Incremental Genetic Algorithm

Nashat Mansour[1], Mohamad Awad[1], and Khaled El-Fakih[2]
[1]Computer Science Division, Lebanese American University, Lebanon
[2]Department of Computer Science, American University of Sharjah, UAE

Abstract: *Classical Genetic Algorithms (CGA) are known to find good sub-optimal solutions for complex and intractable optimization problems. In many cases, problems undergo frequent minor modifications, each producing a new problem version. If these problems are not small in size, it becomes costly to use a genetic algorithm to reoptimize them after each modification. In this paper, we propose an Incremental Genetic Algorithm (IGA) to reduce the time needed to reoptimize modified problems. The idea of IGA is simple and leads to useful results. IGA is similar to CGA except that it starts with an initial population that contains chromosomes saved from the CGA run for the initial problem version (prior to modifying it). These chromosomes are best feasible and best infeasible chromosomes to which we apply two techniques in order to ensure sufficient diversity within them. To validate the proposed approach, we consider three problems: Optimal regression software testing, general optimization, and exam scheduling. The empirical results obtained by applying IGA to the three optimization problems show that IGA requires a smaller number of generations than those of a CGA to find a solution. In addition, the quality of the solutions produced by IGA is comparable to those of CGA.*

Keywords: *Exam scheduling, general optimization, incremental genetic algorithms, optimization algorithms, regression testing, soft computing.*

Received September 28, 2004; accepted December 21, 2004

1. Introduction

Genetic algorithms are based on the mechanics of natural evolution [4, 7]. They mimic natural populations reproduction and selection operations to achieve efficient and robust optimization. Through their artificial evolution, successive generations search for beneficial adaptations in order to solve a problem. Each generation consists of a population of chromosomes, also called individuals, and each chromosome represents a possible solution to the problem. The initial generation consists of randomly created individuals. Each individual acquires a fitness level, which is usually based on a cost function given by the problem under consideration.

Reproduction, survival of the fittest principle, and the genetic operations of recombination (crossover) and mutation are used to create new offspring population from the current population. The reproduction operation involves selecting, in proportion to fitness, a chromosome from the current population of chromosomes, and allowing it to survive by copying it into the new population. Then, two mates are randomly selected from this population, and crossover and mutation are carried out to create two new offspring chromosomes. Crossover involves swapping two randomly located sub-chromosomes (within the same boundaries) of the two mating chromosomes. Mutation is applied to randomly selected genes, where the values associated with such genes are randomly changed to other values within an allowed range. The offspring population replaces the parent population, and the process is repeated for many generations with the aim of maximizing the fitness of the individuals. In this paper we refer to genetic algorithms that start with randomly- generated initial population as Classical Genetic Algorithms (CGA). An outline of CGA is given in Figure1.

```
Random generation of initial population, size POP;
Evaluate fitness of individuals;
Repeat
    Rank individuals and allocate reproduction trials;
    For (I = 1 to POP step 2) do
        Randomly select 2 parents from the list of
         reproduction trials
        Apply crossover and mutation;
    Endfor
    Evaluate fitness of offspring;
Until (convergence criterion is satisfied)
Solution = Fittest
```

Figure 1. Classical genetic algorithm.

CGAs have been adapted for solving a variety of engineering, science, economics, and operational research problems. Some examples of such applications can be found in [1, 2, 6, 8, 9, 12, 13, 14, 15]. Usually, a CGA starts with random chromosomes that make up the initial generation. Then, it evolves until it converges to one (best) solution. A CGA can be hybridized and augmented with a variety of techniques to improve its efficiency, to ensure feasibility of final solution, etc… Examples of such techniques can be found in [3, 5]. But, all these CGAs are 'ab-initio' algorithms. That is, if they are to re-solve a problem that has undergone a small modification, they start again with a random population of chromosomes. Ab-initio runs take a comparable number of generations (and time) to the first run of the CGA on the initial version of the problem prior to its modification. For example, the modification can be due to a change in a parameter or in a feasibility constraint. If the problem size is not small, it may be too costly to spend the same evolution time for every modification. This would be particularly costly if such modifications are frequent and for large problem sizes.

