Adaptive Genetic Algorithm and Quasi-Parallel Genetic Algorithm: Application to Knapsack Problem

K.Y Szeto[1,†] and Zhang Jian[1,2]

(1) Department of Physics, (2) Department of Mathematics, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong SAR, China
†Corresponding author: phszeto@ust.hk

Abstract. A new adaptive genetic algorithm using mutation matrix is introduced and implemented in a single computer using the quasi-parallel time sharing algorithm for the solution of the zero/one knapsack problem. The mutation matrix $M(t)$ is constructed using the locus statistics and the fitness distribution in a population $A(t)$ with N rows and L columns, where N is the size of the population and L is the length of the encoded chromosomes. The mutation matrix is parameter free and adaptive as it is time dependent and captures the accumulated information in the past generation. Two strategies of evolution, mutation by row (chromosome), and mutation by column (locus) are discussed. Time sharing experiment on these two strategies is performed on a single computer for solving the knapsack problem. Based on the investment frontier of time allocation, the optimal configuration for solving the knapsack problem is found.

1 Introduction

Parallel computation using the Darwinian principle of survival of the fittest has been implemented quite successfully in the framework of Genetic algorithms [1,2] with many successful application in many areas, such as solving the crypto-arithmetic problem [3], time series forecasting [4], traveling salesman problem [5], function optimization [6], and adaptive agents in stock markets [7,8]. However, the necessity of parameter setting in the application of genetic algorithm is a serious drawback for its practitioners, as its efficiency depends very much on the experience of the user on the problem at hand. One notable example of this drawback concerns the ad-hoc manner in the choice of the selection mechanism. One may need to use different percentage of the population for survival for different problems. Indeed, even for the same problem, the percentage of survivors in the evolution process should be time dependent for higher efficiency. Though some advances in adaptive parameter control on selection have been made, such as in the solution of the financial knapsack problem [9], the need for parameters setting remains. Here we like to address a novel way to do the selection process by the introduction of a mutation matrix that is time dependent but problem *independent.* We call our method mutation only genetic algorithm, or MOGA.

A second issue of parallel computation is the allocation of computer resource. It is desirable to devise a method for locating the optimal parameters in running the bottleneck program in a single computer that satisfies the criteria of both high speed and high confidence. Based on the ideas of Hogg and Huberman and collaborators [10], Szeto and Jiang [11] developed a formalism of quasi-parallel genetic algorithm, which is a method of combining existing algorithms into new ones that are unequivocally preferable to any of the component algorithms using the notion of risk in economics [12]. Here we assume that only one computer is available and the sharing of resource is realized only in the time domain. The concept of optimal usage is defined economically by the "investment frontier", characterized by low risk and high speed to solution. In this paper, we combine the work on mutation only genetic algorithm and time sharing in the framework of quasi-parallel genetic algorithm. We test this approach on the 0/1 knapsack problem with satisfactory results, locating the investment frontier for the knapsack problem.

2 Mutation Matrix

2.1 Mutation Matrix for Traditional Genetic Algorithm

In traditional simple genetic algorithm, the mutation/crossover operators are processed on the chromosome indiscriminately over the loci. The loci statistics is never employed. The recent work of Ma and Szeto [13] on Locus Oriented Adaptive Genetic Algorithm (LOAGA) has demonstrated the importance of the locus specific mutation rate for solving the zero/one knapsack problem. In this paper, we generalize their method and further demonstrate the advantage of using the information on the loci statistics on mutation operator. First let's show that traditional genetic algorithm can be treated as a special case in our formulation. We consider a population of N chromosomes, each of length L and binary encoded. We describe the population by a $N \times L$ matrix, which entry $A_{ij}(t), i = 1, ..., N; j = 1, ..., L$ being the value of the jth locus of the ith chromosome. The convention is to order the rows of A by the fitness of the chromosomes, $f_i(t) \le f_k(t)$ for $i \ge k$. Next we introduce a mutation matrix with elements $M_{ij} \equiv a_i(t)b_j(t), i = 1, ..., N; j = 1, ..., L$, where $0 \le a_i(t), b_j(t) \le 1$ are called the row mutation probability and column mutation probability respectively. Traditionally we divide the population of N chromosomes into three groups: (1) Survivors who are the fit ones. They form the first N_1 rows of the population matrix $A(t+1)$. Here $N_1 = c_1 N$ with the survival selection ratio $0 < c_1 < 1$. (2) The number of children is $N_2 = c_2 N$ and is generated from the fit chromosomes by genetic operators such as mutation. Here $0 < c_2 < 1 - c_1$ is the second parameter of the model. We replace the next N_2 population matrix $A(t+1)$ (3) The remaining $N_3 = N - N_1 - N_2$ rows are the randomly generated chromosomes to ensure the diversity of the population so that the genetic algorithm continuously explores the solution space. In our formalism, traditional genetic algorithm with mutation only corresponds to a time independent mutation matrix with elements $M_{ij} \equiv 0$ for $i = 1, ..., N_1$, $M_{ij} \equiv m \in (0, 1)$ for

