Optimal Mutation rates and Recombination: The effect of population size and gene length

Alice Eldridge

October 2, 2002

Abstract

Optimal parameters setting have been the subject of much research. Early work suffered from methodological and theoretical weaknesses that limited the predictive value of the findings. A recent paper by Ochoa et al [20] reported the influence of recombination on optimal mutation rates and proposed a theoretical link between optimal mutation rates and error thresholds. Findings are presented from a preliminary investigation into the effects of population size and chromosome length on the influence of recombination. Results suggest that the effects are independent of population size, but may be affected by changes in chromosome length. The possibility that the effect persists under different selection schemes is also considered.

1 GA Parameter Settings

Problems with early research The performance of a GA is crucially dependent on the choice of the main parameters: mutation rate, crossover rate and population size. This has long been acknowledged, and the search for optimal parameter settings has been the subject of much research.

The 'optimal' settings for parameters is however in turn dependent on the other operators and characteristics of the problem. This was not been so well recognised in previous research. For example, previous theoretical research on optimal mutation rates has examined mutation operator in isolation, ignoring recombination in order to simplify the analysis. [15],[17]. Although there have been empirical studies that recognise and examine the interactions between settings [22], many studies have tended to focus on finding a 'general' set of optimal settings experimentally, [9],[14] using a fixed test suite. The results of such studies typically have weak predictive value regarding relative performance on new problems, as the results may not be generaliseable beyond the particular test-suite used.

Research dedicated to finding a 'general' set of parameter settings is indicative of the early conceptualisation of the GA as a robust general purpose optimiser that will exhibit the same behaviour over a range of problems . It is now acknowledged that lunch must be paid for [24] : specific problem types require specific settings for satisfactory performance [4].

Currently a variety of approaches are advocated for tuning parameters. One option is to employ theoretical analysis to parameters by "analogy", but the practical value of current theoretical results is unclear [1]. There has been extensive research into the use of dynamic parameter control - deterministic (time-dependent deterministic rule), adaptive (through feedback) or self-adaptive (where the parameters themselves are encoded and sub-

[1]L.Davis advised the best thing a practitioner can do is 'stay away from theoretical results'. Workshop on evolutionary Algorithms, University of minesotat, Oct 1996

ject to evolution) [2]. In particular the use of adaptive mutation rates is advocated.

Although there is a lack of consensus regarding the superiority of using complex self-adaptive control procedures in general, it seems that for mutation rates, the optimal strategy (that which solves the optimisation or search problem with most efficiency), is a generally acknowledged heuristic of starting with a relatively high mutation rate and decreasing it over the course of a run. This is applicable to a GA without recombination .

1.1 Error Thresholds, Optimal Mutation Rates and Recombination

Rather than developing GA theory or individually evolving parameter control , a more intuitively attractive approach has been to refer back to the original inspiration for EAs, namely biological theory. In particular several authors has proposed a relationship between optimal mutation rates and error threshold - a notion from molecular evolution [15],[16].

The error threshold is the critical mutation rate beyond which structures obtained by the evolutionary process are destroyed more frequently than selection can produce them. Optimal solutions will only be stable in the population for mutation rates below this critical value. The correspondence between optimal mutation rates and the error threshold is seen as both are intuitively related to the idea of a balance between exploitation and exploration in genetic search. The relationship is supported by biological evidence [11].

Recombination also contributes to this balance, as in the early stages of an algorithm run, when the genetic variance is high, it acts as a diverging operation, much like mutation. We would intuitively think that the effects of two divergent operators would be additive, and indeed, the effect of recombination in lowering the error threshold has been demonstrated for infinite populations in mathematical models [8]. Ochoa, Harvey and Buxton (1999) developed an empirical means of determining the error threshold in a GA and showed that for finite populations, this threshold, as well as optimal mutation rate was higher for GAs without recombination [21].

These results are important in that they relate empirical findings regarding the optimal strategy for mutation rates [12],[17],[1] to an aspect of molecular evolution theory. This offers the potential for furthering our theoretical understanding of EAs. It may also prove to be of practical importance as the heuristics advanced for determining error thresholds could be useful guidelines for establishing optimal mutation rates. [2]

The effects of recombination on optimal mutation rates in a GA were replicated in a recent GA paper by Ochoa et al (1999). For a variety of problems, it was demonstrated that the optimal mutation rate is dependent on whether or not recombination is used. With recombination (GA), the best performance is observed for a low mutation rate throughout the run. Without recombination (GA-m), however, a higher mutation rate is more effective at the start of the run, whilst as the population converges upon the optimal solution towards the end of the run, a lower mutation rate is favoured.

1.2 Methods

The results have arguably more predictive power than those of previous empirical studies, as they were obtained using random problem generators, removing the possibility of hand-tuning. This new empirical methodology pro-

[2]This potential is slightly limited in that the error threshold can only be ascertained once the global peak is known. The application in real world problems will therefore be limited to cases where a region of landscape or member of one class of landscapes, analogous to the problem is already familiar.

posed by De Jong, Potter and Spears (1997), [10] affords a more valid means of testing EAs. They have developed a series of 'problem generators', abstract models that produce randomly generated models. By recording the average behaviour of GA over a number of such random problems, the predictive power of the results for the particular class of problem increases with the number of trials.

The NK landscape generator One of the problem generators used was the NK Landscape generator [10]. This is based on Kauffman's tunable NK model of fitness landscapes [16]. Points on the landscape are bit strings of length N and K describes the degree of epistatic interactions or linkages between each gene. The fitness of each locus is determined by taking the bit value at that locus, together with the value of the K interacting loci as an index to a look up table of size $N2^{k+1}$ containing randomly generated uniformly distributed values over [0.0, 1.0]. The set of K interacting bits for each locus can be assigned randomly or selected adjacently. The fitness of each chromosome is simply the average of the sum of the fitness values at each locus as follows.

$$f(chromosome)=\frac{1}{N}\sum_{i=1}^{N} f(locus_i).$$

One draw back with this problem generator, is that the amount of storage used ($N2^{k+1}$) for teh fitness table makes it prohibitive to explore large values of K for long chromosomes.

2 Population size and Chromosome length

Ochoa et la (1999) demonstrated the effect of recombination on optimal mutation rates for both NK and multi-modal problems. However, only one size of population (100) and chromosome length (100) were used. If these results are to stand as theoretical or practical guides, it must be shown that they are independent of the parameter values used. If mutation rate is related to error threshold, then we would expect recombination to have the same effect irrespective of population size or chromosome length (N), although the value of the optimal rate may change.

The current study aims to replicate the results of recombination on optimal mutation rates reported in Ochoa et al (1999) for NK problems, and explore the effects of reducing the population size and chromosome length.

Population size
This is a relatively high population size compared with many previous empirical findings : Grefenstette's meta-GA produced a population size of 30-80 [14], De Jong experimented with sizes from 50 to 100 [9] and Schaeffer et al recommended 20-30 [22]. In the original development of the NK problem generators [10] GAs with and without crossover and mutation operators were compared. Population sizes of 50 - 200 were explored and reported to have no effect. There is no empirical or theoretical evidence to suggest that the effects reported in the Ochoa paper will not persist for a smaller population size. However conventional wisdom dictates that the population may converge quicker, perhaps prematurely, resulting in a reduction in the optimal solution achieved.[14].

Chromosome length
The chromosome length of 100 used in the original study is also high compared to that used in comparable studies (De jong et al, 1997 use N = 48). Here the effects of reducing chromosome length are explored. Ochoa proposes that the effects will persist independent of chromosome length, although the actual optimum mutation rates may change. For smaller values of N, we may expect a higher mutation rate [3]. Chromosome length effects are reported to be independent of population size. Each parameter will therefore be varied independently,

maintaining the other parameter at the values used in Ochoa et al, allowing a valid comparison with the original results.

2.1 Initial Replication

In order to test whether the effects reported in by Ochoa could be observed for different population sizes and chromosome lengths, it was necessary to first replicate the results using the original parameters. This turned out to be somewhat of an investigation in itself!. The actual parameters were clearly reported as follows:

chromosome length	100
population size	100
crossover rate	0.6
Mutation rates	0.001, 0.005, 0.01
generations	1000
No of problems	20

The description of operators and selection procedure were ambiguous. The operators were reported as 'two-point crossover ' and 'standard bit mutation', the selection procedure simply as 'fitness proportionate selection'.

