AN OPERATOR BASED ADAPTIVE GENETIC ALGORITHM

Koh Sueyi, Leowsoo Kar, Loke Kar Seng
koh.sue.yi@gmail.com,{leow.soo.kar,loke.kar.seng}@infotech.monash.edu.my
Monash University Malaysia, No.2 Jalan Kolej, Bandar Sunway, 46150 Petaling Jaya, Selangor, Malaysia.

Abstract: Genetic Algorithms (GAs) are a robust heuristic search technique capable of taking on a broad range of optimization problems. In most GAs, components and parameters are predetermined and remain static throughout its run. In this paper, it is hypothesized that a GA's performance and robustness can be enhanced through the 'online' adaptation of the operators and an operator based adaptive genetic algorithm (AGA) based on these concepts is designed and implemented. A number of permutation based problems were selected to evaluate the performance of AGA.

Key words: adaptive genetic algorithm, permutation based optimisation problems

1. INTRODUCTION

Genetic algorithms have been known to be a robust technique in finding optimal or near solutions to many large and complex optimization problems. However, its parameter settings are required to be tuned for the specific problem at hand in order for it to perform efficiently and effectively during its search for an acceptable solution in a reasonable amount of time. While there exist a number of methods that can and have been used to predetermine such settings before running it on a problem, the problem specific nature of some of these settings coupled with the mutual dependence of these settings on one another still render the tuning of a GA to be a tedious process. Additionally, some research has indicated that the use of more than the usual one crossover and one mutation operator during the run of the GA can be beneficial. This however, results in more parameters being needed to be tuned. In this paper, an operator based adaptive genetic algorithm is presented with the aim of combining the benefits of having more operators available while decreasing the amount of parameter tuning required.

2. RELATED WORK

For a GA to be efficient, it has to be configured with settings suitable to the problem at hand. The most commonly used method is empirically through hand tuning. DeJong [2], Grefenstette [4], Davis [3] and David et al. [5] used this technique to find some recommended settings that were 'good' for a set of numerical function optimization problems, known as the DeJong test suite. Using a traditional algorithm with a fixed population size, they worked out 'good' parameter values for single point crossover and bit mutation.
Recently, Maruo et al. [15] used an approach of self-adapting parameters to relieve the burden of hand tuning. However, due to the number of components and the mutual dependence of these components on one another, finding the appropriate settings was still a tedious task.

Another approach [6] adaptively varies the settings. It involves the utilization of information gained about the state of the GA during its run, This information may be about the population as a whole, individuals in the population or components of the GA. This knowledge is then used to vary 'online' its settings. Based on how the adaptive mechanism is employed, the approaches in this category can further be split into three subcategories, namely, the rule based approach, [9], the co-evolutionary approach generations [6, 8] and the evolutionary approach.[10]

3. THE ADAPTIVE GENETIC ALGORITHM

The proposed GA consists of two genetic algorithms, an operator level (OL) GA and a problem level (PL) GA. The PL GA is used to search the problem space for good or optimal solutions to the problem at hand, while the OL GA synchronously selects the operators to be applied in the PL GA based on the status and requirements of the search. A list of 17 operators is selected for this purpose [11]. To evaluate the operator level population, the encoded operators in each chromosome are utilized on one generation of the PL GA with the same PL population. After which, feedback on the performance of each chromosome in the form of a resulting PL population is returned to the OL GA and each OL chromosome is assigned a fitness value based on its relative performance to the other OL chromosomes. Among the resulting PL populations, the one that resulted from the OL chromosome with the highest fitness value is selected to be the next PL GA population for the next generation of the OL GA.

Both the OL GA and PL GA use the same selection and replacement technique. The selection technique is used to select chromosomes or parents for recombination, while the replacement strategy refers to how individuals from the present population are selected to live on in the population used in the next generation. In Figures 1 and 5, they are denoted by the term Select and Perform_replacement respectively. After some experimentation, the tournament selection technique with the number of competitors set to 2 and the steady state with elitism strategy were selected as the selection technique and replacement strategy respectively.

In the next two subsections, the main components of the operator level GA and the problem level GA are described in detail.