In this paper, we propose a different approach to re-solving modified problems by genetic algorithms. Our approach is based on saving chromosomes from the first CGA run (for the initial problem). The chromosomes to be saved are the best feasible chromosomes and the best infeasible ones. We apply two techniques in order to ensure diversity among these saved chromosomes. Then, we construct the initial population of the genetic algorithm from these chromosomes, in addition to random ones. The underlying assumption is that a solution to a modified problem might lie close to that of the initial version, but it may be on either side of the border of the feasibility region. Thus, starting with the saved chromosomes ensures faster convergence to a new solution. Obviously, the random portion of the population is included in order to increase diversity. We refer to genetic algorithms that are based on this approach as Incremental Genetic Algorithms (IGA), since they deal with incremental changes.

We empirically explore the IGA approach by comparing it with CGAs for re-solving modified optimization problems. The subject problems used are: Optimal regression testing, general optimization, and exam scheduling. The empirical results show that the IGA idea is simple and yet leads to useful results. For the three problems, IGA takes a smaller number of generations (and less execution time) than CGA. Yet, it yields comparable or slightly better solution quality.

This paper is organized as follows. Section 2 describes the IGA. Section 3 presents the empirical results. Section 4 contains our conclusion.

2. Incremental Genetic Algorithm

The idea of an IGA for optimizing modified problems is simple. Instead of starting with randomly generated population of chromosomes, start by using information saved from running an ab-initio CGA on the initial problem (before modification). The underlying assumptions of the IGA idea are:

1. A modification made to a problem does not shift optimal and good sub-optimal solution points much in the solution space.
2. The information saved during the application of a CGA for the initial problem will be useful for subsequent application of a genetic algorithm to modified versions of this problem.

The aim is to reduce evolution time, measured in number of generations, of a genetic algorithm used for re-optimizing modified problems. This is particularly useful for complex and large-scale optimization problems, which take many generations and long execution times. This approach leads to faster solutions for incrementally modified problem. Thus, we refer to genetic algorithms based on this approach as incremental genetic algorithms.

IGA is based on two phases. In phase 1, we collect useful information during the execution of a CGA on the initial version of the problem. Useful information consists of best feasible and best infeasible chromosomes in every generation of CGA. To ensure diversity in these chromosomes, we use two techniques: Duplication- prevention and FCL-enriching. In phase 2, we run IGA starting with chromosomes selected from the useful information saved, in addition to randomly generated chromosomes. Figure 2 shows the steps involved in IGA, which are described in the following subsections.

```
Create empty best feasible chromosome list (FCL);
Create empty best infeasible chromosome list (ICL);
gen_counter = 0;

Phase 1: In every generation of CGA (run for the initial problem):
        Determine best feasible chromosome and apply duplication-
        prevention with FCL
            If no duplicates found then
               add it to FCL;
               reset gen_counter;
            Else increment gen_counter;
        Determine best infeasible chromosome and apply duplication-
        prevention with ICL
            If no duplicate found then add it to ICL;
            If gen_counter = 3 then
               apply the FCL-enriching technique and add selected
               chromosomes to FCL;
               reset gen_counter;

Phase 2:  IGA (run for the modified problem):
         Sort FCL and ICL according to fitness;
         Run IGA starting with initial population which is typically
          composed of:
                    50 %  best  feasible  from FCL
                    25 % best infeasible from ICL
                    25 % randomly generated
```

Figure 2. Incremental genetic algorithm.

2.1. Saving Chromosomes in CGA

In phase 1, we save chromosomes from every generation of CGA. The candidate chromosomes are normally the best feasible and the best infeasible, which have the highest fitness in their feasible and infeasible regions, respectively. We apply diversity-assuring techniques, described next, to the candidate chromosomes and the selected ones are added to the feasible and infeasible chromosome lists, FCL and ICL, respectively. For diversity purposes, the candidate chromosomes may be those other than the best one.

