Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing (SAC'93), Indianapolis, IN., February 14-16, 1993, pp. 118-125.

MANIPULATING SUBPOPULATIONS OF FEASIBLE AND INFEASIBLE SOLUTIONS IN GENETIC ALGORITHMS*

D. Ansa Sekharan
Roger L. Wainwright

Department of Mathematical and Computer Sciences
The University of Tulsa

Abstract

This paper explores the partitioning of the population pool in genetic algorithms into separate subpopulations of feasible and infeasible solutions, and the interaction on a regular basis of crossover operations among and within the subpopulations. The set covering problem was chosen as a representative optimization problem to apply our subpopulation strategies. We designed a class of nine algorithms for manipulating the two population pools and compared this against the traditional GA. The traditional GA uses a single population pool where infeasible solutions are generally considered infrequently or ignored. All of our algorithms significantly and consistently outperformed the traditional GA in all of the test problems illustrating the importance of infeasible solutions as a source of good genetic material. Furthermore, results show that the random select consistently outperformed the bias select from the infeasible pool suggesting all infeasible solutions should be considered equal.

Introduction

The class NP of problems denotes the set of all decision problems solvable by a non-deterministic polynomial time algorithm. The class P of problems denotes the set of all decision problems solvable by a deterministic polynomial time algorithm. According to Cooke's Theorem, every problem in NP can be transformed into the Boolean satisfiability problem (SAT). Only those problems in NP for which the reverse transformation exists are considered equally "hard" problems and define the class of NP-complete problems. The class of NP-complete problems are, in theory, considered computationaly equivalent. NP-complete problems have no known deterministic polynomial time algorithms; that is, there is no known solution except to try all combinations [1,14]. These problems are sometimes referred to as combinatorial optimization problems.

It is impossible to optimally solve any of these problems, except for trivial cases. Finding a solution requires an organized search through the problem space. An unguided search is extremely inefficient. Consequently, research has focused on approximation techniques which provide efficient, near optimal solutions. Some of these techniques which are applicable to NP-complete problems include heuristic techniques, simulated annealing, neural networks, and genetic algorithms.

In this paper we consider genetic algorithms as a technique for solving combinatorial optimization problems. This paper explores the partitioning of the population pool into subpopulations of feasible and infeasible solutions, and the interaction of crossover operations among and within the subpopulations. This work has not been investigated before. The set covering problem (SCP) was chosen as a representative combinatorial optimization problem to apply our subpopulation strategies. The SCP problem was chosen because many practical applications can be expressed as a SCP problem. These include information retrieval, graph coloring, various AI applications, VLSI logic design, operations research, assignment problems, scheduling problems such as assembly line scheduling and airline crew scheduling, design of computer systems, political districting, circuit simulation, etc. [2].

The Set Covering Problem

The set covering problem has been shown to be NP-complete. In fact, it is one of the "core" NP-complete problems. Note the vertex covering problem is a special case of the SCP problem. The SCP problem is the problem of finding the minimum number of columns in a Boolean matrix such that all rows of the Boolean matrix are "covered" by at least one element from any column and the sum of the costs associated with the covering columns is optimal (minimum cost in our case). A Boolean matrix is a rec-

* Research partially supported by OCAST Grant ARO-038 and Sun Microsystems Inc.

ACM-SAC '93/2/93/IN, USA
© 1993 ACM 0-89791-568-2/93/0002/0118...$1.50

tangular matrix of zeros and ones where a covered row is denoted by a one in the covering columns. There is no known algorithm for the optimal solution except to try all possibilities. This involves trying all combinations of the subsets of columns. For a matrix with n columns, the number of combinations to try is the power set or 2^n. Consider Figure 1 which shows an example Boolean matrix with $m = 7$ rows and $n = 8$ columns. Columns 1, 2, 3 and 4 form a cover with a cost of 15; columns 1, 4, 7 and 8 also form a covering with a cost of 19. The optimal cost is 7 formed by columns 3, 4, and 6.

