictions to the number of dependencies that a variable can have, and therefore there is no limitation in the number of arcs that a node can have in every generation.

With continuous EDAs we can appreciate the analogous behavior: the continuous MIMIC$_c$ case is illustrated in Figure 6.10, where a behavior similar to Figure 6.9 can be seen. Again, this was expected as the discrete MIMIC does also consider pairwise dependencies. Nevertheless, the estimation of the distribution is performed using different methods in both algorithms according to the domain of the variables (MIMIC generated a Bayesian network and MIMIC$_c$ a Gaussian network).

With continuous EDAs we obtain structures such as the ones illustrated in Figure 6.10 (for the MIMIC$_c$ approach) and Figure 6.12 (for EGNA$_{ee}$). These two examples show clearly that the algorithm is learning a structure according to the complexity we expected: MIMIC$_c$



(a) 1st generation (b) 150th generation

Figure 6.10: Figures showing the structures learned during the MIMIC$_c$ continuous EDA.

takes into account pairwise dependencies and generates a structure in the form of a chain in every generation, and the EGNA$_{ee}$ algorithm imposes no restrictions to the number of dependencies that a variable can have (there is no limitation in the number of arcs that a node can have in every generation).

Figure 6.13 is a special case, as the structures of both the UMDA$_c$ algorithm and EMNA$_{global}$ are always fixed during the whole search process (i.e. the estimation of the probabilities does not imply the learning of a structure, this is fixed), and therefore the structure is considered to exist as a fixed one.

We can appreciate in these experiments as well as in others such as the ones mentioned in [Bengoetxea et al., 2001c,e] that EBNA and EGNA algorithms, although they are analogous in complexity in their domains, they have different tendencies. Both algorithms do not set any restriction to the number of dependencies that variables can have (i.e. the probabilistic structures can have any number of arcs for each node). In EBNA, the algorithm tends to finish with an arc-less structure, which is influenced by the fact that in the last generations the best individual appears many times in the population, and therefore the algorithm finds the same value in a variable too often to detect dependencies regarding the rest of the variables –see Figure 6.11. In EGNAs, values are continuous and cannot be repeated as easily as in the discrete domain, and therefore as values are different the dependencies can also be found and represented as arcs in the structure. This is why at the last generations of the search EGNAs show structures with a lot of arcs. This effect can be appreciated for both EBNA and EGNA in Figure 6.14. In addition, there are also some additional factors that may influence this result:

- The complexity of the problem is not as high as expected, and therefore in this case UMDA and EBNA would return similar results.