Cross-over

It was not clear whether the two point crossover used ensured that recombination occurred at exactly two loci, or allowed for the possibility of one point or none (if for example both loci were at end-points of the chromosome). I adopted a 'middle-ground' and implemented a two-point crossover procedure that ensured that cross-over did take place, but allowed for the possibility of one point. (ie, ensured that at least one of the crossover points was not an end point).

Mutation rate

It was not explcicitly stated whether the mutation rates reported refered to the chromosome or locus. However, given a chromosome length of 100, the values of reported mutation rates (0.001 - 0.01) suggest that these are the probabilities of mutation at each locus. The mutation operator in the current GA was implemented under this assumption.

Selection procedure

The selection procedure was reported simply as 'fitness proportionate selection'. This is slightly ambiguous. The term could be seen to refer to 'roulette wheel'[5] style or Stochastic Universal Sampling [5], although in some literature 'fitness proportionate' refers to the fact that absolute fitness values are used, distinguishing these techniques from rank-based selection procedures. This would therefore include truncation selection [18], local selection [13] and tournament selection [7].

2.1.1 Programme implementation and testing

GA Initially, a generational GA using 'roulette-wheel' selection was written with the operators specified above. Each operator was tested by printing out the population before and after application of operators, and ensuring that mutation and recombination occurred at the expected loci. The GA was then tested on a trivial 'max ones' problem and performed as expected.

Problem generator The NK generator was implemented by creating a lookup table (hashtable)with N columns and 2^{k+1} rows representing each of possible patterns of interacting genes. The table is initialised for each new problem with uniformly distributed numbers in the range [0.0 - 1.0].

The fitness of each chromosome is found by first determining an integer representation of the binary 'pattern' of the K interacting loci at each locus and then using this 'pattern' number and locus position to index the look-up table. The sets of interacting bits were determined by according to the 'neighbourhood model', by selecting adjacent loci. [3]. For example for the bit string 10111, when K = 2, the 'pattern' associated with the gene at the second locus is 101 and is stored as 5. When the chromosome is evaluated, the fitness value of the second locus is the value that has been randomly assigned to the sixth row of the second column. This is repeated for each locus, and the fitness value of the entire chromosome computed by summing the fitness contribution of each locus and dividing by N. An example output is given in appendix 1.

2.1.2 Replication results

Roulette Wheel A GA using roulette wheel style selection, two point crossover and standard bit mutation with the above parameters did not produce the effects reported in Ochoa et al. With recombination (fig 1), the highest mutation rate was optimal throughout the run and without recombination (fig 2) the effect was exaggerated. Re-writing the selection procedure using stochastic universal sampling produced a similar effect. In an attempt to ascertain if the GA itself was at fault, a previously validated GA was run on the NK problem. Roulette wheel selection with this GA produced similar results, suggesting that roulette-wheel selction was not used.

Tournament Selection By replacing roulette wheel with tournament selection (tournament size 4) however, the effect of recombination on optimal mutation rates described by Ochoa for K=0 is observed (figs 3 & 4). Standard deviations are not shown for the sake of clarity , but were all below 0.01 and are given in appendix 1.

[3]due to restraints, only even K values can be used!

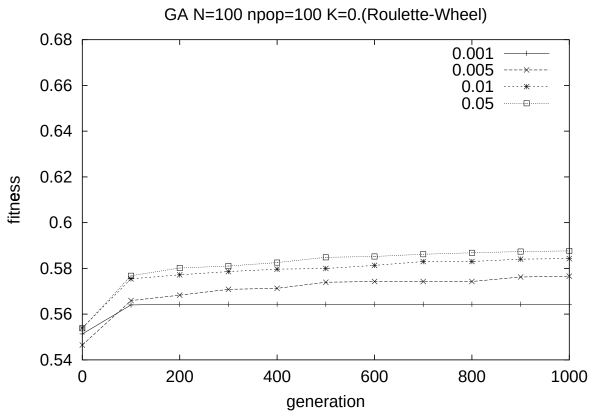

Figure 1: Average best-so-far curves GA Using Roulette-Wheel selection

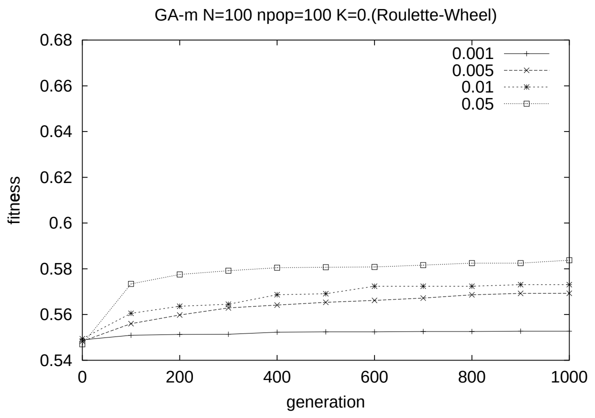

Figure 2: Average best-so-far curves GA-m Using Roulette-Wheel selection

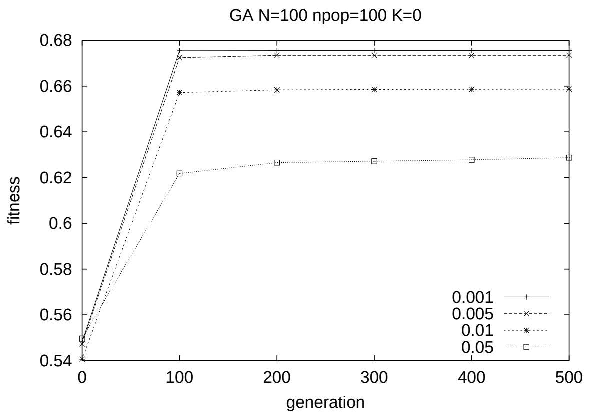

Figure 3: Average best-so-far curves GA using Tournament selection

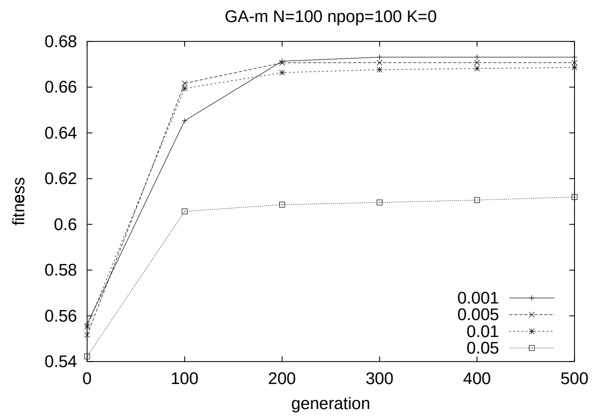

Figure 4: Average best-so-far curves GA-m using Tournament selection

Although the effect is small, it can be seen that with recombination (GA), the lowest mutation rate used produces the best results over the entire run. Without recombination (GA-m) the higher mutation rates are initially more effective, but once converged, the lower rate produces a higher fitness. It is of note that the GA converges extremely rapidly (1000 generations were run, but only the first 500 are displayed). It is recognised that there exists a fundamental difference between the Ochoa algorithm and the current implementation. Demonstration of the pattern for NK problems with zero epistasis inspite of this discrepancy can be seen to strengthen the generality of the effects.

3 Design

An investigation into the effects of population size and chromosome length was carried out using a generational GA employing tournament selection with a tournament size of 4. This tournament size affords a good balance between loss of divergence, selection variance and selection intensity. Two point crossover and point mutation were implemented as above.

Experimental conditions Due to the storage requirements of the look up tables (and the limitation of of my home computer!), it was not possible to explore large values of K for N = 100. Therefore for the population size conditions, K = 0 was employed. Whilst error thresholds are not applicable to such trivial landscapes, the optimum mutation rate effects are reported for this single peak landscape. Larger values of K were explored for smaller chromosome lengths.

The values of population size and chromosome length examined are shown below. The separate effects of altering these parameters were explored by varying them independently, keeping all other parameters constant. Each was run on both GA and GA-m. Possible interaction was not explored any chromosome length effects are reported to be independent of population size [3],[18].