As a practical application of the SCP problem, consider the airline crew scheduling problem. An airline has m flights to schedule. This is represented as the m rows in a Boolean matrix such as Figure 1. The airline has n flight crews depicted by the columns in Figure 1. The Boolean entries in the matrix indicates which flights each crew is able to service and at what cost. The optimal solution schedules flight crews 3, 4, and 6 to service all seven flights at a cost of 7 units. An excellent formal description of the SCP problem is given by Moret and Shapiro [11].

Genetic Algorithms

Genetic algorithms(GA) are based on the principles of natural genetics and survival of the fittest. Genetic algorithms search for solutions by emulating biological selection and reproduction. In a GA the parameters of the model to be optimized are encoded into a finite length string, usually a string of bits. Each parameter is represented by a portion of the string. The string is called a chromosome, and each bit is called a gene. Each string is given a measure of "fitness" by the fitness function, sometimes called the objective or evaluation function. The fitness of a chromosome determines its ability to survive and reproduce offspring. The "least fit" or weakest chromosomes of the population are displaced by more fit chromosomes.

The genetic algorithm is a robust search and optimization technique using probabilistic rules to evolve a population from one generation to the next. The transition rules going from one generation to the next are called genetic recombination operators. These include Reproduction (of the more "fit" chromosomes), Crossover, where portions of two chromosomes are exchanged in some manner, and Mutation. Crossover combines the "fittest" chromosomes and passes superior genes to the next generation thus providing new points in the solution space. Mutation is performed infrequently. A new individual (point in the solution space) is created by altering some of the bits of an individual. Mutation ensures the entire state space will eventually be searched (given enough time), and can lead the population out of a local minima. Genetic algorithms retain information from one generation to the next. This information is used to prune the search space and generate plausible solutions within the specified constraints [3,12].

Genetic algorithm packages for a single processor have been available for several years. A steady-state GA such as GENITOR [15] and a generational GA such as GENESIS [10] are the two example packages. Goldberg and others provide an excellent in depth study of genetic algorithms [6,7,9,13]. Furthermore, several researchers have investigated the benefits of using genetic algorithms for solving combinatorial optimization problems [3-8,12,16].

GA Subpopulations Applied To SCP

Consider a SCP problem with m rows and n columns. A Chromosome defining a possible covering is depicted as a bit string of length n. The ith bit of the chromosome corresponds to the ith column of the Boolean matrix representing the problem. A one bit means the column is included in the covering, and a zero bit means the column is not included in the covering. There are obviously 2^n possible chromosomes. A feasible solution is defined as any covering of the rows of the Boolean matrix. For example, in Figure 1, the bit strings (11110000), (10010011), and (11111111) are feasible solutions. An infeasible solution is represented by a bit string that does not define a covering, such as (01001000), (10101001), (01010111), and (00000000). Hence, the optimal solution is a feasible solution with the minimum cost.

The traditional GA uses a single population pool. The initial population (either seeded or generated randomly) can contain both feasible and infeasible solutions. In some GA's infeasible solutions are discarded and not placed in the pool. Infeasible solutions are generally given an extremely poor or non-existent fitness value, hence they rarely, if ever, participate in the genetic recombination operations. In some instances, feasible solutions may be difficult to initially generate due to the type of problem under investigation. For example, given a sparse Boolean matrix representing the SCP, the initial randomly generated population may have very few, if any, feasible solutions. The traditional single pool GA could be made to manipulate infeasible solutions if the fitness function were properly adjusted. However, as generations continue and more and more feasible solutions are generated, the infeasible solutions are considered more infrequently or not at all.

In this paper, we studied the effect of maintaining separate population pools of feasible and infeasible solutions and periodically involving infeasible solutions in crossover operations regardless of their fitness value. We believe infeasible solutions often contain an excellent source of genetic material that should not be overlooked. For example, in Figure 1 consider the feasible solution (01011011) with a cost of 27 versus the infeasible solution (00100100) where a cost is not defined since a covering does not occur.