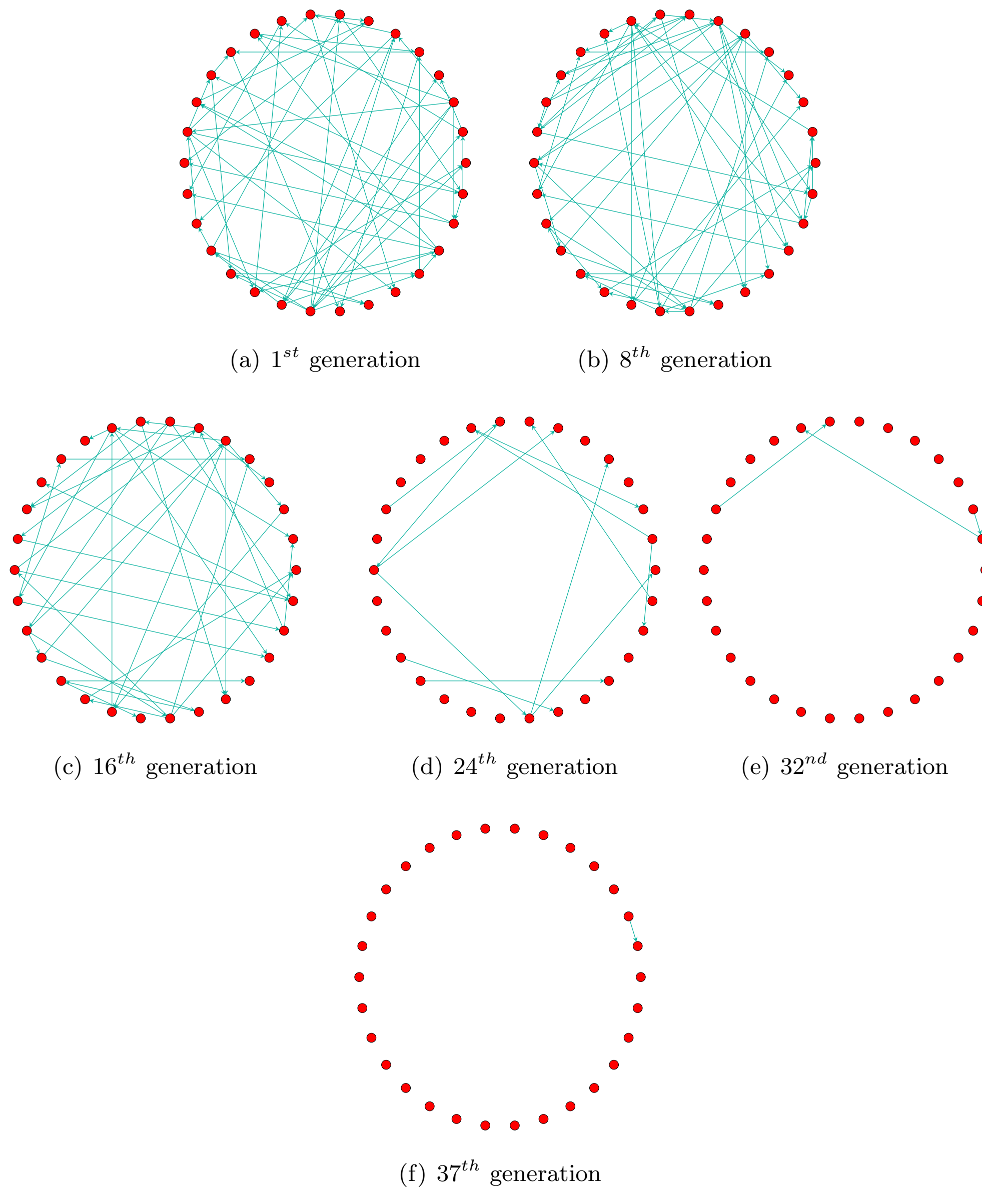
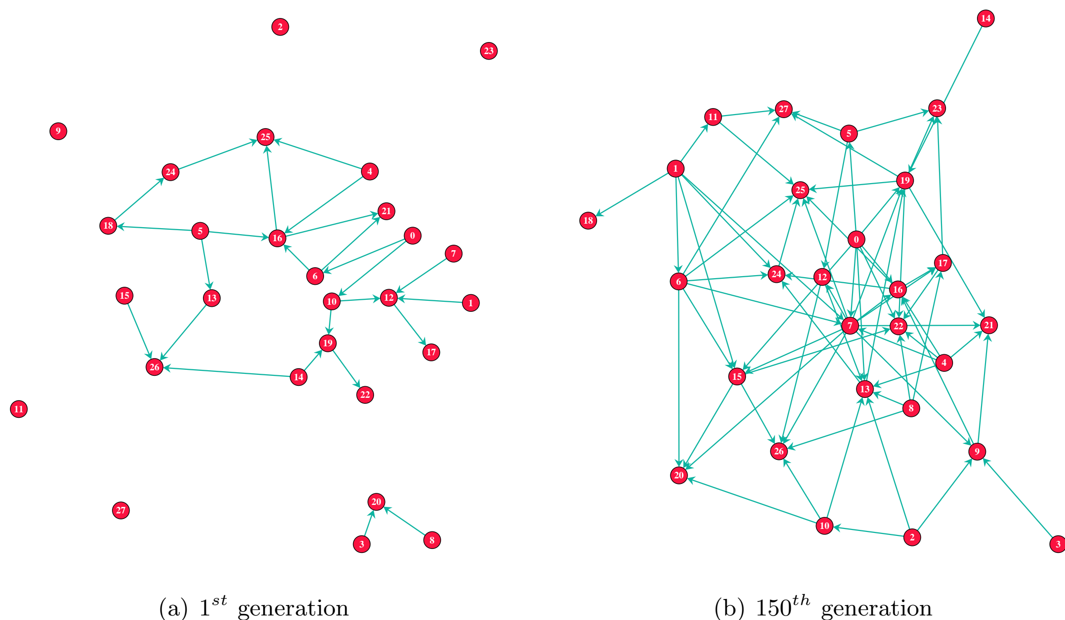
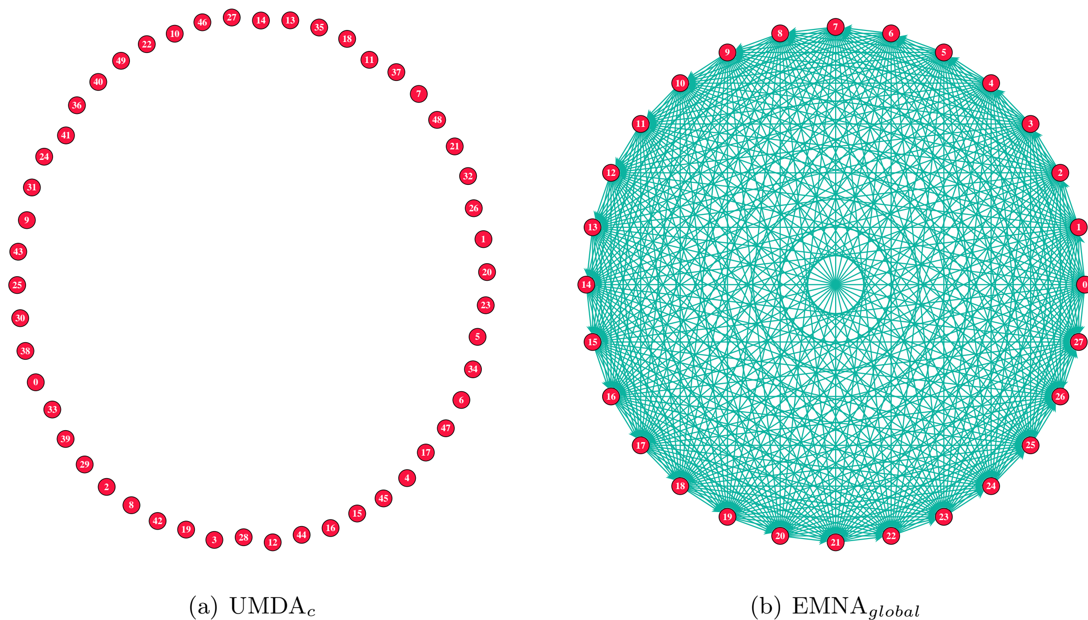