The performance metric monitored was "best-so-far" curves. Standard deviations were recorded but due to a technical problem are not available. All were of the same order as those for the control condition given in appendices 2 and 3.

experimental variables	
population sizes	80 60 40 20
chromosome length(N)	80 60 40 20
mutation rates	0.001 0.005 0.01 0.05
constant parameter values	
Recombination probability	0.6
generations	1000
no problems	20

4 Experimental results

4.1 Population size

Experiments were run for population sizes of 80, 60, 40 and 20 for NK problems with K = 0. Figs 5 & 6 show the average best-so-far curves for GA and GA-m with a population size of 60. Figures 7 & 8 illustrate the results for both

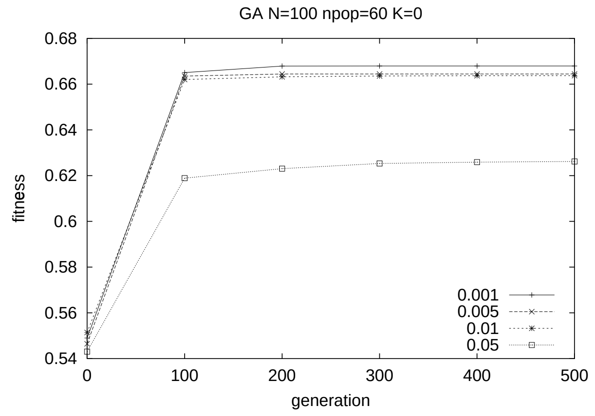

Figure 5: Average best-so-far curves.
GA population size 60

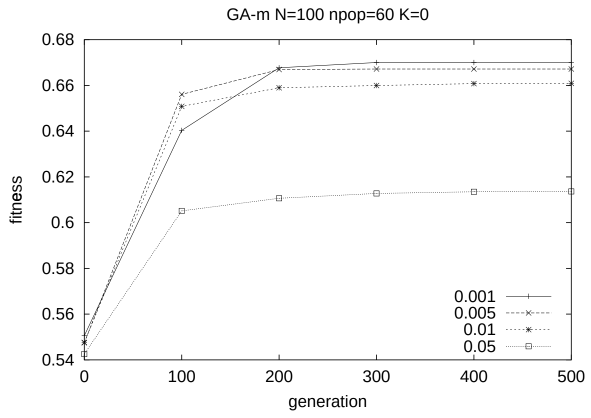

Figure 6: Average best-so-far curves.
GA-m population size 60

GAs with a population size of 20. These results are representative of all values tested, full results are presented in appendix 4

It can be seen that although small, the main effect did not differ significantly across conditions, and is comparable to that on the control condition. Although the difference is small, when recombination is used, the lowest mutation rate explored produces the best results over the whole algorithm (Figures 5 and 7) without recombination, search is initially enhanced by the higher mutation rates, but the lower rate achieves a higher fitness ultimately.

4.2 Chromosome length

GA with recombination Figures 9, 10, 11 and 12 illustrate the effects of decreasing the chromosome length on the optimal mutation rate for a GA with recombination. As expected, as N is decreased, the optimal mutation rates increase. The initial mutation rates employed were conservatively low, the highest rate employed (0.05) proving to be the optimal for N = 20. An additional investigation was carried out with higher mutation rates to establish if 0.05 exceeded the performance of

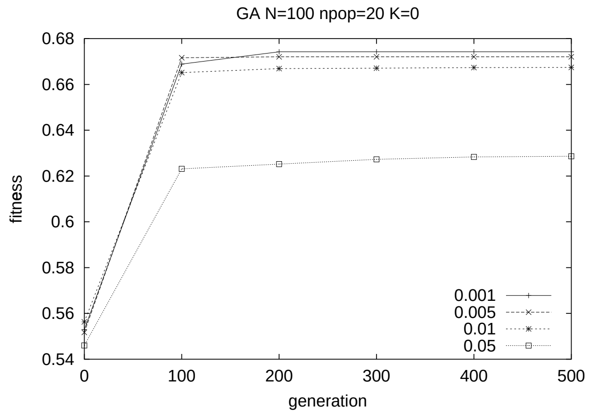

Figure 7: Average best-so-far curves
GA for population size 20

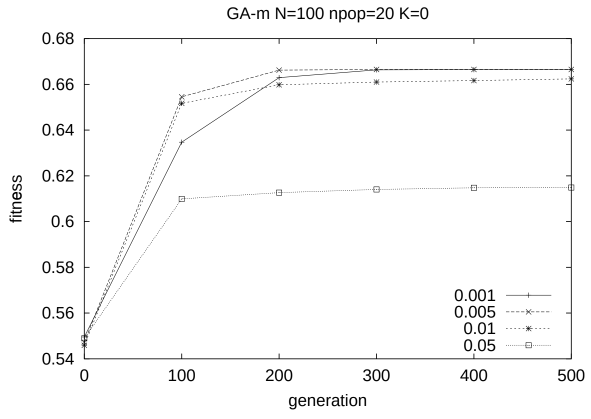

Figure 8: Average best-so-far curves
GA-m for population size 20

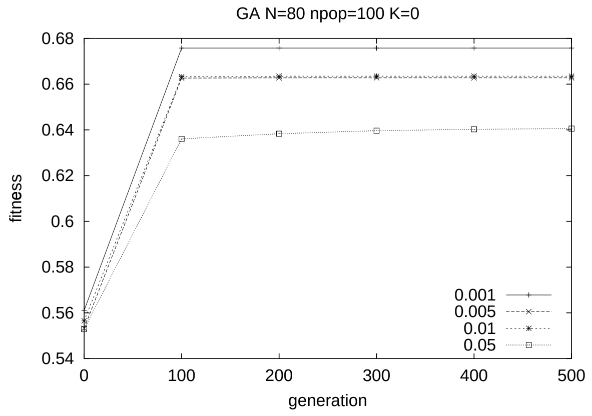

Figure 9: Average best-so-far curves
GA, N = 80

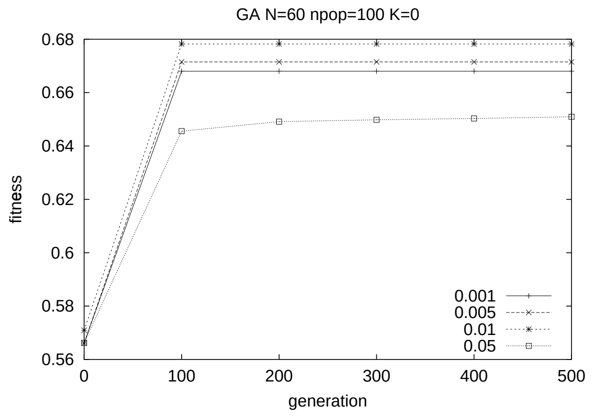

Figure 10: Average best-so-far curves
GA, N = 60

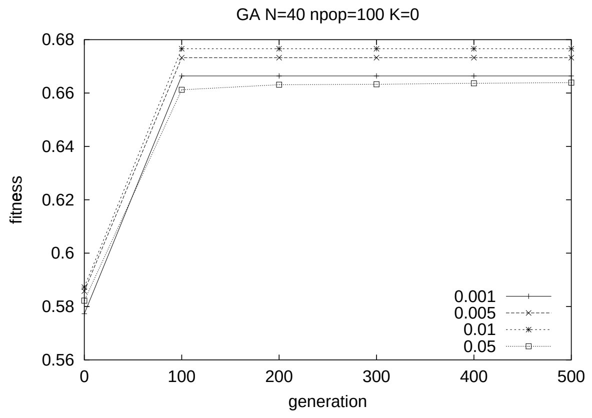

Figure 11: Average best-so-far curves
GA, N = 40

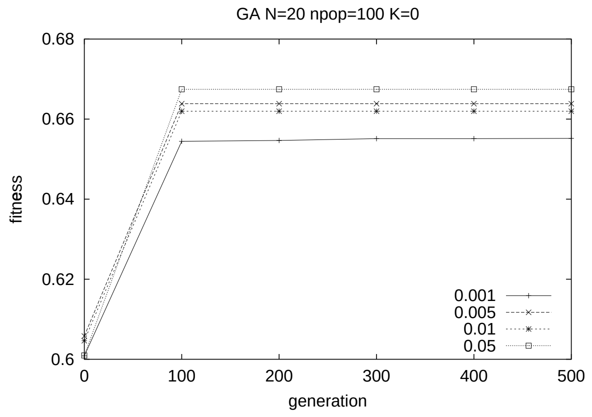

Figure 12: Average best-so-far curves
GA, N = 20

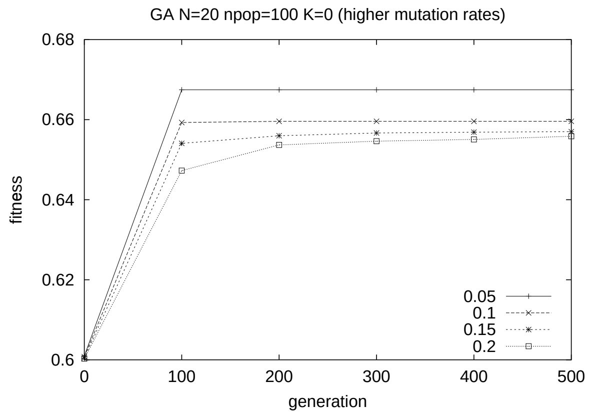

Figure 13: Average best-so-far curves
GA, N = 20 using higher mutation rates

higher rates, as in the Ochoa study. As fig 13 shows it is possible to replicate the qualitative results reported in the Ochoa paper for a GA with recombination, in that the lowest mutation rate used achieves the highest fitness.