This is also explained next in the FCL-enriching technique.

2.2. Duplication-Prevention Technique

To prevent duplication of chromosomes in the FCL and ICL lists, we find the Hamming distance between a candidate chromosome and all chromosomes in the relevant list. The number of bits by which the two chromosomes differ gives the Hamming distance. If the Hamming distance is different from zero, the candidate chromosome is added to the list. Otherwise, the chromosome is not added. For best feasible chromosomes, if they fail to be added to the FCL for 3 generations, we apply the FCL-enriching technique, which aims to ensure that FCL contains sufficient and diverse chromosomes for IGA. A similar technique to FCL-enriching is not applied to infeasible chromosomes, since any shortage in infeasible chromosomes can be substituted in IGA with randomly-generated ones without losing useful/feasible information.

2.3. FCL-Enriching Technique

The best feasible chromosome may fail to pass the duplication-prevention test in FCL. This means that FCL may end up with a small, insufficient number of chromosomes at the end of a CGA run. Equally important is that the chromosomes to be added to FCL be as diverse as possible. The FCL-enriching technique aims to fulfill the diversity objective while enriching FCL with useful feasible chromosomes. It is applied only if the duplication-prevention test fails 3 successive times.

The FCL-enriching procedure is composed of the following steps:

1. Compute f_i, the fitness value of every feasible chromosome i in the current generation.
2. Compute d_i, the Hamming distance from the local feasible chromosome i to the best-so-far chromosome in FCL.
3. Compute the average of both the fitness values and Hamming distances of the feasible chromosomes, av_f and av_d.
4. Add to FCL feasible chromosomes i that satisfy the following three conditions:
    1. Passes the duplication-prevention test in FCL.
    2. $f_i > av_f$ (i. e., i has high fitness).
    3. $d_i < av_d$ (i. e., i is close to best-so-far chromosome).

2.4. Constructing the Initial Population for IGA

At the end of a CGA run on the initial problem version, the FCL and ICL lists are constructed. These lists are then sorted by fitness value. A typical initial population of IGA, to be run for a modified version of the problem, is constructed as follows: 50% of the population size is taken from the best FCL chromosomes, 25% from the best ICL chromosomes, and the remaining 25% are generated randomly. In cases where these typical percentages from FCL and ICL cannot be achieved, the random component is increased to compensate for the shortage. Obviously, these typical percentages are empirically determined. They emphasize the contribution of the feasible chromosomes without undermining the possibility that the solution of the modified problem might lie near 'good' infeasible solution points. Further, a random component is kept in the initial population in order to increase the level of diversity.

3. Empirical Comparison of IGA and CGA

In order to support our claim about the performance of IGA, we compare it with CGA for three optimization problems from software engineering and operations research: Optimal regression software testing, general optimization, and exam scheduling. For each problem, we run CGA and incorporate into it the Phase 1 steps necessary for IGA. Then, we introduce small modifications to the initial formulation of the problem. We compare IGA and CGA by running both for re-solving the modified problem and counting the number of generations each algorithm take to converge to a solution. The quality of the solutions obtained by IGA and CGA are also recorded to ensure that a reduction in the number of generations of IGA is not accompanied by deterioration in the solution quality.

3.1. Results for Optimal Regression Software Testing

Optimal regression software testing aims to select minimum number of tests from an initial suite of N tests such that the paths affected by fixing a program segment are covered. We assume the program is made up of M segments. Given that program segment k has been modified, the optimal retesting problem consists of finding values for ($X_1, X_2, \dots, X_N$) that minimize the cost function:

$$Z = X_1 + X_2 + \dots\dots + X_N$$

Subjectto the constraints :

$$\sum_{j=1}^{N} a_{ij} X_j \geq b_i;\ i = 1, .., M$$

Where X_j = 1 (or 0) indicates the inclusion (or exclusion) of test case j in the selected subset of retests. The matrix $[a_{ij}]$ is derived directly from the test-segment coverage table, i. e., a_{ij} = 1 if segment i covered by test case j; b_i = 1 (or 0) indicates whether segment i needs to (or need not) be covered by the subset of retests due to the modification of segment k, where the values b_i are derived from the segment reachability information [11].