Figure 6.11: Graphs showing the evolution of the Bayesian network in a EBNA search, illustrating clearly that the number of arcs of the probabilistic structure decreases gradually from the first generation to the last ones. A circular layout has been chosen in order to show the same nodes in the same position. The number of arcs decreases as follows respectively: 57, 56, 40, 12, 3, and 1. After the 37th generation and until the last (the 43th one) the Bayesian network does not contain any arc.



(a) 1st generation (b) 150th generation

Figure 6.12: Figures showing the structures learned during the Edge Exclusion EGNA$_{ee}$ continuous EDA.

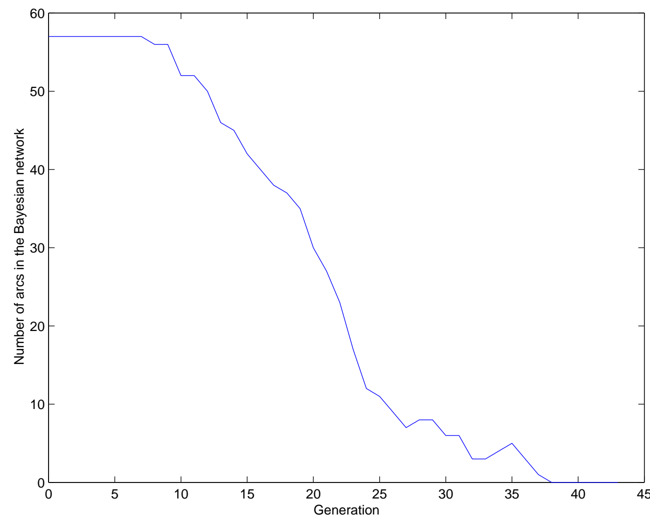


(a) UMDA$_c$ (b) EMNA$_{global}$

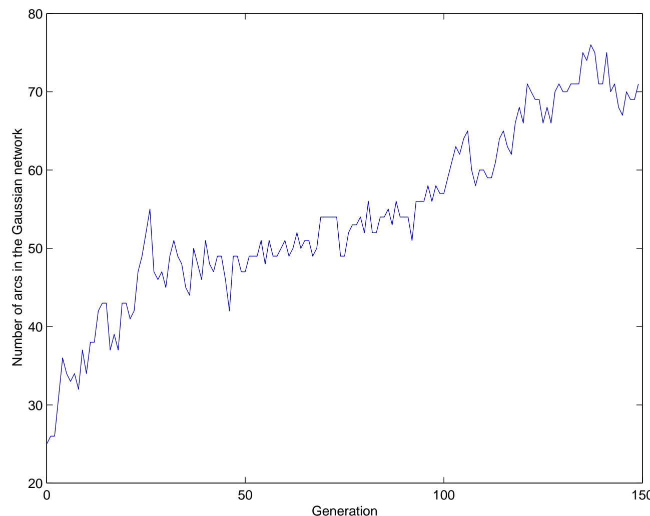
Figure 6.13: Figures showing the structures learned during the UMDA$_c$ and EMNA$_{global}$ continuous EDAs.