$i = N_1, ..., N_2$, and finally we have $M_{ij} = 1$ for $i = N_2, ..., N$. Here m is the time independent mutation rate. We see that traditional genetic algorithm with mutation only requires at least three parameters: N_1, N_2 and m .

2.2 Mutation Probability

We first consider the case of mutation on a fit chromosome. We expect to mutate only a few loci so that it keeps most of the information unchanged. This corresponds to "*exploitation*" of the features of fit chromosomes. On the other hand, when an unfit chromosome undergoes mutation, it should change many of its loci so that it can explore more regions of the solution space. This corresponds "*exploration*". Therefore, we require that $M_{ij}(t)$ should be a monotonic increasing function of the row index i since we order the population in descending order of fitness. One simple solution is to use $a_i(t) = (i-1)/(N-1)$. Next, we must decide on the choice of loci for mutation once we have selected a chromosome to undergo mutation. This is accomplished by computing the locus mutation probability of changing to $X(X = 0 \text{or} 1)$ at locus j as p_{jX} by

$$p_{jX} = \frac{\sum_{k=1}^{N}(N+1-k) \times \delta_{kj}(X)}{\sum_{m=1}^{N} m} \tag{1}$$

Here k is the rank of the chromosome in the population. $\delta_{kj}(X) = 1$ if the jth locus of the kth chromosome assume the value X, and zero otherwise. The factor in the denominator is for normalization. Note that p_{jX} contains information of both locus and row and the locus statistics is biased so that heavier weight for chromosomes with high fitness is assumed. This is in general better than the original method of Ma and Szeto[13] where there is no bias on the row. After defining p_{jX}, we define the column mutation rate as

$$b_j = \frac{1 - |p_{j0} - 0.5| - |p_{j1} - 0.5|}{\sum_{j'=1}^{N} b_{j'}} \tag{2}$$

For example, if 0 and 1 are randomly distributed, then $p_{j0} = p_{j1} = 0.5$. We have no useful information about the locus, so we should mutate this locus, and $b_j = 1$. When there is definitive information, such as when $p_{j0} = 1 - p_{j1} = 0$ or 1, we should not mutate this column and $b_j = 0$.

3 Mutation Only Genetic Algorithm: MOGA

Once the mutation matrix $\mathbf{M}$ is obtained, we are ready to discuss the strategy of using $\mathbf{M}$ to evolve $\mathbf{A}$. There are two ways to do Mutation Only Genetic Algorithm. We can first decide which row (chromosome) to mutate, then which column (locus) to mutate, we call this particular method the *Mutation Only Genetic Algorithm by Row* or abbreviated as MOGAR. Alternatively, we can first select the column and then the row to mutate, and we call this the *Mutation Only Genetic Algorithm by Column* or abbreviated as MOGAC.

For MOGAR, we go through the population matrix $A(t)$ by row first. The first step is to order the set of locus mutation probability $b_j(t)$ in descending order. This ordered set will be used for the determining of the set of column position (locus) in the mutation process. Now, for a given row i, we generate a random number x. If $x < a_i(t)$, then we perform mutation on this row, otherwise we proceed to the next row and $A_{ij}(t+1) = A_{ij}(t), j = 1, ..., L$. If row i is to be mutated, we determine the set $R_i(t)$ of loci in row i to be changed by choosing the loci with $b_j(t)$ in descending order, till we obtain $K_i(t) = a_i(t) \times L$ members. Once the set $R_i(t)$ has been constructed, mutation will be performed on these columns of the ith row of the $A(t)$ matrix to obtain the matrix elements $A_{ij}(t+1), j = 1, ..., L$. We then go through all N rows, so that in one generation, we need to sort a list of L probabilities and generate N random numbers for the rows. After we obtained $A(t+1)$, we need to compute the $M_{ij}(t+1) = a_i b_i(t+1)$and proceed to the next generation.