Increasing epistasis For lower values of N, it was possible to examine problems with higher epistasis. Figure 14 shows the average best-so-far curve for a chromosome length of 60, level of epistasis 4. Figures 15 illustrates the effects

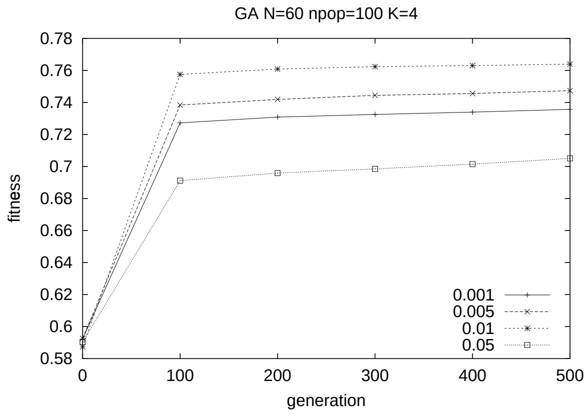

Figure 14: Average best-so-far curves
GA, N = 60, K = 4

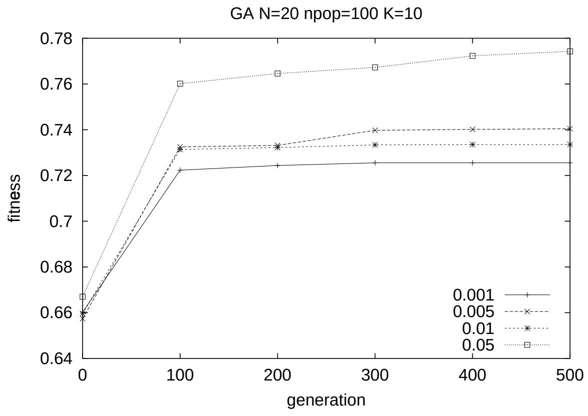

Figure 15: Average best-so-far curves
GA, N = 20, K = 10

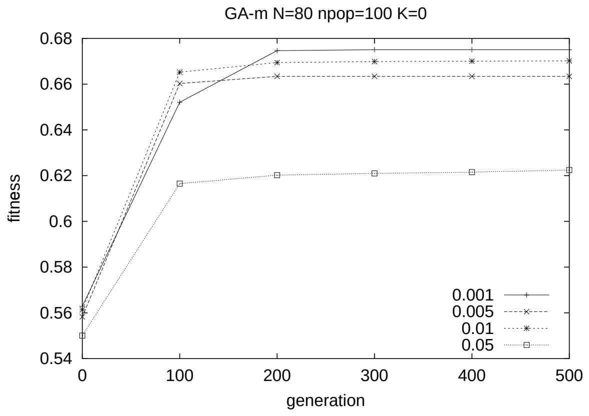

Figure 16: Average best-so-far curves
GA-m, N = 80

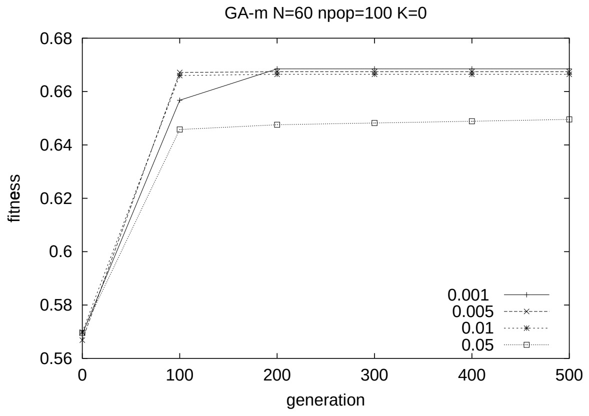

Figure 17: Average best-so-far curves
GA-m, N = 60

of medium epistasis (K = 10) on GA, N = 20. These results suggest that increasing the level of epistasis increases the relative effectiveness of the optimal mutation rate, as in Ochoa et al.

GA-m without recombination However, the results for a GA without recombination (figs 16, 17, 18 and 19) suggest that this is not the same effect that Ochoa reported. Although reducing N seems to have an effect mutation rate in increasing the effectiveness of the highest rate used (0.05), the optimal mutation rate does not vary across different values of N. Lower mutation rates are outperformed by higher rates in the early stages for longer chromosome lengths, this effect decreases as N is decreased. For lower values of N, the lowest mutation rate produces the best performance over the whole run. It should be noted that as N decreases, the maximum fitness is achieved even earlier, and this may be obscuring the effect.

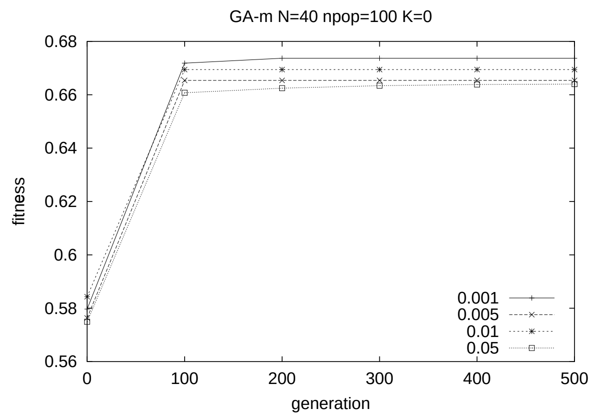

Figure 18: Average best-so-far curves GA-m, N = 40

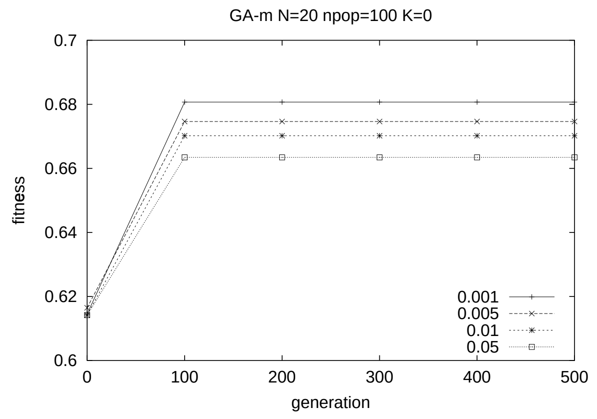

Figure 19: Average best-so-far curves GA-m, N = 20

5 Discussion

The results of this preliminary investigation suggest that the effects of recombination on optimal mutation rates are independent of population size as expected. It is perhaps surprising that reducing the population size did not affect the value of the fitness achieved as conventional wisdom dictates that a larger population will achieve a better solution. The reduced search space resulting from a diminished population size can cause convergence to a suboptimal solution, however this effect is dependent on problem characteristics. It may be that the uniform distribution of fitness values in this single peak landscape means that achieving the optimal solution is not dependent upon the size of search space. It would be interesting to examine the results of decreasing population size on problems with higher epistasis.

The results obtained from decreasing the chromosome length are interesting, although it is difficult to ascertain the exact nature of the effects due to the rapid convergence of the GA and the minimal differences in fitness values for the mutation rates used. The GA-m average best-so-far plots suggest that the mutation rate of 0.001 is optimal for *all* values of N. For longer chromosome (N = 80 and N = 60) this lower rate is initially outperformed by higher rates as for N = 100 . Due to the rapid convergence for lower values of N it is not possible to establish if this persists for shorter chromosome.

Irrespective of this uncertainty, in contrast to teh GA without recombination, the optimal mutation rates for a GA with recombination increased for successively shorter chromosome as is typical. This discrepancy in optimal mutation rates with and without recombination suggests that recombination does influences the optimal mutation rate for shorter chromosome, but the effect is different to that for larger values of N.