- The fact that using a representation with that many values per variable requires a bigger population per generation so that more complex dependencies can be analyzed.

It is important to note that in the former case, even if the best results obtained are similar, the execution time for EBNA would be much higher. This is caused by the first part of



(a) EBNA



(b) EGNA$_{ee}$

Figure 6.14: Graphs showing the evolution in number of arcs of the respective probabilistic structures for EBNA and EGNA$_{ee}$. The two different tendencies are illustrated: EBNA tends to a structure with less arcs when the search goes on, while EGNA-type algorithms tend to a structure with more arcs.

the learning process of EBNA that requires searching for the best structure to model the probability distribution, as in UMDA there is no such a step.

6.4 Study 3: parallelization

This study concentrates on the parallelization issues concerning EDA algorithms. Following the techniques and explanations given in Chapter 5, this section concentrates in the implementation and in obtaining experimental results to show the behavior of the different communication methods available in communication for parallel programs: the use of shared

memory and threads, and the use of message passing and processes in different machines. This section is subdivided in two parts regarding both paradigms.

6.4.1 Parallelizing EBNA using threads and shared memory

Taking into account all the properties of the BIC score described in Section 4.3.4 and analyzed in Section 5.5.3, we perform a basic modification on the sequential program in order to allow to compute in parallel all the arc modifications. For this task, the broadly known *pthreads* standard library is used, which allows threads to communicate through memory and to synchronize them using semaphores.

The score+search procedure is therefore organized in a very different way, as now threads have to divide the work: a thread plays the role of the *manager* that distributes the work among the rest, and the others are the *workers* that have to compute all the possible arc modifications for a same number of nodes.

Reorganizing the execution of the program

The fact of using a multithread program allows us to use shared memory for communication between them. This is implemented by creating a single process which contains many threads executing within its memory space. Global variables are defined, and these are used for direct communication between threads. However, the use of shared variables for communication leads to the existence of race conditions within the program. Therefore, a synchronization mechanism is required to ensure exclusive access to the critical sections in the program. The critical sections have to be identified by the programmer, and it is also his responsibility to integrate the synchronization primitives within the code.

In order to accomplish the parallelization task, the first thing to do is to choose a multithread standard library to program. The *pthreads* library is commonly available in many operating systems, so we decided to select it.

Once having decided this, the next step is to select the working scheme that all the parallel program will use for organizing the work of all the threads. EDA programs do have several parts that can only be executed sequentially, and the only part that we intend to parallelize is the computation of the BIC score each generation. Therefore the use of a master-slave scheme is very suitable for this case: a first thread will be executing the sequential parts of the EDA program, and when it reaches the step of estimating the probability distribution it will divide the work and send it to the workers. The worker threads will compute the BIC score.

An important aspect to consider is the number of worker-threads that will be created at the same time. We could think at a first glance that creating as many threads as possible is the best to finish the job, but it is important to take into account that creating a thread also has a cost associated, and that each thread has to be given enough work in order to justify its generation time. In addition, another limiting factor to decide how many simultaneous threads can be working is the number of CPUs of our system: having too many workers will lead to a system with workers competing between them to take ownership of CPUs instead of having them cooperating. For this reason, a semaphore is used to limit the number of threads created at any time. This semaphore is initialized to the maximum number of threads that can exist. In our case we have a two processor computer, and therefore this limit was set to 4 threads[3].

[3] In fact, since in our particular case we have only two processors in our computer, we could limit the

EDA Algorithm	10 & 30 ex. sequential exec. time	10 & 30 ex. parallel exec. time	50 & 250 ex. sequential exec. time	50 & 250 ex. parallel exec. time
$EBNA_{BIC}$	00:04:34	00:03:20	52:59:49	28:00:04
$EGNA_{BGe}$	00:13:23	00:16:35	461:59:00	67:48:01
$EGNA_{BIC}$	02:29:41	01:59:31	(*)	(*)

Table 6.10: Execution time for the 10 & 30 and 50 & 250 examples using the $EBNA_{BIC}$, $EGNA_{BGe}$ and $EGNA_{BIC}$ algorithms regarding their sequential and parallel versions of computing the BIC score (hh:mm:ss). The values with the symbol (*) required more than a month of execution time to be properly computed.