3.1 Operator Level Genetic Algorithm

```
OL_GA (end_condition)
BEGIN
   Generate population
   Generate plPopulation
   PLEvaluate plPopulation
   /* evaluate operator level GA's population returns
   best resulting PL population */
   plPopulation = Evaluate (population, plPopulation)
   WHILE end_condition has not been reached
      best_chromosome = Get_fittest_chromosome (population)
      Initialise children of size num_children_required
      num_children_required = crossover_rate * population_size
      num_parents_required = Get_num_parents_required (crossover_operator)
      num_children_produced = Get_num_children_produced (crossover_operator)
      num_children = 0
      WHILE num_children < num_children_required
         parent_chromosomes = Select (population, num_parents_required )
         child_chromosomes =
            Perform_crossover_operation (parent_chromosomes)
            /* ensure that children does not go out of bounds during Append */
         Append child_chromosomes to children
         Increment num_children by num_children_produced
      ENDWHILE
      new_population = Perform_replacement (population, children)
      num_mutations_required = mutation_rate * population_size
      num_mutations = 0
      WHILE num_mutations < num_mutations_required
         Randomly_select chromosome from new_population
         Perform_mutation_operation (chromosome)
         Increment num_mutations
      ENDWHILE
      Replace last chromosome in new_population with best_chromosome
      plPopulation = Evaluate (population, plPopulation)
      Evaluate_population (population)
      population = new_population
   ENDWHILE
END
```

Fig. 1. Pseudo Code of the Operator Level GA

3.1.1 Chromosomal Representation

The OL GA employs a permutation based encoding for its chromosomes. Based on this encoding, each chromosome is comprised of all the available operators in the search space and it is the positions of the genes in the chromosome that determine which operators are passed down to the PL GA. A typical operator level chromosome is shown below:

8	3	14	5	15	16	6	1	9	17	7	2	12	11	13	4	10

Fig. 2. Example of an Operator Level Chromosome

The value of each gene denotes the operator being represented by it. The first n genes denote the operators that are to be passed to the PL GA, where n is a predetermined variable set before the run. Should $n = 5$, for example, the operators represented by values 8, 3, 14, 5 and 15 are passed to the PL GA to be utilized during its search.

3.1.2 Operators

Following classic style genetic algorithms, the operator level GA of AGA uses a standard set of one crossover operator and one mutation operator. Noting that only the first n encoded operators in the OL chromosome will be utilized during the evaluation of the OL chromosome, perhaps positional information can be said to be the primary information that needs to be captured. With this observation, it was decided that the Coin Toss operator [11] would play the role of the crossover operator, and Random Create Coin Toss operator would play the role of the mutation operator, where the use of this mutation operator ensures that each gene in each locus of the chromosome has a chance of being mutated. After some arbitrary testing, the rates are set to be 0.8 and 0.2 for the crossover and mutation respectively.

3.1.3 Evaluation Function

In AGA, the adaptation mechanism is the OL GA and the object to be adapted is the configuration of the PL GA. The OL GA evaluates the performances of the various configurations of the PL GA through the evaluation of the OL chromosomes. Thus, in AGA, communications between the OL GA and the PL GA occur in the operator level evaluation function, known as **Evaluate** in *Fig 1.* Because it is the settings of the PL GA that guide the search through the PL search space, and it is the evaluation function of the OL GA that guides the forming of the sets of operators used in the PL GA through the assignment of fitness values to these sets of operators, it can be said that the OL GA's evaluation function is one of the main components whose performance is vital to the overall performance of the AGA. The interaction between the two levels of AGA is illustrated in *Fig. 3*.

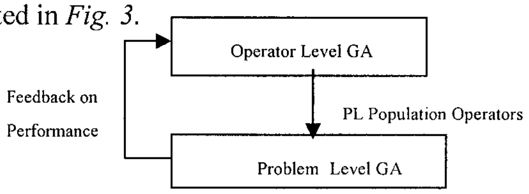

Fig. 3 Interaction between the Operator Level and Problem Level of the AGA

The pseudo code of the Evaluate function is given in *Fig. 4* and a detailed discussion of this function can be found in [11].

```
Population Evaluate (population: Population, plPopulation: Population)
BEGIN
  Initialise the array temp_plPopulations to be of size population_size
  FOR each chromosome in population
      operators = Decode_operators (chromosome, num_operators)
          PL_GA_set_operators (operators)
          pop = plPopulation
          /* num_generations in AGA is set to one */
          FOR (num = 0; num < num_generations; num+1)
          pop = PL_GA_Evolve (pop)
      ENDFOR
      Append pop to temp_plPopulations
  ENDFOR
  /* Calculate_and_assign_fitnesses returns best problem level population
  index based on the fitness values assigned by the fitness function */
  best_plPopulation_index =
          Calculate_and_assign_fitnesses (population, temp_plPopulations)
  best_plPopulation = temp_plPopulations[best_plPopulation]
  /* Updates the number of children produced by each of the operators in
    the problem level GA for during the evolution of best plPopulation */
  Update_PL_operator_children_produced (best plPopulation index)
  RETURN best_plPopulation
END
```