In explaining the effect for longer chromosomes, Ochoa proposed a dual role for recombination. It was proposed that the nature of the operator changed as a function of genetic variance. Therefore at the start of a run, when the genetic variance is high, recombination has a divergent influence, enhancing the search power of the algorithm. As the genetic homogeneity increases over the run, recombination functions to focus the population around the optimum, acting as an error repair mechanism.

The current results do not provide enough information to speculate over the exact effects of reducing the chromosome length on this dual role for recombination. It is possible that the differences are a results of the selection method used, or reveal an anomaly in the GA or problem generator. If tournament selection was not used in the original study, the results provide support for the generality of the the effect of recombination on optimal mutation rates over a range of population sizes as well as selection methods.

Although the NK problem generator provides are more representative than previous test suites used (eg [9]) it seems necessary to test for the effects of recombination on landscapes with neutrality such as the NK-p landscapes [6].

6 Postcript

Although framed as a scientific study, the principle purpose of this piece of work was to develop the skills and conceptual understanding required to perform it! In that respect, it has proved an invaluable experiment. Not only in an initiation in programming but the use of all the peripheral software and technique necessary to analyse and realise the results, as well as an introduction to the workings of evolutionary algorithms. That which I have learnt from my mistakes outweighs any accomplishments.

References

[1] Back, T. (1991). Self adaptation in genetic algorithms. In Varela, F.J. and Bougine, P., (eds) *ECAL 1. Toward a practice of Autonomous systems*, p 263-271, Paris, France, MIT press, Cambridge, MA

[2] Back, T. (1992) The interaction of mutation rate, selection and adaptation within a genetic algorithm. Manner, B. and Manderik, R., (eds) *Parallel Problem Solving from Nature, 2: Proceedings from the second Conference on Parallel Problem Solving from nature, Brussesl*, p 15-25. North Holland.

[3] Back, T. (1993). Optimal mutation rates in genetic search. In Forrest, S., (ed)*ICGA 5*, p 2-8, san Mateo, CA, USA. Morgan Kaufmann

[4] Back, T., Fogel, D. and Michalewicz, Z. (eds)(1997) *Handbook of Evolutionary Computation* Institute of Physics Publishing ltd, Bristol, Oxford University Press, New York.

[5] Baker, J.E. (1987) *Reducing Bias and inefficiency in the selection algorithm* in ICGA2 pp14-21

[6] Barnett, L.,(1997) Tangled webs: Evolutionary dynamics on fitness landscapes with neutrality. Masrter's thesis, School of Cognitive an Computing Sciences, University of Sussex.

[7] Blickle, T. & Thiele, L. (1995) : *A comparison of selection schemes used in Genetic Algorithms*TIK report Nr 11.

[8] Boerlijst, M.C., Bonhoeffer, S., and Nowak, M.A. (1996). Viral quasi-species and recombination. *Proc. R.Soc. London. B*, 263:1577-1584.

[9] De Jong, K. A. (1975) *Analysis of behaviour of a class of Genetic Adaptive Systems* PhD Thesis, University of Michigan, Ann Arbor. MI.

[10] De Jong , K.A., Potter, M.A., and Spears, W.M.(1997) Using problem generators to explore the efects of epistasis. In Back, T.*ICAL 7*p 338-345, San Fransisco. Morgan Kaufmann.

[11] Eigen, M. and Schuster, P. (1979) *The Hypercycle: A principle of Self-organisation.* Springer Verlag.

[12] Fogarty, T.C. (1989). Varying the probability of mutation in teh genetic algorithm. In Schaffer, J. D., (ed) *ICAL 3* p 104-109, George Mason University. Morgan Kaufmann.

[13] Goldberg, D.E. & Deb, K. (1991) *A comparitive analysis of selection schemes used in Genetic Algorithms*FGAI p 69-93.

[14] Greffenstette, J.J. (1986). Optimisation of control parameters for genetic algorithms. *IEE Trans SMC*, 16(1): 122-128.

[15] Hesser, J. and Manner, R. (1991) Towards an optimal probability for genetic algoriths. In Schwefel, H.P. and Manner, R. (eds). *parallel Problem Solving from Nature* Springer Verlag, Lecture Notes in Computer Science Vol. 496.

[16] Kaufmann, S.A. (993). *The Origins of Order: Self-Organisation and Selection in Evolution.* Oxford University Press.

[17] Muhlenbein, H. (1992). How genetic allgorithms relly work: Mutation and hillclimbing. In Manner, B. and Manderik, R., (eds) *Parallel Problem Solving from Nature, 2: proceedings form the second Conference on Parallel Problem Solving from nature, Brussesl*, p 15-25. North Holland.

[18] Muhlenbeim, H. and Schlierkamp,D. (1993) Analysis of Selection, mutation and recombination in genetic algorithms. tech report. 93-24.

[19] Ochoa, G., Harvey, I.(1998). Recombination and error thresholds in finite populations. In Banzhaf, W. and Reeves, C., (eds), *FOGA-5* San Fransisco, CA. Morgan Kaufmann.

[20] Ochoa, G., Harvey, I. and Buxton, H. (1999) On Recombination and Optimal Mutation Rates. In GECCO 1999: *proceedings of the Genetics and Evolutionary Computation Conference.* Banzhat, W., Daida, J., & Eiben, A.E. (eds) Morgan Kaufman

[21] Ochoa, G., Harvey, I. and Buxton, H. Recombination and Error Thresholds in Finite populations. In *FOGA-5* Morgan Kaufmann.

[22] Schaffer, J., Caruana, R., Eshelman, L., and Das, R. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimisation. In Schaffer, J.D., (ed) *ICGA 3*, San Mateo CA. Morgan Kaufmann.

[23] Spears, W.M. (1998) *The Role os Mutation and Recombination in evolutionary Algorithms* Phd. Thesis, George Morgan Iniversit, Fairfax, Virginia

[24] Wolpert, D.H., and Macready, W.G., (1997) No Free Lunch Theorum for optimisation. In *IEEE Transactionon Evolutionary Computation*, 1(1).

Appendix 1
Example output of Hashtable for N = 4 K = 2

```
Hashtable: N = 4 K = 2

0       0.510946 0.044487 0.059357 0.959725
1       0.257903 0.411568 0.435447 0.491216
2       0.804151 0.940436 0.273011 0.690617
3       0.966860 0.928621 0.779307 0.993503
4       0.864460 0.842180 0.875743 0.003698
5       0.320475 0.227893 0.127966 0.635086
6       0.820596 0.369551 0.038560 0.827571
7       0.533508 0.415474 0.703155 0.212820


population:

0101
0110
0111
1000
1110
1100


Fitness:

fitness of 0    0.519874
fitness of 1    0.307196
fitness of 2    0.694956
fitness of 3    0.549226
fitness of 4    0.513995
fitness of 5    0.675843



-----------------------------------------------
```

Appendix 2

Mean Best-so-far fitness and standard deviations N = 100 K = 0 population size 100 GA with Recombination

generation	**average bsof**	**sd**
mutation rate **0.001**		
0	0.540561	0.010468
100	0.657155	0.008879
200	0.658338	0.008700
300	0.658510	0.009110
400	0.658600	0.009110
500	0.658635	0.009075
600	0.658658	0.009075
700	0.658660	0.009075
800	0.658675	0.009075
900	0.658679	0.009075
1000	0.658679	0.009075
mutation rate **0.05**		
0	0.547314	0.001216
100	0.672429	0.009103
200	0.673452	0.008894
300	0.673452	0.008894
400	0.673452	0.008894
500	0.673452	0.008894
600	0.673452	0.008894
700	0.673452	0.008894
800	0.673452	0.008894
900	0.673452	0.008894
1000	0.673452	0.008894
mutation rate **0.01**		
0	0.540561	0.010468
100	0.657155	0.008879
200	0.658338	0.008700
300	0.658510	0.009110
400	0.658600	0.009110
500	0.658635	0.009075
600	0.658658	0.009075
700	0.658660	0.009075
800	0.658675	0.009075
900	0.658679	0.009075
1000	0.658679	0.009075
mutation rate **0.05**		
0	0.549678	0.035644
100	0.621803	0.037528
200	0.626542	0.038327
300	0.627174	0.032561
400	0.627780	0.031133
500	0.628723	0.040620
600	0.629031	0.040620
700	0.629334	0.040620
800	0.629347	0.040620
900	0.629347	0.040620
1000	0.629784	0.040620