Experimental results of the multithread BIC procedure using shared memory

As the parallel BIC algorithm does not follow a new algorithm, the best fitness values obtained with the new parallel version of the EBNA approach are exactly the same as the sequential version. The only differences that can be expected are just in the execution time. Table 6.10 shows the effect of applying the parallel algorithm on the 50 & 250 example in both the $EBNA_{BIC}$ and $EGNA_{BIC}$ algorithms for their sequential and parallel versions.

The results show clearly that the use of threads reduces considerably the execution time for the 30 & 100 and 50 & 250 examples. A special mention is for the 10 & 30 example in the $EGNA_{BGe}$ case, as these particular results show that the parallel version requires longer time than the sequential program. This is the result of parallelizing the learning step: in Table 5.2 we can see clearly that the percentage of computing time for such an small example is of 5.6 %, while in the 50 & 250 example we obtain a computation percentage of 55 %. This illustrates that parallel programming techniques can improve the overall execution time of the program, but that the cost of creating new processes or threads has also to be taken into account, as already explained in the previous sections. In the small example with the $EGNA_{BGe}$ algorithm the relative weight of the function is so small in the whole execution time that each of the worker-threads do not have enough processing tasks in order to justify their creation, while in the big example the learning step is the one requiring the most computation time and the fact of being computed in different threads gives very satisfactory results.

For the rest of the cases and EDAs the final result was a considerable improvement in execution time in both the small and big examples, and the application of parallel programming techniques such as the use of the library *pthreads* appears to be very advisable.

In brief, the results obtained could be summarized as follows: multithread libraries applied on multiprocessors are a very powerful tool for programs such as EDAs that require a big amount of CPU time, but it is necessary to perform previously an analysis on the relative time consumed by each of its functions in order to be sure to be applying them correctly.

6.4.2 Parallelizing EBNA using processes and MPI

MPI is a message passing interface introduced in Section 5.3.3. As its name indicates, MPI has been designed to use message passing as the communication mechanism for inter-process

number of threads to 2. However, any program has always some small periods of time where a thread is blocked waiting for a operating system job, and another thread could use the CPU in the meanwhile. This is the reason why it is convenient to create some more threads than processors.

communication. Threads that are attached to the same process usually communicate through global variables in shared memory. However, any two threads from two different processes (either if the processes are on the same computer or connected through a network) cannot use shared memory and therefore message passing is the only possible mechanism. MPI provides an efficient mechanism for threads from different processes, and it can also been applied even when shared memory is available.

MPI has been designed as an interface for communicating processes that could even be executing in different computers at the same time. We have chosen the MPI implementation called MPICH for our experiments. The reasons for choosing both MPI and this particular implementation MPICH is their portability, good performance and availability for operating systems such as Windows, Linux and Solaris. MPICH even contains versions for very fast computer network configurations such as Myrinet, which allows us for further reduction in execution times.

As in MPI the communication is based on message passing primitives, the synchronization between sender and receiver is done implicitly. In addition, using MPI allows us to use a cluster formed by many 2-processor simple architecture PCs under Linux connected by a very fast local area network (LAN) to collaborate and cooperate each other by creating processes in all the machines without having to change the program. This means that in this case we can use the same master-slave working scheme as with the parallel version of the *pthreads* library, but this time workers would be processes that could be executing in different CPUs and even in different computers at the same time.

However, in the particular example of the EDA program, the fact that processes cannot share any memory among them forces the manager to send all the data structures required to compute the BIC function to each of the workers. Afterwards when all the workers have completed their part of the job, each one will have to send a message to the master with the amount of work done. In addition, as in MPI processes are created and not threads, it is important to take into account that the creation of a process requires more time that creating a thread. Moreover, if processes are to be created in other computers within a cluster this operation will take even longer.

All the latter consideration make us modify partly the structure of the EDA program. In addition, the fact that in MPI the master and the slaves have to execute the same program and that processes cannot be created dynamically is also another reason for a deep restructure of the whole sequential program.

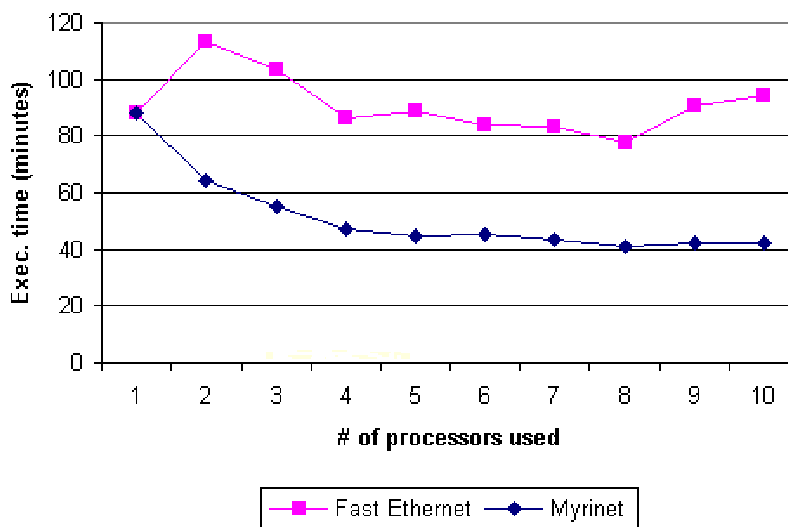
Experimental results of the parallel BIC procedure using MPI