The optimal solution is (00110100) which has a hamming distance of one from the infeasible solution and a hamming distance of 6 from the feasible solution. In this case the infeasible solution is much "closer" to the optimal solution than the feasible solution. There are several reasons one might want to consider infeasible solutions in the population pool. First, it may be extremely difficult to generate a feasible solution, perhaps as difficult as generating the optimal solution. Secondly, involving infeasible solutions in the search insures diversity in the search space and allows for a more robust algorithm. Furthermore, if the solution space is disjoint or non-convex, a more direct path to an optimal solution might be to traverse through a section of the search space containing infeasible solutions.

In our GA model, feasible and infeasible solutions are maintained in separate subpopulation pools of equal size of $n/2$. The total population of size n at each generation remains constant. We used a steady-state GA, generating one child at a time and removing the worst chromosome. We used GENITOR which was modified for our particular application. The initial population is generated randomly by placing feasible and infeasible solutions into their proper pool until one pool has been filled. Thus, the other pool may not be completely filled initially, but is allowed to grow up to size $n/2$ as future children are generated.

The evaluation function for the feasible solution pool is the cost of the covering. We define *rnc* as the number of "rows not covered". The evaluation function for infeasible solutions is $rnc * k + cost$ (of the partial covering), where k is some constant larger than the sum of the costs of all of the columns. In this way the infeasible solutions are ranked first by *rnc* then secondly by the *cost* of the partial covering. In all cases we used the uniform crossover operator. The mutation operation consists of flipping a randomly selected bit in a string.

We define an *X*-crossover as one that involves two chromosomes from the feasible pool. A *Y*-crossover involves one chromosome from the feasible pool and one from the infeasible pool. A *Z*-crossover involves two chromosomes from the infeasible pool. In all cases the resulting feasible solutions are placed into the feasible pool and the resulting infeasible solutions are placed into the infeasible pool.

Results and Conclusions

We have designed a class of algorithms for manipulating the two population pools. Each algorithm is define in the form of a triple (x,y,z), where x, y, z represents the percentage of time the X, Y, and Z-crossovers are performed, respectively. Hence algorithm *(95, 4, 1)* means that for any given crossover operation, there is a 95% chance that two chromosomes from the feasible pool will be selected, a 4% chance one chromosome from each of the pools will be selected, and 1% chance that two chromosomes from the infeasible pool will be selected.

We have also developed a class of oscillation algorithms that varies x and y percentages with each new trial. As an example, the first oscillation algorithm (*OSC1*) holds z constant at 1% and oscillates x from 75 to 99, then back down to 75, and so forth in increments and decrements of 1. A second oscillation algorithm (*OSC2*) holds z constant at 1% and oscillates x from 60 to 85, then back down to 60, and so forth in increments and decrements of 1. In all cases, the y percent is such that $x+y$ is always 99. The x, y, z values for the oscillation algorithms were selected arbitrarily.

To test these algorithms we generated SCP problems of size 100x100. We generated various sparse Boolean matrices consisting of 5, 8, 12, and 16% random fill (of ones). The weights of each column were randomly generated in the range of 1 to 20. For each of the four test matrices we ran the following GA algorithms: *(95,4,1)*, *(90,9,1)*, *(85,14,1)*, *(80,19,1)*, *(75,24,1)*, *(70,29,1)*, *(65,34,1)*, *OSC1*, *OSC2*. In each of these algorithms, the x, y, z values were selected arbitrarily. These algorithms were designed to test the effect of using the infeasible population pool at different rates of involvement. We compared these results with the traditional single pool GA (Tradnl). The fitness function described in the previous section was used in all of the algorithms.

Table I and Table II show the results of solving the 5% random fill 100x100 SCP problem. The pool size was 600, and the uniform crossover operator was used. Since we chose $z = 1$, the algorithms are represented in the Tables in form of a (x,y) tuple. We ran each algorithm on 10 different test cases. Entries show the resulting cost of the covering. Table I reports the results using a bias selection from the infeasible pool. The bias function was described in the previous section. Table II reports the results when using a random selection from the infeasible pool. The *Best Result* row in Table I and Table II indicates how many times out of 10 the algorithm generated the best solution. In many cases several algorithms generated the best solution. We have no way of knowing if this was the optimal solution or not. The *Beat Tradnl* row in Table I and Table II indicates how many times out of 10 a given algorithm obtained the same or better results than the traditional GA(*Tradnl*).