Tables 1-4 show the number of generations and the objective function value of the final solution for CGA and IGA. The different tables show the results for different problem sizes and for different modifications made to the initial problem. The modifications are made to a small number of constraints, specifically to the values on the right-hand side of the inequalities. This corresponds to a change in the number of the affected paths that are required to be covered by the selected tests.

The results in Tables 1-4 show that:

1. IGA evolves a solution faster (in number of generations) than CGA in most cases; this advantage is clearer for larger problem sizes.
2. The solution quality of IGA is comparable to that of CGA, even where the number of generations is significantly less (see Table 4).

Further, to give an idea about the actual saving in execution time, we note that for the 6000x6000 problem with 256 changes, CGA takes 1hour on a 1GHz-Pentium based PC, whereas IGA takes 0.65 hour.

3.2. Results for General Optimization

In general optimization, we minimize:

$$\sum_{i=1}^{N} (1)^{i-1} X_i$$

Subject to the constraints:

$$\sum_{j=1}^{N} a_{ij} X_j \geq b_i;\ i = 1, .., M$$

Where b_i varies between 0 and 20, or it could be only 0 and 1.

Two optimization problems are used. In the first problem, the variables can take values in {–1, 0, 1}, and the right-hand values of the constraints range between 0 and 20. In the second problem, the variables can be {–1, 0, 1}, and the right-hand side values can be {0, 1}. The changes for this problem are made to the values on the right-hand side of the constraints/inequalities.

Table 1. Results for regression testing, MxN = 1000x1000.

	CGA		IGA	
	Number of Generations	Objective Function	Number of Generations	Objective Function
1000x1000 16 Changes	10	19	10	18
1000x1000 32 Changes	17	18	10	18

Tables 5-8 show the results for different problem sizes and number of changes. The advantage of IGA is remarkable over CGA in terms of the number of generations required to find a solution. Again, the solution quality of IGA is similar to that of CGA.

Table 2. Results for regression testing, MxN = 2000x2000.

	CGA		IGA	
	Number of Generations	Objective Function	Number of Generations	Objective Function
2000x2000 16 Changes	17	20	10	20
2000x2000 64 Changes	10	21	10	20

Table 3. Results for regression testing, MxN = 4000x4000.

	CGA		IGA	
	Number of Generations	Objective Function	Number of Generations	Objective Function
4000x4000 128 Changes	20	16	20	16
4000x4000 256 Changes	23	16	20	16

Table 4. Results for regression testing, MxN = 6000x6000.

	CGA		IGA	
	Number of Generations	Objective Function	Number of Generations	Objective Function
6000x6000 256 Changes	26	45	16	45
6000x6000 512 Changes	35	44	10	45

Table 5. Results for general optimization, variables {-1, 0, 1} and constraints {0, 1}.

	CGA		IGA	
	Number of Generations	Objective Function	Number of Generations	Objective Function
1000x1000 16 Changes	36	100	15	99
1000x1000 32 Changes	43	111	32	111

Table 6. Results for general optimization, variables {-1, 0, 1} and constraints 0..20.

	CGA		IGA	
	Number of Generations	Objective Function	Number of Generations	Objective Function
1000x1000 16 Changes	37	195	10	194
1000x1000 32 Changes	29	199	20	189

Table 7. Results for general optimization, variables {-1, 0, 1} and constraints {0, 1}.

	CGA		IGA	
	Number of Generations	Objective Function	Number of Generations	Objective Function
4000x4000 128 Changes	23	163	10	151
4000x4000 192 Changes	56	150	10	151

Table 8. Results for general optimization, variables {-1, 0, 1} and constraints 0..20.