The implementation of the EBNA_{BIC} parallel version using MPI was tested on a different machine than the one used in the multithread program. The reason for this is that in this case for an efficient use of MPI a cluster of workstations is more suitable rather than a single machine with fast processors. In our case, we tested the parallel program based on MPI using a cluster formed by 5 computers with 2 Intel Pentium II processors at 350 MHz, 512 KBytes cache, and 128 MBytes of RAM each. The operating system used is GNU-Linux. These computers are connected by two different local area networks: one is a Fast Ethernet and Myrinet. The different between them is that Myrinet has a bigger bandwidth, provides shorter latency times for communication, but it is much more expensive. The decision of using a network or another does not imply any modification on the source code, and it does only affect the compilation options of the MPI distribution. Execution times with both types of local area networks (LANs) were tested and are presented in this section.

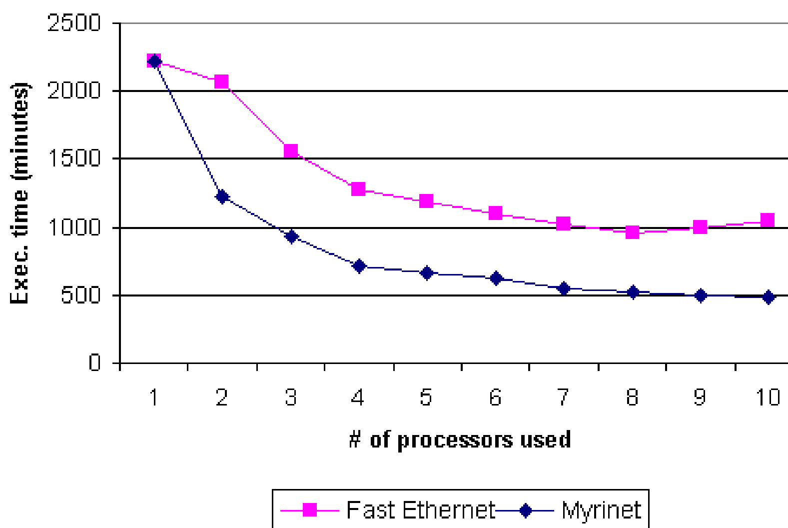
Different experiments were performed using both the 30 & 100 and 50 & 250 examples of Study 1. Both networks were tested using different number of processors (i.e. number of worker-processes created). Figure 6.15 shows the evolution of the execution time for the two examples and in the two networks. The case of the 30 & 100 example with Fast Ethernet illustrated in Figure 6.15a is specially illustrative, as it shows the typical case in which in two moments the fact of increasing the number of workers also increases the execution time required. The explanation of this is as follows: the first and initial increase in execution time is due to the added computation required to implement the communication between workers and the manager, while the latter at the end is a result of dividing too much the work load per worker and therefore requiring more time to create each of them rather than having them working. In between both moments, we have a part in which the parallelization techniques give the expected results of reducing the execution time. In this example but for the Myrinet network this does not happen because the network is faster and therefore the overhead created by the communication is also smaller.

The 50 & 250 example presents acceptable figures for both the Fast Ethernet and Myrinet networks. Both graphics in Figure 6.15 show also the differences in data transfer speed between the Fast Ethernet and Myrinet networks. On the other hand, with smaller examples the increase in execution time could also happen even when using a fast network such as Myrinet. These results are also shown in numeric format in Table 6.11. Therefore, our experiments show the expected effects when parallelizing complex and CPU intensive programs, and these also serve as an idea of the minimum size that the problems need to have in order to obtain shorter execution times. This table also shows that in both the Fast Ethernet and Myrinet networks, the optimum number of workers is 7 (1 manager process + 7 worker processes), and for the Myrinet case the higher the number of processes the shorter the execution time (we arrived until a total of 9 worker processes), although in the Myrinet case we would find a moment after which the execution time would also show the increase when augmenting too much the number of workers. However, these results are only valid for the particular configuration of the cluster where the experiments were carried out, and will have considerable differences depending on the RAM memory available, and the number and working frequency of the processes in each computer that forms the cluster.