For MOGAC, the operation is similar to MOGAR mathematically except now we rotate the matrix $\mathbf{A}$ by 90 degrees. Now, for a given column j we generate a random number y. If $y < b_j(t)$, then we mutate this column, otherwise we proceed to the next column and $A_{ij}(t+1) = A_{ij}(t), i = 1, ..., N$. If column j is to be mutated, we determine the set $S_j(t)$ of chromosomes in column j to be changed by choosing the rows with the $a_i(t)$ in descending order, till we obtain $W_j(t) = b_j(t) \times N$ members. Since our matrix $\mathbf{A}$ is assumed to be row ordered by fitness, we simply need to choose the $N, N-1, ..., N-W_j+1$ rows to have the jth column in these row mutated to obtain the matrix elements $A_{ij}(t+1), i = 1, ..., N$. We then go through all L columns, so that in one generation, we need to sort a list of N fitness values and generate L random numbers for the columns.

4 Quasi-parallel Algorithm

Now we switch our discussion from MOGA to the problem of allocation of computational resource to two algorithms: MOGAR and MOGAC in one single computer when solving a particular optimization problem. The framework for proper mixing of computing algorithms is the quasi-parallel algorithm of Szeto and Jiang [11]. Here we first summarize this algorithm. A simple version of our quasi-parallel genetic algorithm ($QPGA = (M, SubGA, \Gamma, T)$) consists of M independent sub-algorithms $SubGA$. The time sharing of the computing resource is described by the resource allocation vector Γ. If the total computing resource is R, shared by M sub-algorithms $G_i, i = 1, 2, ..., M$, with resource R_i assigned to G_i in unit time, then we introduce $\tau_i = R_i/R$, $0 \le \tau_i \le 1$ for any $i = 1, 2, ..., M$, and $\sum_{i=1}^{M} \tau_i = 1$, and the allocation of resource for sub-algorithms is defined by the *resource allocation vector*, $\Gamma = (\tau_1, \tau_2, ..., \tau_M)'$. In our case, we have $M = 2$ and our $SubGA$ are MOGAR and MOGAC. For resource allocation, we only have one parameter $0 < \gamma < 1$ which is the fraction of time the computer is using MOGAR, and the remaining fraction of time $(1 - \gamma)$ we use MOGAC. The termination criterion T is used to determine whether a $QPGA$ should stop running. Thus, we have a mixture of MOGAR with MOGAC in the framework of quasi-parallel genetic algorithm with a mixing parameter γ. The parallel genetic algorithm described above can be implemented in a serial computer. For a particular generation t, we will generate a random number z. If $z < \gamma$, then we perform MOGAR, otherwise we perform MOGAC to generate the population $A(t+1)$. We now apply this quasi-parallel mutation only genetic algorithm to solve the 0/1 knapsack problem and try to obtain the investment frontier that give the mixing parameter γ that yields the fastest speed for solving the problem while also running with most certainty (minimum risk) of getting the solution.

5 The Zero/One Knapsack Problem

The model problem to test our ideas on mutation only genetic algorithm is the Knapsack problem. We define the 0/1 knapsack problem [14] as follow. Given L items, each with profit P_i, weight w_i and the total capacity limit c, we need to select a subset of L items to maximize the total profit, but its total weight does not exceed the capacity limit. Mathematically, we want to find the set $x_i \in \{0,1\}, i = 1, ..., L$ to

$$\text{Maximize} \sum_{j=1}^{L} P_j x_j \text{ subjected to constraint } c \geq \sum_{j}^{L} w_j x_j . \tag{3}$$

We consider a particular knapsack problem with size $L = 150$ items, $c = 4000$. The set $P_i \in [0, 1000]$ and $w_i \in [0, 100]$ are chosen randomly to define our problem, but afterwards fixed. In order not to violate the constraint of the problem, we use two tricks, *"Punishment"* and *"Repairing"*. *Punishment* reduces the fitness when the constraint is violated, while *Repairing* modifies the chromosome (adding/deleting items) until the constraint is satisfied. We will use a method called Greedy Repair. If a chromosome violates the constraint (total weight is over the constraint in the Knapsack), the repair scheme will find the site k with minimum value of P_k/w_k and reset x_k to zero, i.e., removing the kth item from the knapsack. This process continues till the constraint is satisfied. When the constraint is satisfied, and if some empty space remains, Greedy Repair will tried to fill the knapsack "as full as possible" by picking up the unselected item (those sites m where x_m=0) and fill them in the knapsack in descending order of P_m/w_m. Repair operation stops once the constraint is violated. This scheme can repair all chromosomes into local optimal solution in Hamming space.