	CGA		IGA	
	Number of Generations	Objective Function	Number of Generations	Objective Function
4000x4000 128 Changes	31	449	10	440
4000x4000 192 Changes	31	459	10	440

3.3. Results for Exam Scheduling

The exam scheduling problem is a complex optimization problem. It refers to assigning exams to periods so that the following quantities are minimized: The number of students with simultaneous exams (S_{SE}), the number of students with consecutive exams (S_{CE}), and the number of students having multiple exams on the same day (S_{ME}). In addition, we have constraints, such as the total number of exam periods, total number of available rooms with predetermined capacities, etc… The objective function is given by a weighted sum of S_{SE}, S_{CE}, S_{ME}, and the number of room violations [10]. We use three instances of the exam scheduling problem based on real data for the semesters S95, F96, F98. The three instances differ in the number of exams and students.

Table 9 shows the results for the three semesters. For S95, the modification to the problem instance is made by forcing the exams of the sections of the same course to be scheduled to the same period. For F96 and F98, the modification is made by deleting some courses (i. e., reducing the number of variables). Table 10 shows the results for S95, where the modification is made to the maximum number of exam periods allowed. Figure 3 illustrates the results of Table 10 graphically. All these results clearly show that IGA yields a solution in a smaller number of generations than CGA and that its solutions are comparable or somewhat better. To give an idea about the reduction in execution time, we note that in Table 10, periods = 40, CGA takes 1.1 hours, whereas IGA takes 0.55 hours (50 % reduction).

Table 9. Results for exam scheduling, 3 semesters, 32 periods.

		S_{SE}	S_{CE}	S_{ME}	No. of Rooms Used	Violation of Room Capacity	Objective Function	No. of Generations
S95	CGA	0	344	810	21	0	1154	84
	IGA	0	345	802	21	0	1147	62
F96	CGA	0	268	600	21	0	868	43
	IGA	0	224	494	21	0	718	32
F98	CGA	0	41	197	21	0	238	59
	IGA	0	45	185	21	0	230	33

Table 10. Results for exam scheduling for S95 with different periods.

CGA	Periods = 24	Periods = 28	Periods = 36	Periods = 40
S_{SE}	2	0	0	0
S_{CE}	879	572	231	95
S_{ME}	1680	1060	647	422
Room Violation	0	0	0	0
Objective Function	2759	1632	878	517
Generations	60	89	72	77
IGA	**Periods = 24**	**Periods = 28**	**Periods = 36**	**Periods = 40**
S_{SE}	2	0	0	0
S_{CE}	804	455	180	91
S_{ME}	1477	1109	527	423
Room Violation	0	0	0	0
Objective Function	2481	1564	707	514
Generations	40	40	63	34

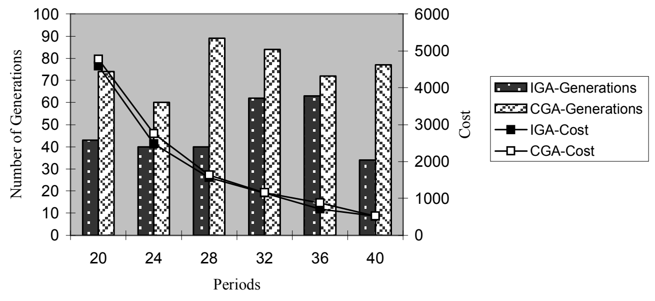


Figure 3. Bar chart for the results in Table 10.

4. Conclusion

We have presented an Incremental Genetic Algorithm (IGA), which is useful for re-optimizing problems that undergo small changes. Instead of starting with a randomly generated initial population, as in CGA, IGA uses information collected from the first run of a CGA on the initial version of the problem (prior to the changes).

The empirical results for three subject problems show that IGA evolves solutions faster than CGA and that these solutions have similar quality to those of CGA.

References

[1] Ahmad R. and Bath P. A., "The Use of Cox Regression and Genetic Algorithm (CoRGA) for Identifying Risk Factors for Mortality in Older People," *Health Informatics Journal*, vol. 10, pp. 221-236, 2004.