Figure 6.16 is an example of the types of traces that we can obtain using the MPICH implementation of MPI. These traces are used to identify bottlenecks in MPI programs, but here are presented for illustrating the communication requirements that the evolution of each generation requires. The state and MPI primitive executed by each of the processes are shown using different colors. These figures demonstrate that while the manager is executing the rest of the workers are simply waiting, and therefore no parallelization is occurring during this time. Later, there is a phase in which the manager is executing *sending* operations and workers are mostly executing receiving ones. The latter is the time in which the work is being distributed to and executed by the workers while the manager is administrating and coordinating all the job. This repeats once and again until the total number of generations have been completed. Obviously, in order to achieve a considerable reduction in execution time the parallel phase needs to occupy most of the time in the diagram, which has been shown to happen in our case when having graphs of big sizes. The speed of the network is also another factor to consider, and this fact is shown in the different times illustrated in the figure.



(a) 30 & 100 example



(b) 50 & 250 example

Figure 6.15: Illustration of the evolution in execution time when using MPI and depending on the number of processes.

6.5 Conclusions of the studies on synthetic problems

Very different studies have been described in this chapter, from which different results and conclusions for the application of these techniques on real graph matching problems. We can summarize them as follows:

Study 1: the main conclusion that is obtained from this study is that when additional constraints are present in real problems a mechanism is available in EDAs to ensure that the final solution will satisfy all of them. Specially in graph matching problems that have a complexity similar in size to the ones obtained in real images, the control of

Number of Processes	Total Time Fast Eth.	Speed Up Fast Eth.	Total Time Myrinet	Speed Up Myrinet
1	01:28:07	1	01:28:07	1
2	01:53:03	0.779	01:04:28	1.368
3	01:43:40	1.850	00:54:59	1.602
4	01:26:04	1.023	00:47:19	1.862
5	01:28:55	0.991	00:44:47	1.967
6	01:23:47	1.052	00:45:13	1.949
7	01:23:02	1.061	00:43:33	2.023
8	01:17:57	1.130	00:41:02	2.147
9	01:30:22	0.975	00:42:18	2.083
10	01:34:20	0.934	00:42:02	2.096

(a) 30 & 100 vertices

Number of Processes	Total Time Fast Eth.	Speed Up Fast Eth.	Total Time Myrinet	Speed Up Myrinet
1	37:00:09	1	37.00:09	1
2	34:24:55	1,075	20:21:01	1.818
3	25:54:22	1.428	15:29:36	2.388
4	21:16:22	1.739	11:50:02	3.126
5	19:50:30	1.864	11:06:42	3.330
6	18:15:10	2.027	10:20:06	3.580
7	16:59:37	2.177	09:10:19	4.034
8	15:52:58	2.329	08:42:15	4.251
9	16:30:14	2.242	08:21:53	4.423
10	17:22:47	2.129	08:10:07	4.529

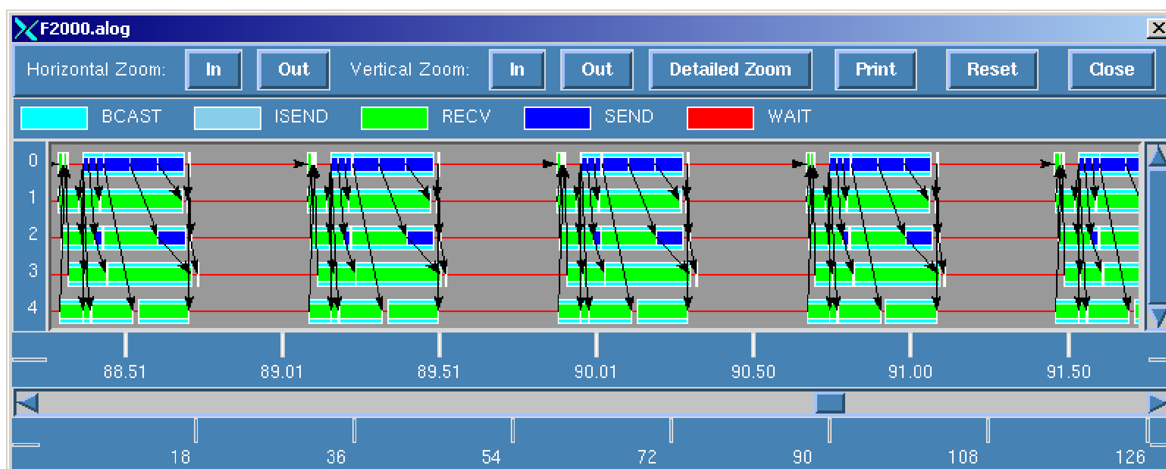
(b) 50 & 250 vertices

Table 6.11: Execution times obtained when increasing the number of processes (processors used) for the medium-sized example with 30 & 100 vertices (above) and for the big example with 50 & 250 vertices (below). Times are presented in *hh:mm:ss* format.