In a similar manner, Table III and Table IV show the results of the 8% random fill 100x100 SCP problem. Table V and Table VI show results of the 12% random fill 100x100 SCP problem, and Table VII and Table VIII show the results of the 16% random fill 100x100 SCP problem. We did not consider a random fill problem less that 5% because our tests showed it was rarely possible to determine any feasible solution, even after several generations.

Furthermore, we did not include tests for a random fill problem over 16%, because in all of the cases no infeasible solutions were ever generated, making our algorithms inappropriate.

Consider the *Beat Tradnl* row in Table I through Table VIII. Our nine algorithms performed about the same as the traditional GA in Table I (5% fill). Our algorithms perform better than the traditional GA in Table II, suggesting there may be some importance to random selection over bias selection in the infeasible pool. Note, however, the excellent performance of all of our nine algorithms as shown in the remaining Tables. All of our algorithms significantly and consistently outperformed the traditional GA in Tables III through Tables VIII (8%, 12% and 16% fill).

We also compared the difference in executing each problem using a bias selection from the infeasible pool, and a random selection. That is, in each of the problems (5%, 8%, 12%, 16% fill) we compared corresponding entries of our nine algorithms (Table I versus Table II, Table III versus Table IV, Table V versus Table VI, and Table VII versus Table VIII). The results are shown in Table IX. Notice in all four problems the random select outperformed the bias select from the infeasible pool. In fact, random outperformed bias about two to one, except in the 12% case. When a bias selection was used, perhaps several of the infeasible solutions were used over and over to the exclusion of others. The goal of using infeasible solutions is to find a short cut through the infeasible search space to a feasible solution space where, perhaps, an optimal solution is located. Our results suggest, in this case, that all infeasible solutions should be considered equal.

It is possible for other problems that random selection from the infeasible pool may be no better or perhaps worse than a bias selection. However, if random selection from the infeasible pool for a given application is better than a bias selection, then there is no reason to maintain an expensive separate infeasible pool. At some given rate crossover could be performed with a feasible solution and a newly created string (which may or may not be feasible). This could accomplish the same desired results at much less expense.

Table IX also records the percent of the infeasible and feasible solutions present in the initial population pools of the ten test cases for all four problem sizes. In the four problems (5%, 8%, 12%, 16% fill) the percent of infeasible solutions present in the initial population was 98.7%, 77.0%, 12.4%, and 1.0% respectively. This re-enforces the two main points of this paper: (1) The importance of random selection from the infeasible pool. In Table I, with only 1.3% feasible solutions initially, it was important to consider all of the infeasible solutions as possible crossover mates rather than the selected bias. (Table I versus Table II), and (2) the importance of considering infeasible solutions, even if there are very few of them. Consider Table VII and Table VIII. Only 1% infeasible solutions were generated initially. Yet their influence was extremely significant, yielding superior results over the traditional GA. The traditional GA, of course, with 99% feasible solutions generated initially, would never consider any of the infeasible solutions.

Finally, among the nine algorithms we tested, no one algorithm appears to be significantly better than any of the others. It is possible, however, for a different problem that one or several of the nine algorithms could be significantly superior to the others. This simply suggests if the problem one is solving generates infeasible solutions, then it is important to include the infeasible solutions in the crossover operations on some regular basis.

Future Research

We are currently evaluating other problems that yield a high percentage of infeasible solutions, such as partitioning problems. We have access to a parallel GA implemented for a hypercube. We will be investigating how infeasible solutions should be manipulated among parallel processors. Perhaps each processor should be executing a different (x,y,z) algorithm. There is certainly more work to be done on fine-tuning the relationship between x, y, and z in an algorithm. Determining the proper population size is another important issue in order to maintain an adequate initial mix of feasible and infeasible solutions. We are investigating dynamically determining the initial x, y, and z values for the algorithm based on the percent of feasible and infeasible solutions found in the randomly generated initial population pools. Finally, we are looking into dynamically altering x, y, and z values according to some criteria as the population matures.