Fig. 4. Pseudo Code of Evaluate in OL GA

Seven fitness functions for the OL GA were formulated to properly gauge and capture the desired properties of the performances of the various configurations on the PL GA, or in other words, the fitness values of the chromosomes in the population of the OL GA. The candidate fitness functions are shown below:

$$Fitness\_Function\_1_{OL} = R(f_b) \tag{1}$$

$$Fitness\_Function\_2_{OL} = R(BC) \tag{2}$$

$$Fitness\_Function\_3_{OL} = \begin{array}{l} R(BC) \times Weight_{BC} + R(f_{sd}) \times Weight_{sd} \\ + R(f_m) \times Weight_m \end{array} \quad (3)$$

$$Fitness\_Function\_4_{OL} = \begin{cases} R(f_b) & f_b > prev.f_b \\ R(f_{sd}) + R(f_m), & f_b \le prev.f_b \end{cases} \quad (4)$$

$$Fitness\_Function\_5_{OL} = \begin{cases} R(BC) & f_b > prev.f_b \\ R(f_{sd}) + R(f_m) & f_b \le prev.f_b \end{cases} \quad (5)$$

$$Fitness\_Function\_6_{OL} = \begin{cases} R(f_b) & f_b > prev.f_b \\ R(f_{sd}) + R(BC) & f_b \le prev.f_b \end{cases} \quad (6)$$

$$Fitness\_Function\_7_{OL} = \begin{cases} R(BC) & f_b > prev.f_b \\ R(f_{sd}) + R(BC) & f_b \le prev.f_b \end{cases} \quad (7)$$

where $R(x)$ returns the rank of x associated with the chromosomes in the OL population,. A rank of 1 is the lowest rank and the rank value equal to the *population_size* is the highest rank. f_b is the fitness of the best candidate solution found in the associated PL GA's population; *BC* stands for Better Children, and is used to denote the number of child chromosomes that have been produced during a particular configuration that have better fitness values than the parents involved in creating them; f_{sd} is the standard deviation of the fitness values of the chromosomes in the associated PL population; f_m is the mean of the fitness values of the chromosomes in the associated PL population; $Weight_{BC}$, $Weight_{\mathrm{m}}$ and $Weight_{sd}$ are the weights assigned to *BC*, f_m and f_{sd} respectively, where they are set to $Weight_{BC} = 1.0$, $Weight_{sd} = 0.8$ and $Weight_m = 0.8$; and $prev.f_b$ refers to the fitness of the best PL chromosome found during the previous OL generation. For a detailed description of the 7 fitness functions the reader is referred to [11].

3.2 Problem Level Genetic Algorithm

The primary purpose of the problem level genetic algorithm is to find good solutions to the problem at hand. At the same time, it is also used to evaluate the OL GA chromosomes and return feedback about the performances of the settings encoded in these chromosomes for the current state of the PL search. In order to serve these purposes, during each generation of the OL GA, each OL chromosome is decoded and the PL GA is configured with these decoded operators that are then used in its search of the current search space in the form of a PL population. How this search is conducted is illustrated in the pseudo code of the PL_GA_Evolve function of the PL GA shown below in *Fig 5*. The operators to be used for the current search, known as *operators* in *Fig 4*, are set just before this PL_GA_Evolve function is called in the Evaluate function of the OL GA.

```
Population PL_GA_Evolve (plPopulation)
BEGIN
   Initialise children of size num_children_required
   FOR each operator in operators
       num_parents_required = Get_num_parents_needed (operator)
       num_children_produced = Get_num_children_produced (operator)
       number = Get_num_children_to_be_produced_by_operator (operator)
       num = 0
       WHILE (num < number)
           parents = Select (plPopulation, num_parents_required)
           child = Perform_Operation (operator, parents)
           PLEvaluate (child)
           Append child to children
           Increment num by num_children_produced
       ENDWHILE
   ENDFOR
   new_population = Perform_replacement (plPopulation, children)
   evaluate_population (new_population)
   RETURN new_population
END
```

Fig. 5. Pseudo Code of (PL)_GA_Evolve

Get_num_children_to_be_produced_by_operator is a function used to calculate the allotted number of children that are to be produced by the specified operator. This is calculated by getting the number of children needed to be produced in total per PL generation and evenly dividing them among the operators, operators, to be used in that generation. Should there be leftover children from the division, these are then distributed to the first occurring operators from the OL chromosome. As for the number of children to be produced in each generation, this is defined by a predetermined rate expressed as a fraction of the PL population size and is set before the start of the application of the AGA. In the implementations of the AGAs, this parameter is arbitrarily set to a value of 0.9.