Appendix 3

Mean Best-so-far fitness and standard deviations N = 100 K = 0 population size 100 GA-m without Recombination

generation	**average bsof**	**sd**
mutation rate **0.001**		
0	0.528859	0.004801
100	0.608875	0.001598
200	0.634452	0.000353
300	0.635423	0.001246
400	0.635423	0.001246
500	0.635423	0.001246
600	0.635423	0.001246
700	0.635423	0.001246
800	0.635423	0.001246
900	0.635423	0.001246
1000	0.635423	0.001246
mutation rate **0.005**		
0	0.553865	0.001310
100	0.670748	0.005674
200	0.680865	0.003829
300	0.680865	0.003829
400	0.680865	0.003829
500	0.680865	0.003829
600	0.680865	0.003829
700	0.680865	0.003829
800	0.680865	0.003829
900	0.680865	0.003829
1000	0.680865	0.003829
mutation rate **0.01**		
0	0.551007	0.001806
100	0.639815	0.002819
200	0.647939	0.000547
300	0.648263	0.002864
400	0.650447	0.002864
500	0.650447	0.002864
600	0.650447	0.002864
700	0.650447	0.002864
800	0.650447	0.002864
900	0.650447	0.002864
1000	0.650447	0.002864
mutation rate **0.05**		
0	0.545801	0.000702
100	0.610863	0.001392
200	0.615785	0.002317
300	0.618977	0.002197
400	0.618977	0.002197
500	0.620127	0.002713
600	0.620127	0.002713
700	0.620127	0.002713
800	0.620127	0.002713
900	0.620127	0.002713
1000	0.620127	0.002713