The authors' address is Department of Mathematical and Computer Sciences, The University of Tulsa, 600 South College Avenue, Tulsa, Oklahoma 74104-3189,
sekhar@euler.mcs.utulsa.edu,
rogerw@penguin.mcs.utulsa.edu

Acknowledgements

This research has been partially supported by OCAST Grant ARO-038. The authors also wish to acknowledge the support of Sun Microsystems Inc. The authors wish to acknowledge referee #46 for many helpful suggestions and enhancements to the paper.

References

[1] S. Baase, "Computer Algorithms, Introduction to Design and Analysis", Addison-Wesley, 1988.

[2] Beard, R.A., Lamont, G.B., "Determination of Algorithm Parallelism in NP-Complete Problems for Distributed Architectures", *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, SC., April, 1990, pp.42-51.

[3] Blanton, J. L., and Wainwright, R. L., "Vehicle Routing with Time Windows using Genetic Algorithms" Proceedings of the Sixth Oklahoma Symposium on Artificial Intelligence, November, 1992, pp. 242-251.

[4] D.E. Brown, C.L. Huntley and A.R. Spillane, "A Parallel Genetic Heuristic for the Quadratic Assignment Problem", *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989.

[5] Corcoran, A.L. and Wainwright, R.L. "A Genetic Algorithm for Packing in Three Dimensions", Proceedings of the 1992 ACM/IEEE Symposium on Applied Computing, March 1-3, 1992, pp. 1021-1030.

[6] L. Davis, ed., *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann Publisher, 1987.

[7] L. Davis, ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.

[8] K.A. De Jong and W.M. Spears, "Using Genetic Algorithms to Solve NP- Complete Problems", *Proceedings of the Third International Conference on Genetic Algorithms*, June, 1989, pp. 124-132.

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

[10] J. Grefenstette, GENESIS, Navy Center for Applied Research in Artificial Intelligence, Navy research Lab., Wash. D.C. 20375-5000.

[11] Moret, B.M.E., Shapiro, H.D., *Algorithms from P to NP, Volume I Design & Efficiency*, Benjamin/Cummings, 1991.

[12] Mutalik, P.M., Knight, L.R., Blanton, J.L. and Wainwright, R.L. "Solving Combinatorial Optimization Problems Using Parallel Simulated Annealing and Parallel Genetic Algorithms", Proceedings of the 1992 ACM/IEEE Symposium on Applied Computing, March 1-3, 1992. pp. 1031-1038.

[13] G. Rawling, ed., *Foundations of Genetic Algorithms*, Morgan Kaufmann Publishers, 1991.

[14] R. Sedgewick, "Algorithms", Addision-Wesley, 1988.

[15] D. Whitney and J. Kauth, GENITOR: A Different Genetic Algorithm, *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, Denver, Co., 1988, pp. 118-130.

[16] D. Whitney, T. Starkweather, and D. Fuquat, "Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator", *Proceedings of the Third International Conference on Genetic Algorithms*, June, 1989.

Test Case	Algorithm									
	(95,4)	(90,9)	(85,14)	(80,19)	(75,24)	(70,29)	(65,34)	OSC1	OSC2	Tradnl
1	167	128	128	125	130	128	130	129	121	130
2	158	160	136	132	143	133	130	144	137	137
3	155	145	160	159	134	164	140	159	168	149
4	146	148	147	145	154	154	146	148	154	149
5	182	157	145	144	150	139	169	155	147	139
6	197	193	210	209	191	187	183	190	187	182
7	192	185	166	179	176	176	176	193	172	178
8	140	152	140	144	140	140	140	140	140	152
9	169	165	162	162	165	151	174	190	154	154
10	129	126	126	136	126	133	127	130	130	129
Best Result	1/10	1/10	2/10	1/10	3/10	3/10	2/10	1/10	3/10	2/10
Beat Tradnl	3/10	5/10	6/10	4/10	5/10	6/10	7/10	3/10	5/10	–