3.2.1 Operator Settings

As mentioned in section 3.1.1, the operators to be used in the PL GA are determined by chromosomes in the OL GA population, with the number of operators to be deciphered from each chromosome to be used, predetermined before the run of the GA. With this parameter set, the rate for each operator is also indirectly determined, since each operator is assigned an equal number of children for it to produce from the number of children to be produced each generation. Hence, when compared to a classic GA that requires the crossover and mutation operators and rates to be retuned for different problems in the same domain, the proposed AGA, which only requires the number of operators to be used retuned, can be said to be more easily applied on different problems in the same domain.

3.2.2 Reproduction Strategy

In classic GAs, the crossover operator creates child chromosomes, while the mutation operator modify the existing chromosomes present in the new generation's population that may or may not have been produced by a crossover operator. However, because of the inability of distinguishing between crossover and mutation operators in a relatively problem independent GA such as this as well as for simplicity's sake, it was decided to have all operators produce children, such that a modified chromosome is also considered a child chromosome.

4. EVALUATION OF THE PERFORMANCE OF AGA

Seven AGAs with fitness functions corresponding to those described in 3.1.3 are implemented. Henceforth, the AGA that utilizes Fitness_Function_1OL will be referred to as (1), the AGA that utilises Fitness_Function_2OL will be referred to as (2) and so on. For all variations of the AGA, the OL population size is set to 20 and the PL population size is set to 50.

To provide benchmarks against which the proposed variations of the AGA may be tested against, a classic GA (RGA), and a variation of the AGA to act as a control (Control) are designed and implemented. To allow for fairer testing, RGA uses most of the same components as the problem level GA of the AGA, while Control uses the same components as the AGAs except that no evolutionary procedures and evaluations are performed in the OL GA. Hence, RGA provides a benchmark against classic GAs and Control provides a benchmark to gauge the effectiveness of the role of the OL GA.

The tuning process for the AGAs and Control consists of predetermining the number of operators to be used in the PL GA in the range of 1 to 9, while for RGA, this consists of finding the optimal crossover and mutation rates in the range of 0.5 to 1.0 and 0.0 to 0.5, in steps of 0.1, respectively.

4.1 Experimental Results

A number of experiments were conducted on a variety of problem sets [11]. The results of two problem sets are presented in this paper. The following describes the details of the tables; in the first column "RGA" refers to the regular GA, "Control" refers to the control adaptive genetic algorithm and (1), (2), (3), (4), (5), (6) and (7) represent the proposed adaptive genetic algorithms; the second column Opt shows the number of times the corresponding algorithm found optimal solutions; B.B.Fitness is an acronym for best best fitness and is the fitness of the overall best solution found among the set of tests; Mean.B.Fitness refers to mean best fitness and is the mean of the fitness values of the best solutions found for the set of tests of that particular problem; SD.B.Fitness is the standard deviation of the fitness values of the best solutions found during the set of tests for that problem; M.Time refers to the average amount of time taken to run the set of tests, where each algorithm is run on a particular instance of a problem until it finds the optimal or until it reaches the time limit; and under the column rank, states the rank of the algorithms according to their performance evaluation and in relation to one another, where a rank of 1 signifies the best performing algorithm and 9, the worst.

Algo	Opt	B.B.Fitness	M.B.Fitness	SD.B.Fitness	M.Time	Rank
RGA	0	0.070819	0.070825	3.43E-06	120.00	7
Control	0	0.070893	0.070946	2.64E-05	120.04	9
(1)	0	0.070825	0.070838	8.97E-06	120.06	8
(2)	7	0.070818	0.070818	1.19E-07	107.73	3
(3)	10	0.070818	0.070818	0	95.541	1
(4)	1	0.070818	0.070818	1.97E-07	119.94	4
(5)	1	0.070818	0.070819	3.88E-07	119.75	5
(6)	0	0.070818	0.070819	4.22E-07	120.05	6
(7)	7	0.070818	0.070818	2.98E-08	111.28	2