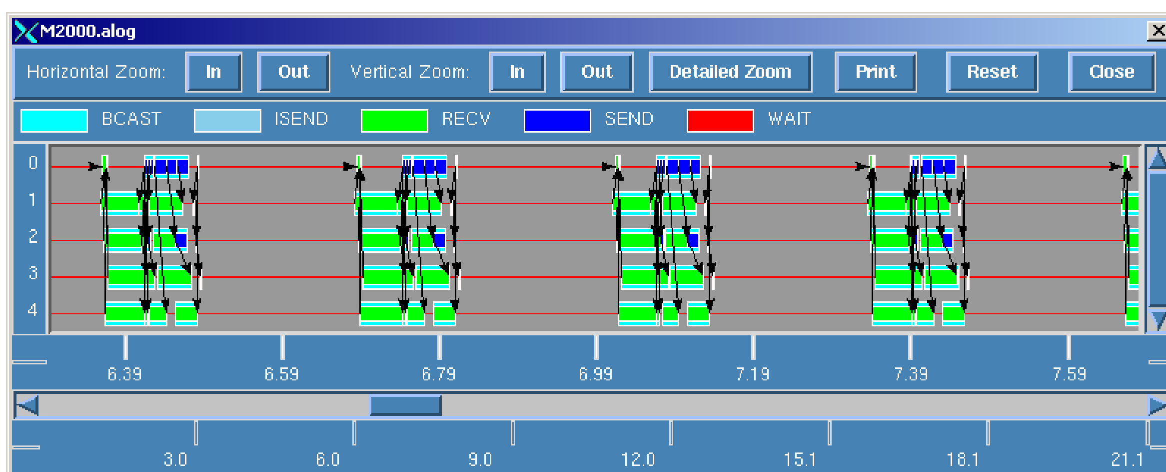
constraints can be performed applying to EDAs the methods described in Section 4.5.1.

In addition, this study presents a comparison of the performance of the different EDAs and other evolutionary computation algorithms which illustrate their differences depending on the complexity of the graph matching problem. In addition, the same graphics show the evolution in best solution obtained per generation, which can be used to select an algorithm over the rest regarding the stopping criterion of the search. In any case, EDAs show on the whole a better performance than other evolutionary computation techniques for both the discrete and continuous domains.

It is important to note that the behavior of the algorithms is always very dependent on the fitness function selected. Fitness functions are defined for each particular problem, and algorithms also adapt in a different way to the type of fitness functions. From the experiments carried out we can see that the different algorithms have different tendency to fall on local maxima (or local minima if the fitness function has to be minimized)



(a) Trace of execution in a Fast Ethernet network



(b) Trace of execution in a Myrinet network

Figure 6.16: Figures for the communication mechanisms and other MPI primitives on the parallel version of EBNA$_{BIC}$ for the particular configuration of our cluster.

or to scape from them. In this study, EDAs showed a better propensity to avoid local maxima, although in fitness function with less local maxima GAs obtain the optimum in a shorter time.

Study 2: this study illustrates the behavior of the different EDAs during the search process. This behavior is specially interesting in case of EBNA and EGNA, as they take into account all the possible dependencies. This study illustrates the evolution of the probabilistic graphical models on EDAs and explains why they converge to solution at the end of the search process.

Study 3: from the results presented in Study 3, we can conclude that the mechanism used in this thesis reduces considerably the execution time required for CPU intensive programs such as EBNA$_{BIC}$. The two versions of the parallel program, the multithread and the MPI ones, can be applied to multiprocessors and clusters respectively according to the hardware availability. In addition, results of two types of LANs are presented showing

the relevance of the network speed on the execution time.

In both cases, the paradigm applied is a master-slave scheme as described in Section 5.6. This mechanism has been designed for its easy adaptation to other EDAs or evolutionary computation algorithms. The source code of the multithread and MPI version including detailed explanations about the communication and synchronization implementations is presented in Appendix D.

However, another important conclusion that is obtained from these results is that the number of workers is an important parameter that is very dependent on the characteristics of the computers that has to be chosen with care in order to obtain a satisfactory reduction in execution time without increasing the overhead of the communication between the workers and the master.