Appendix 4

Average beset-so-far plots for all population sizes tested

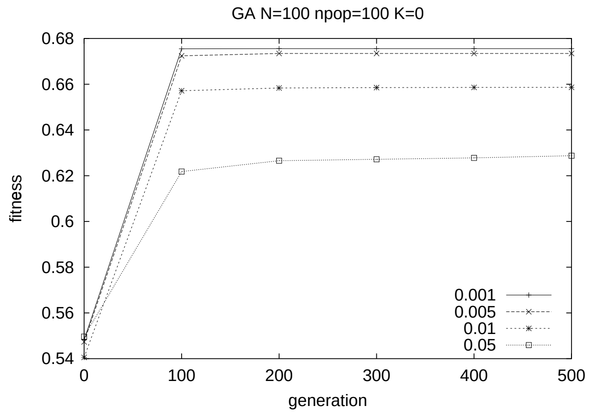

Figure 20: Average best-so-far curves for GA population size 100

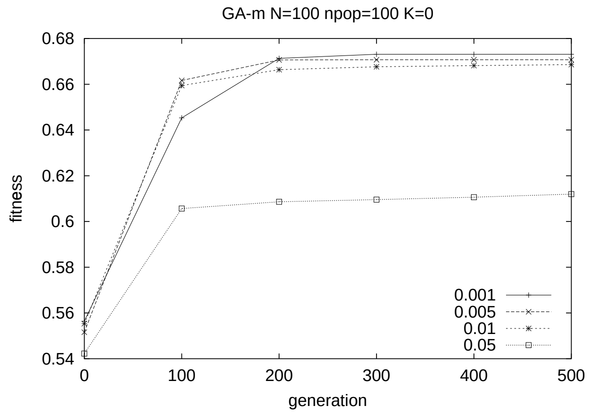

Figure 21: Average best-so-far curves for GA-m for population size 100

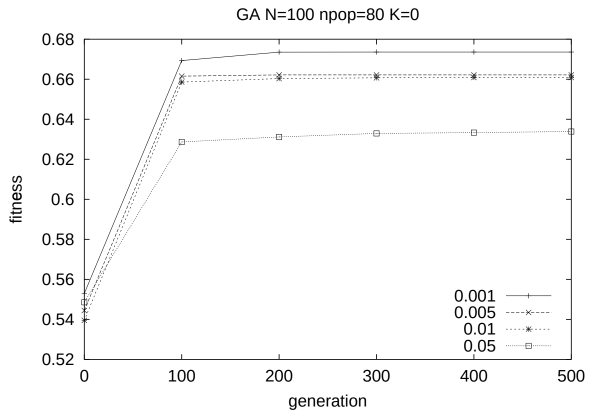

Figure 22: Average best-so-far curves for GA population size 80

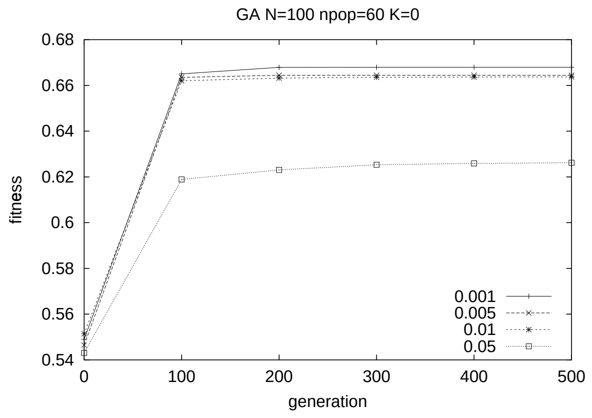

Figure 23: Average best-so-far curves for GA for population size 60

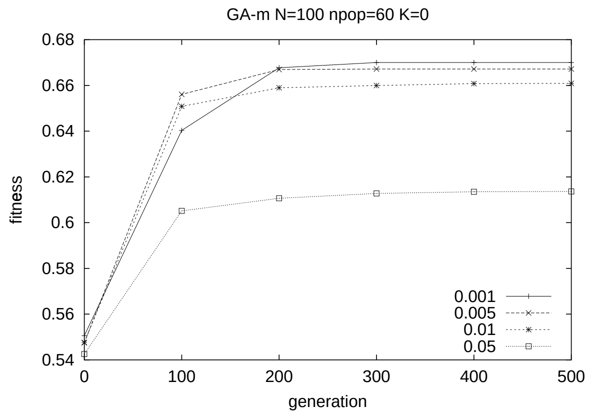

Figure 26: Average best-so-far curves.GA-m population size 60

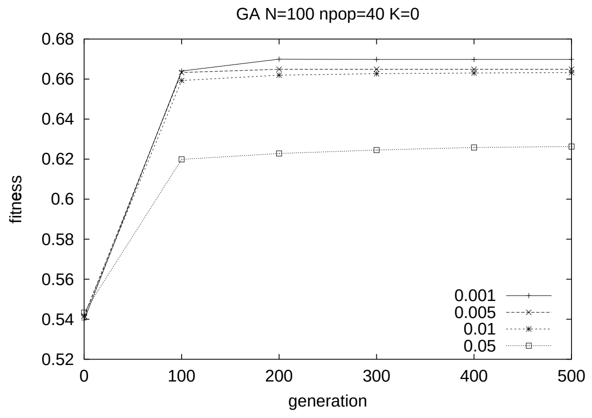

Figure 24: Average best-so-far curves for GA for population size 40

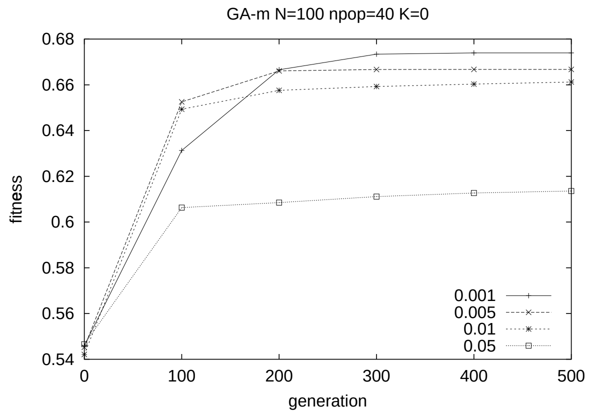

Figure 27: Average best-so-far curves for GA-m for population size 40

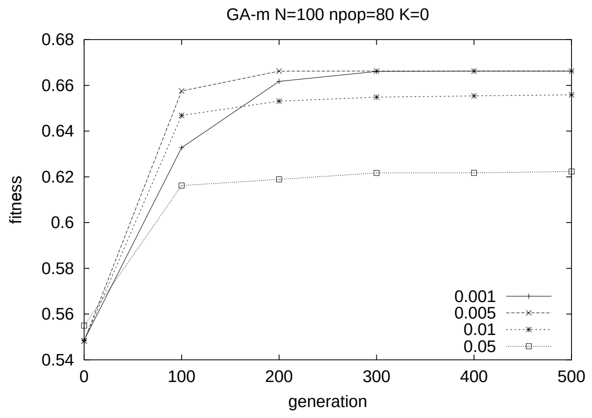

Figure 25: Average best-so-far curves. GA-m population size 80

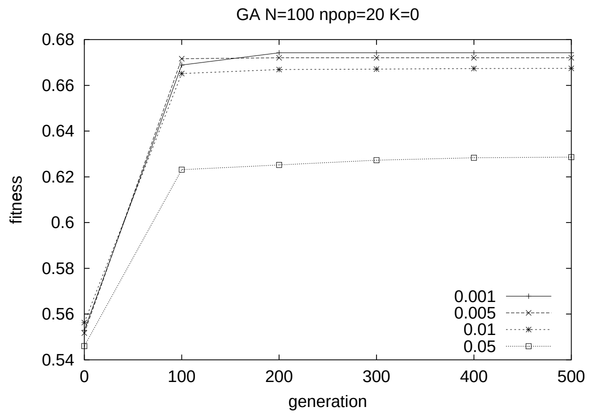

Figure 28: Average best-so-far curves for GA for population size 20

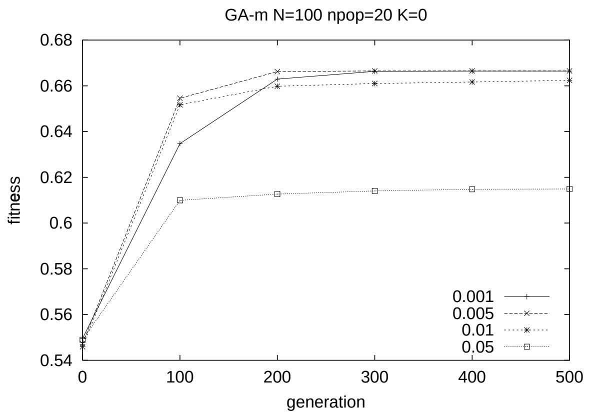

Figure 29: Average best-so-far curves for GA-m for population size 20

Appendix 5

Source code for GA and problem generator used.

```
//*******************************************************

//The data processing section is included, but is a bit of a 'rough',
//Apologies.
//I have also included the original Roulette wheel selection procedure




#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>

//*****structures for Look up tables for NK generator******

typedef struct
{
int key;
double *value;
  int ngen;
} HashVal;

typedef struct
{
HashVal **rows;
int nop;
int k;
} HashTab;

//*****structure for each individual******

typedef struct
{
  int len;
  int *chrom;
  double fit;
  double expval;

}Ind;
```

```
//*****structure for pop******

typedef struct
{
  Ind **ind;
  int npop;
  double sumev;
} Pop;

#define NPOP 100
#define GEN 10
#define RNDSD 234
#define INTERVAL 1
#define NOINTERVAL (GEN/INTERVAL)
#define NPROB 2
#define NTOURS 4
//*****Hashtable and population initialisation and evaluation*****

HashTab *initHashTab(int n, int k);
double getval(HashTab *ht,int pat,int gene);
int getpat(int *chrom, int n, int k, int gene);

Pop *initpop(int len, int npop);
void randPop(Pop *popu);
double evalChrom(HashTab *ht, int *chrom, int n);
double evalPop(Pop *popu, HashTab *ht);

//*****GA operators*****

void iterateGA(HashTab *ht,int gen,int n,int npop,double* best, double pm,
 double pc);
void tourselnPar(Pop *popu,int npar, Ind **par);
void cross(Ind *par1, Ind *par2, Ind **kid1, Ind **kid2, double pc);
void mut(Ind *kid,double pm);

//*****original roulette wheel*****
//void selnPar(Pop *popu,int npar, Ind **par);

void freeInd(Ind *ind);
void freePop(Pop *popu);
void freeHashTab(HashTab *ht);

//*****Debugging******
void printPop(Pop *popu);
void printInd(Ind *ind);
void printHT(HashTab *ht);

int main (int argc, char **argv)
{
  int i,j,n,k,m;
  double **best; //2d array for best of each gen for each prob
  double bsof[NOINTERVAL+1]={0};
  double pm,pc,tbsof=0.0;
  char *filename;

  time_t t;
  HashTab  *ht;
  FILE *fp;
  if (argc!=6)
    {
     printf("re-enter command line alga n k pm pc fname\n");
     exit(EXIT_FAILURE);
    }
  n = atoi(argv[1]);
  k = atoi(argv[2]);
  pm  = atof(argv[3]);
  pc = atof(argv[4]);
  filename = (argv[5]);

  srand(time(&t));
  srand48(time(&t));

  best = calloc(NPROB, sizeof(double *));
  for(i=0;i<NPROB;i++)
   *(best+i) = calloc(GEN, sizeof(double));

  for(i=0;i<NPROB;i++)
    {
    ht = initHashTab(n,k);
iterateGA(ht,GEN, n, NPOP, *(best+i), pm, pc);
  freeHashTab(ht);
    }


//*****DATA PROCESSING - calculates mean bsof at INTERVAL across GEN
//samples at every INTERVAL, increments m up to GEN/INTERVAL

  for(i=0;i<NPROB;i++)
  {
      tbsof=0.0;
      m=0;
      for(j=0;j<=GEN;j++)
      {
       if(*((*(best+i))+j)> tbsof)
           tbsof = *((*(best+i))+j);
        if((j%(INTERVAL))==0)
{
   bsof[m]+=(tbsof/NPROB);
   m++;
        }
       }

  }

  fp = fopen(filename, "w");
  fprintf(fp,"# N=%d,K=%d,pm=%f,pc=%f, popsize=%d fpsel\n", n, k, pm, pc,NPOP);
  for(i=0;i<=NOINTERVAL;i++)
   {
      fprintf(fp,"%d\t\t%f\n",i*INTERVAL,bsof[i]);
    }
  fclose(fp);
  exit(EXIT_SUCCESS);
}


//*****   Controls Generation loop   *****
//takes as arguments: pointer to hashtable, no of generations,
//N(chromosomelenght), population size, pointer to array to store 'best'
//individuals in and probabilities of crossover and mutaiton

void iterateGA(HashTab *ht,int gen,int n,int npop,double *best,double pm,double
pc) {
  Pop *popu;
  Ind **par; //temp arrays for generation cycle
  Ind **kid;
  int i,j;

  popu = initpop(n, npop);
  randPop(popu);
  par = calloc(npop,sizeof(Ind*));
  kid = calloc(npop, sizeof(Ind*));
//GENERATION LOOP
```

```
  for(i=0;i<gen;i++)
  {
best[i] = evalPop(popu,ht);
      tourselnPar(popu, npop, par);
for(j=0;j<npop;j+=2)
        {
        cross(*(par+j), *(par+j+1),kid+j, kid+j+1,pc);
        mut(*(kid+j), pm);
        mut(*(kid+j+1),pm);
        }

      for(j=0;j<npop;j++)
        {
        freeInd(*(popu->ind+j));    //releases old population
        *(popu->ind+j) = *(kid+j);  //inserts new population
    }

  }

  free(par);
  free(kid);
  freePop(popu);

}

//*****   Selects 2 parents at time using tournament selection size NTOURS*****
//takes pointer to the population structure, poulation size and pointer to
//pointer to 'parent' array used in generational loop.
//places two individuals at a time into parent array

 void tourselnPar(Pop *popu,int npar, Ind **par)
{
   register unsigned int i, j, k;
   unsigned int cp1, cp2, ptmp;

   k = 0;
   j = npar / 2;
   while(j--)
   {
cp1 = rand() % popu->npop;
cp2 = rand() % popu->npop;
if(((*(popu->ind+cp1))->fit) < ((*(popu->ind+cp2))->fit))
{
ptmp = cp1;
cp1 = cp2;
cp2 = ptmp;
}
i = NTOURS - 2;
while(i--)
{
ptmp = rand() % popu->npop;
if(((*(popu->ind+ptmp))->fit) > ((*(popu->ind+cp2))->fit))
{
if(((*(popu->ind+ptmp))->fit) > ((*(popu->ind+cp1))->fit))
{
cp2 = cp1;
cp1 = ptmp;
}
else
{
cp2 = ptmp;
}
}
}

*(par + (k++))= *(popu->ind + cp1);  //places fittest two individuals
  *(par + (k++)) = *(popu->ind + cp2); //in parent array.
   }



}
//*****Roulettewheel selection procedure*****

void selnPar(Pop *popu,int npar, Ind **par)
{
  int i,j;
  double pin;
  double sum;
  for(i=0;i<npar;i++)
    {
     sum = (*(popu->ind))->expval;
       pin = drand48()*popu->sumev;
       for(j=0;j<(popu->npop);j++)
if (sum<pin)
sum+=(*(popu->ind+j))->expval;
else