6 Results

In the early version of our ideas on mutation matrix [13], we have found that locus oriented adaptive genetic algorithm (LOAGA) outperforms dynamic programming which is the usual method for Knapsack problem. We have found evidence that mixing MOGAR with MOGAC in a time-sharing manner produces superior results compared to (LOAGA) in numerical experiment for the

0/1 knapsack problem. Here we like to find the optimal time sharing parameter by locating the investment frontier of our mutation only genetic algorithm.
We first define our MOGAR and MOGAC. We choose the simplest form of

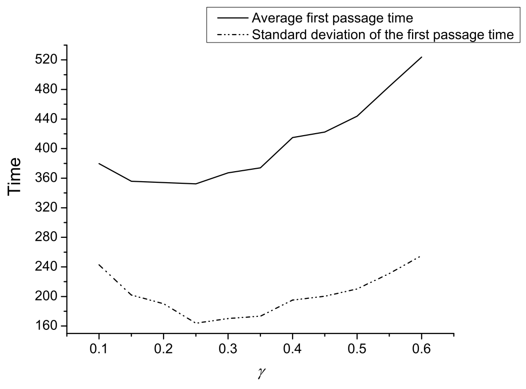

Fig. 1. Mean first passage time to solution and its standard deviation of 1000 runs as a function of time sharing parameter γ

$a_i(t) = (i-1)/(N-1)$. Here $N(= 100)$ is the size of the population in all genetic algorithms. For a given generation time t, we generate a random number z. If $z < \gamma$, then we perform MOGAR, otherwise we perform MOGAC to generate the population $A(t+1)$. Next we address the stopping criterion. We use exhaustive search to locate the true optimal solution of the knapsack problem. Then, we run our mixed MOGA in *QPGA* formalism 1000 times to collect statistical data. For each run, we define the stopping time to be the generation number when we obtain the optimal solution, or 1500, whichever is smaller. The choice of 1500 as the upper bound is based on our numerical experience since for a wide range of γ, all the runs are able to find the optimal solution within 1500 generations. Only for those extreme cases where γ is near 1 or 0, meaning that we use MOGAR alone or MOGAC alone, a few runs fail to find the optimal solution within 1500 generations. This is expected since we know row mutation only GA has low speed of convergence while column mutation only GA has early convergence problem. These extreme cases turn out to be irrelevant in our search of the investment frontier as demonstrated in Fig.1, where we plot the mean first passage time to solution and its standard deviation of 1000 runs as a function of the time sharing parameter γ. These results demonstrate the power of *QPGA* (i.e., time-sharing of computational resource) based on Mutation Only Genetic Algorithm. In Fig.2, we plot average first passage time to solution versus standard deviation. The curve is parameterized by γ. We see that there is a point on the curve which is closest to the origin. This point is unique in this experiment, corresponding to a value of $\gamma_c = 0.22 \pm 0.02$. The interpretation of this time

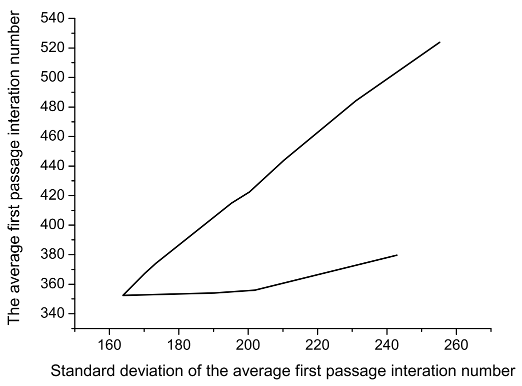

Fig. 2. Average passage time to solution versus standard deviation. The curve is parameterized by γ

sharing parameter is that our $QPGA$ will be fastest and most reliable (least risky) in finding the optimal solution of the 0/1 knapsack problem. In another word, the investment frontier of this problem consists of a single critical point at γ_c.