[2] Celeste A. B., Suzuki K., and Kadota A., "Genetic Algorithms for Real-Time Operation of Multipurpose Water Resource Systems," *Journal of Hydroinformatics*, vol. 6, pp. 19-38, 2004.

[3] Davis L. (Ed), *Handbook of Genetic Algorithms*, New York: Van Nostrand Reinhold, 1991.

[4] Goldberg D. E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Boston, Addison-Wesley, Reading, 1989.

[5] Haupt R. and Haupt S., *Practical Genetic Algorithms*, New York, Wiley & Sons, 1998.

[6] Hicks C., "A Genetic Algorithm Tool for Designing Manufacturing Facilities in the Capital Goods Industry," *International Journal of Production Economics*, vol. 90, no. 2, pp. 199-211, 2004.

[7] Holland J. H., *Adaptation in Natural and Artificial Systems*, Cambridge, Mass., MIT Press, 1992.

[8] Iuspa L., Scaramuzzino F., and Petrenga P., "Optimal Design of an Aircraft Engine Mount via Bit-Masking Oriented Genetic Algorithms," *Advances in Engineering Software*, vol. 34, no. 11-12, pp. 707-720, 2003.

[9] Li F., Zhang X., and Dunn R. W., "Development of an Optimal Contracting Strategy Using Genetic Algorithms in the UK Standing Reserve Market," *IEEE Transactions on Power Systems*, pp. 842-847, May 2003.

[10] Mansour N. and El-Fakih K., "Simulated Annealing and Genetic Algorithms for Optimal Regression Resting," *Journal of Software Maintenance*, vol. 11, pp. 19-34, 1999.

[11] Mansour N. and Timany M., "Soft Computing Algorithms for Exam Scheduling," Submitted for publication, 2003.

[12] Michalewicz Z. and Fogel D., *How to Solve it, Modern Heuristics*, Berlin, Springer, 2000.

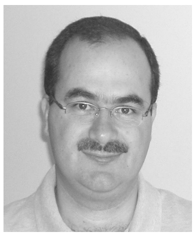
[13] Ombuki B. and Ventresca M., "Local Search Genetic Algorithm for the Job Shop Scheduling Problem," *Journal of Applied Intelligence*, vol. 21, no. 1, pp. 99-109, 2004.

[14] Saleh A. H. and Chelouah R., "The Design of the Global Navigation Satellite Surveying Networks Using Genetic Algorithms," *Journal of Engineering Applications of Artificial Intelligence*, vol. 17, no. 1, pp. 111-122, 2003.

[15] Schulze-Kremer S., "Genetic Algorithms for Protein Tertiary Structure Prediction," in Manner R. and Manderick B. (Eds), *Parallel Problem Solving from Nature*, North Holland, pp. 391-400, 1992.

Nashat Mansour is an associate professor at the Lebanese American University, Lebanon. He received his BE and MS degrees in Electrical Engineering from the University of New South Wales, Australia, and MS in Computer Engineering and PhD in computer science from Syracuse University, USA. Currently, he is an executive member of the Arab Computer Society. His research interests include software testing, applications of soft computing algorithms, and data/web mining.

Mohamad Awad received his BSc and MSc degrees in computer science from the Lebanese American University, Beirut, in 1988 and 2001, respectively. Currently, he is working on his PhD in signal processing at Rennes University, France. He has also been working as a research assistant at the Lebanese National Council for Scientific Research since 1996.



Khaled El-Fakih received the BSc and MSc degrees in computer science from the Lebanese American University and the PhD degree in computer science from the University of Ottawa in 2002. He worked as a graduate fellow at IBM Toronto Laboratory in 1997 and as a Verification Engineer at Cambrian Systems Corporation in 1998. He joined the American University of Sharjah in 2001, where he is currently an assistant professor. His research interests are in testing of communication protocols, fault diagnosis and synthesis of distributed systems, formulation of optimization problems and application of natural optimization heuristics.