           break;
       if (j==popu->npop)      //safeguard to avoid seg fault
j--;
     *(par+i) = *(popu->ind+j);
    }

}

//*****  Tests probability at each locus and mutates accordingly  *****

void mut(Ind *kid,double pm)
{

int i;

   for(i=0;i<kid->len;i++)
     if (drand48()<pm)
       *(kid->chrom+i)=1-(*(kid->chrom+i));

}
//***** 2 point crossover *****
//takes *p to two parents and *p to *p to two children and crossover prob
//allows possibility of one-point (if loci are the same or one is at an end)
//ensures crossover ALWAYS occurs if probability is exceeded (makes sure one is
//NOT an end)
//places 'children' in kid array

 void cross(Ind *par1, Ind *par2, Ind **kid1, Ind **kid2, double pc)
{
  int i,loc1,loc2,locx;

  *kid1 = malloc(sizeof(Ind));
  *kid2 = malloc(sizeof(Ind));
  (*kid1)->chrom = calloc(par1->len,sizeof(int));
  (*kid2)->chrom = calloc(par2->len,sizeof(int));
  (*kid1)->len = par1->len;
  (*kid2)->len = par2->len;

  if (drand48()<pc)
  {
    do{
    loc1 = rand() % par1->len;
    } while((loc1 == 0) || (loc1 == (par1->len-1)));

    loc2 = rand() %par1->len;
    if(loc1>loc2)
    {
      locx=loc1;
      loc1=loc2;
      loc2=locx;
    }
    for(i=0;i<loc1;i++) //crosses from start to locus 1
      {
      *((*kid1)->chrom+i) = *(par1->chrom+i);
      *((*kid2)->chrom+i) = *(par2->chrom+i);
      }
    for(i=loc1;i<=loc2;i++) //crosess middle section
      {
      *((*kid1)->chrom+i) = *(par2->chrom+i);
      *((*kid2)->chrom+i) = *(par1->chrom+i);
      }
    for(i=(loc2+1);i<(*kid1)->len;i++)          //crosses locus 2 to end
      { //ensuring bit AT locus is swapped
      *((*kid1)->chrom+i) = *(par1->chrom+i);  //incase it is an end
      *((*kid2)->chrom+i) = *(par2->chrom+i);
      }
  }
    else
    {
      for(i=0;i<(*kid1)->len;i++)
      {
      *((*kid1)->chrom+i) = *(par1->chrom+i);
      *((*kid2)->chrom+i) = *(par2->chrom+i);
      }
    }

}


//*****creates and initialises look up table with random values
//**takes N and K as arguments, returns a pointer to hashtable
//**creates table of size N by k^(2+1) (ngen by nop)
//**fills with uniformly distributed random numbers

 HashTab *initHashTab(int n, int k)
 {
   HashTab *ht; //
   HashVal *hv;
   int i,j;
   ht=malloc(sizeof(HashTab));
   ht->k=k;
   ht->nop=pow(2,k+1);
   ht->rows=calloc(ht->nop,sizeof(HashVal *));

   for (i=0;i<ht->nop;i++)
     {
    hv=malloc(sizeof(HashVal));
hv->key=i;
hv->ngen=n;
hv->value=calloc(hv->ngen,sizeof(double));
for(j=0;j<hv->ngen;j++)
*(hv->value+j)=drand48();   //fills array in hv
*(ht->rows+i)=hv;  //puts *p to hv into rows
      }


   return(ht);

}

//*****Initialises population*****
//Takes N and population size as arguments, allocates memory for population
//returns pointer to population structure

Pop *initpop(int len, int npop)
{
   Ind *ind;   //temp array for init stage
   Pop *p_pop;
   int i;

   p_pop = malloc(sizeof(Pop));
   p_pop->ind = calloc(npop,sizeof(Ind *));
   p_pop->npop = npop;
   for(i=0;i<npop;i++)
     {
     ind = malloc(sizeof(Ind));
     ind->len = len;
     ind->chrom = calloc(len,sizeof(int));
     *(p_pop->ind+i) = ind;
    }
  return (p_pop);
}
```

```
//*****fills population wiht random 0s and 1s*****

void randPop(Pop *popu)

{
  int i,j;

  for (i=0;i<popu->npop;i++)
   for(j=0;j<((*(popu->ind+i))->len);j++)
        *((*(popu->ind+i))->chrom+j)=rand()%2;
}

//*****Evaluates Population (calls evalchrom() npop times)*****
//Takes pointer to population and hastable structures
//keeps track of and returns 'best' individual

double evalPop(Pop *popu, HashTab *ht)
{
  int i;
  double best=0.0;
  double fitsum=0.0;


  for(i=0;i<(popu->npop);i++)
    {
   (*(popu->ind+i))->fit =
   evalChrom(ht,(*(popu->ind+i))->chrom,(*(popu->ind+i))->len);
     fitsum+=(*(popu->ind+i))->fit;
     if( ((*(popu->ind+i))->fit)>best)
        best = (*(popu->ind+i))->fit;
    }

  //expected fitness values for use in Roulette Wheel selection
  popu->sumev = (double)popu->npop;
  fitsum /= (double)popu->npop;
  for(i=0;i<(popu->npop);i++)
    (*(popu->ind+i))->expval = ((*(popu->ind+i))->fit)/fitsum;

   return(best);
}

//*****Evaluates each chromosome*****
```

```
//Takes pointers to hashtable, specific chromosome and chromosome length
//returns fitness of chromosome (mean fitness contributoin of each locus)

double evalChrom(HashTab *ht, int *chrom, int n)
{
   int pat,i;
   double chromval=0.0;

   for(i=0;i<n;i++)
     {
    pat=getpat(chrom,n,ht->k,i);
chromval+=getval(ht,pat,i);
      }

   return(chromval/n);
}

//***** determines integer representation of pattern of interacting loci*****
 //takes specific chromosome, n, k, and specific locus
//returns integer rep. of 'pattern' (used to reference correct row of hashtable)

int getpat(int *chrom, int n, int k, int gene)
{
   int i,g,pat=0;
   int f=pow(2,k);

   g=gene-(k/2);
   if(g<0)      //enables wraparound for genes at locus[0]
g+=n;
for(i=0;i<(k+1);i++)
{
pat+=chrom[g]*f;
f/=2;
g++;
if(g>(n-1))
g=0; //ends wraparound
}
return(pat);
}

//****************retrieves relevant value from hashtable**********************

//takes pointer to hashtable, and row and column values
//returns relevant fitness value
```

```
double getval(HashTab *ht,int pat,int gene)
 {
   return(*((*(ht->rows+pat))->value+gene));
 }

//**************releases individuals of old population each generation
// before 'children' are inserted
void freeInd(Ind *ind)
{

  free(ind->chrom);
  free(ind);

}

//*****releases population memory*****

void freePop (Pop *popu)
{
  int i;

  for(i=0;i<popu->npop;i++)
    {
    freeInd(*(popu->ind+i));
    }
  free(popu->ind);
  free(popu);
}

//****Releases hashtable after each algorithm run******

void freeHashTab(HashTab *ht)
{
  int i;
  for(i=0;i<ht->nop;i++)
    {
     free((*(ht->rows+i))->value);
     free((*(ht->rows+i)));
    }
  free(ht->rows);
  free(ht);

}
```

```
///////////DEBUG FUNCTIONS//////////////

//*****prints population (0s and 1s)*****
void printPop(Pop *popu)
{
  int i;

  for (i=0;i<popu->npop;i++)
    {
   printf("ind %d: ",i);
   printInd(*(popu->ind+i));
     }

}

//*****prints fitness and expected value*****

void printInd(Ind *ind)
{
  int j;

  printf("fit:%f\t(%f) ", ind->fit, ind->expval);
  for(j=0;j<ind->len;j++)
       printf("%d",*(ind->chrom+j));
  printf("\n");


}


//*****prints hashtable*****

void printHT(HashTab *ht)
{
   int i,j;
   for(i=0;i<ht->nop;i++)
     {
    printf("%d\t",i);
for(j=0;j<((*(ht->rows+i))->ngen);j++)
printf("%5.2f ",*((*(ht->rows+i))->value+j));
printf("\n");
      }
}
```