7 Conclusion

Using the simple observation that ordinary genetic algorithm can be considered as a special case of evolutionary computation using a special static form of mutation matrix, we develop a general formalism for mutation matrix that allows adaptive genetic algorithm without the need to preset selection parameters. We further generalize the evolution by making use of the locus statistics and develop MOGAC, *mutation only genetic algorithm by column*. This new algorithm has high speed of convergence. By combining it with MOGAR, *mutation only genetic algorithm by row*, we find a way to combine efficiently two processes: exploration of solution space and exploitation of the features of locus statistics for the fit chromosomes. The method we use is time sharing of MOGAR and MOGAC in the framework of quasi-parallel genetic algorithm. This methodology is tested on the 0/1 knapsack problem. We succeed in locating the critical value of time sharing in the investment frontier of mixing MOGAR and MOGAC to be 0.22, meaning that statistically we should use 22% of the computational resource on mutation by row, and 78% on mutation by column when solving the knapsack problem. Our general formalism can be used to address various types of optimization problems such as Ising model in random fields, Potts model, and traveling salesman problem. Our future work will address the incorporation of crossover in our formulation.

8 Acknowledgement

K.Y. Szeto acknowledged that this work is supported by RGC grant no. HKUST6157/01P and 603203.

References

[1] J.H. Holland, *Adaptation in natural and artificial systems.* Ann Arbor, MI: University of Michigan Press, 1975.

[2] D.E. Goldberg, *Genetic algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[3] S. P. Li and K.Y. Szeto, *Crytoarithmetic problem using parallel Genetic Algorithms*, Mendl'99, Brno, Czech, 1999.

[4] K.Y. Szeto and K.H. Cheung, Multiple time series prediction using genetic algorithms optimizer. *Proceedings of the International Symposium on Intelligent Data Engineering and Learning*, Hong Kong, IDEAL'98, 127-133, 1998.

[5] R. Jiang and K.Y. Szeto, Y.P. Luo and D.C. Hu, Distributed parallel genetic algorithm with path splitting scheme for the large traveling salesman problems. *Proceedings of Conference on Intelligent Information Processing, 16th World Computer Congress 2000, Aug.21-25, 2000*, Beijing, Ed. Z. Shi, B. Faltings, and M. Musen, Publishing House of Electronic Industry, 478-485, 2000.

[6] K.Y. Szeto, K.H. Cheung and S.P. Li, Effects of dimensionality on parallel genetic algorithms. *Proceedings of the 4th International Conference on Information System, Analysis and Synthesis, Orlando, Florida, USA*, Vol.2, 322 325, 1998.

[7] K.Y. Szeto and L.Y. Fong, *How adaptive agents in stock market perform in the presence of random news: a genetic algorithm approach*, LNCS/LNAI, Vol. 1983, Ed. K. S. Leung et al. Spriger-Verlag, Heidelberg, 2000, IDEAL 2000, 505-510, 2000.

[8] Alex L.Y. Fong and K.Y. Szeto, *Rule Extraction in Short Memory Time Series using Genetic Algorithms*, European Physical Journal B Vol.20, 569-572(2001).

[9] Kwok Yip Szeto and Man Hon Lo, An Application of Adaptive Genetic Algorithm in Financial Knapsack Problem, *The 17th International Conference on Industrial & Engineering Applications of Artificial Intelligenc & Expert Systems, Ed Bob Orchard, et al. May 17-20, 2004*, LNAI3029 Springer Verlag 2004.pp1220-1227.

[10] B.A. Huberman, R.M. Lukose and T. Hogg, *An economics approach to hard computational problems*, Science, Vol.275, No.3: 51 54, 1997.

[11] Kwok Yip Szeto and Jiang Rui, A quasi-parallel realization of the Investment Frontier in Computer Resource Allocation Using Simple Genetic Algorithm on a Single Computer, *LNCS 2367, 6th International Conference, PARA 2002, Espoo, Finland, June 15-18, 2002* pp.116-126. Springer-Verlag.

[12] H. Markowitz, J. of Finance, Vol.7, 77, 1952.

[13] C.W. Ma and K.Y. Szeto, Locus Oriented Adaptive Genetic Algorithm: Application to the Zero/One Knapsack Problem, *Proceeding of The 5th International Conference on Recent Advances in Soft Computing, RASC2004 Nottingham, UK.* p.410-415, 2004.

[14] V. Gordon, A. Bohm, and D. Whitley, A Note on the Performance of Genetic Algorithms on Zero-One Knapsack Problems, *Proceedings of the 9th Symposium on Applied Computing (SAC'94), Genetic Algorithms and Combinatorial Optimization, Phoenix, Az*, pp 194-195, 1994.