Table 1. Summary of the Performances of the GAs on the Knb150 Problem

Algo	Opt	B.B.Fitness	M.B.Fitness	SD.B.Fitness	M.Time	Rank
RGA	0	431	453.5	12.4036285	90.0037	9
Control	0	427	436.4	6.696267617	90.0193	8
(1)	1	426	435.2	4.643274706	82.5172	2
(2)	0	427	433.7	5.020956084	90.0123	4
(3)	0	427	435.4	4.923413450	90.0167	6
(4)	2	426	434	4.582575695	82.7868	1
(5)	0	427	433.7	5.866003750	90.0129	5
(6)	1	426	435.3	5.984145720	82.0365	3
(7)	0	427	435.8	3.841874542	90.0166	7

Table 2. Summary of the Performances of the GAs on the Eil51 Problem

Tables 1 and 2 shows the results of the experiments on the Knb150 problem set [12] and a 51 city travelling salesman problem (Eil51), [13]. From these tables we see that the AGAs outperform both the RGA and Control algorithms. In TSP problem, out of the 7 AGAs, 3 of them (AGA 1, 4 and 6) managed to find optimal solutions, whereas the RGA and control AGA failed to do so. Furthermore, the other 4 AGAs (2,3,5 and 7) performed better in terms of the consistency of the solutions found (lower standard deviations)

5. CONCLUSION

The results of the experiments showed that AGA is an effective approach in enhancing the performance of a GA. It consistently outperformed a regular GA throughout the various test problems, demonstrating the versatility of the adaptation mechanisms. Further, based on the different performances of the various AGAs in relation to one another on the different test problems, it was found that different adaptation mechanisms are better suited to different problems. The results also showed that the previously feared high overhead of such an approach was unfounded, since the proposed AGAs were able to find better solutions in a shorter time.

REFERENCES

1. Tuson, A. L. (1995), "Adapting Operator Probabilities in Genetic Algorithms", Unpublished M.Sc. thesis, Department of Artificial Intelligence, University of Edinburgh.
2. DeJong, K. A. (1975), "An Analysis of the Behavior of a Class of Genetic Adaptive Systems", University of Michigan, Ph.D. Dissertation.
3. Davis, L. (1991), "Handbook of Genetic Algorithms", Van Nostrand Reinhold.
4. Grefenstette, J. J. (1986), "Optimization of control parameters for genetic algorithms," in IEEE Trans. On Systems, Man, and Cybernetics.
5. David, J., Caruana, R. A., Eshelman, L. J. and Das, R. (1989), "A study of control parameters affecting online performance of genetic algorithms for function optimisation", in Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers.
6. Spears, W. M. (1995), "Adapting crossover in a genetic algorithm", in Proceedings of the 5th Conference on Evolutionary Programming, Cambridge MIT Press
7. Herrera, F and Lozano, M. (1996), "Adaptation of Genetic Algorithm Parameters Based on Fuzzy Logic Controllers", in Genetic Algorithms and Soft Computing, Physica-Verlag.
8. Tuson, A. L. and Ross, P. M. (1998), "Adapting Operator Settings in Genetic Algorithms. Evolutionary Computation", 6(2)
9. Hatta, K., Matsuda, K., Wakabayashi, S. and Koide, T. (1997), "On-the-fly crossover adaptation of genetic algorithms," in Proceedings of IEE/IEEE Genetic Algorithms in Engineering Systems: Innovations and Applications, pp.197-202.
10 Chen, S. and Liu, Y. (2001), "The application of multi-level genetic algorithms in assembly planning", Journal of Industrial Technology, 17(4)
11 Koh S. (2004) "An Operator Based Adaptive Genetic Algorithm for Permutation Based Optimization Problems" MIT Thesis, Monash University Malaysia.
12. Leow, S. K. and Koh, S. (2004), "Using the Invariant Optimal Assignment of a k-out-of-n: G system to Test the Effectiveness of Genetic Algorithms", in 8th IEEE International Conference on Intelligent Engineering Systems.
13. TSPLIB (2000), "TSPLIB – Travelling Salesman Problem Library", Available: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/, 24/5/2004
14. Burkard, R. E., Çela, E., Karisch, S.E. and Rendl, F. (2002), "QAPLIB – A Quadratic Assignment Problem Library", Available: http://www.seas.upenn.edu/qaPLib/, 12/6/2004.
15. Marcos H. M, Heitor S. L., Myriam R D. (2005), "Self-Adapting Evolutionary Parameters: Encoding Aspects for Combinatorial Optimization Problems" G.R. Raidl and J. Gottlieb (Eds.): EvoCOP 2005, LNCS 3448, pp. 154-165