Table I: Results of 5% Fill 100x100 SCP. Pool Size = 600
Uniform Crossover, Bias Infeasible Selection

Test Case	Algorithm									
	(95,4)	(90,9)	(85,14)	(80,19)	(75,24)	(70,29)	(65,34)	OSC1	OSC2	Tradnl
1	131	128	134	123	128	126	123	121	126	130
2	145	131	131	131	131	131	131	135	129	137
3	144	134	134	128	138	129	137	153	145	149
4	146	164	148	157	152	163	148	149	147	149
5	160	162	154	145	148	152	152	148	148	139
6	181	183	191	183	192	182	182	182	181	182
7	167	185	181	167	170	168	179	173	166	178
8	142	140	140	140	140	140	148	144	140	152
9	150	169	159	167	154	138	146	162	154	154
10	132	136	136	126	136	136	136	130	129	129
Best Result	3/10	1/10	1/10	4/10	1/10	2/10	0/10	1/10	3/10	1/10
Beat Tradnl	6/10	4/10	4/10	6/10	6/10	7/10	7/10	6/10	9/10	–

Table II: Results of 5% Fill 100x100 SCP. Pool Size = 600
Uniform Crossover, Random Infeasible Selection

Rows	Columns							
	1	2	3	4	5	6	7	8
1	0	1	1	0	1	0	0	1
2	1	1	0	0	1	1	0	0
3	0	0	1	1	0	1	1	0
4	1	1	0	1	0	0	1	1
5	0	0	1	1	0	1	0	0
6	0	1	0	1	0	0	1	0
7	1	0	1	0	1	0	0	0
Cost	3	6	2	4	5	1	7	5

Figure 1: Example SCP Represented as a Boolean Matrix

Test Case	Algorithm									
	(95,4)	(90,9)	(85,14)	(80,19)	(75,24)	(70,29)	(65,34)	OSC1	OSC2	Tradnl
1	88	88	88	92	90	90	87	88	90	88
2	117	124	113	122	114	108	108	108	107	124
3	109	106	105	109	105	106	106	104	104	112
4	118	120	121	129	118	121	117	114	116	120
5	114	109	104	110	113	112	112	114	104	106
6	124	121	121	124	117	119	123	123	124	126
7	108	104	105	104	104	115	110	120	104	118
8	83	87	83	82	85	84	84	82	84	85
9	89	85	84	84	85	92	84	92	84	96
10	147	140	130	137	138	138	130	137	130	145
Best Result	0/10	1/10	2/10	3/10	2/10	0/10	2/10	3/10	5/10	0/10
Beat Tradnl	8/10	8/10	9/10	7/10	8/10	7/10	9/10	8/10	9/10	---

Table III: Results of 8% Fill 100x100 SCP. Pool Size = 400
Uniform Crossover, Bias Infeasible Selection

Test Case	Algorithm									
	(95,4)	(90,9)	(85,14)	(80,19)	(75,24)	(70,29)	(65,34)	OSC1	OSC2	Tradnl
1	90	95	88	90	90	95	87	87	87	88
2	117	112	107	121	107	107	108	117	107	124
3	100	102	100	106	110	105	108	100	101	112
4	114	109	128	115	120	116	116	119	118	120
5	112	108	104	106	106	105	106	104	111	106
6	122	124	118	127	122	123	121	121	121	126
7	108	108	113	102	110	102	104	116	109	118
8	81	82	80	81	81	81	81	81	81	85
9	86	89	84	84	85	84	85	85	84	96
10	135	130	133	133	141	128	130	139	135	145
Best Result	1/10	1/10	6/10	2/10	1/10	4/10	1/10	3/10	3/10	0/10
Beat Tradnl	8/10	8/10	9/10	8/10	9/10	9/10	10/10	10/10	9/10	---

Table IV: Results of 8% Fill 100x100 SCP. Pool Size = 400
Uniform Crossover, Random Infeasible Selection

Test Case	Algorithm									
	(95,4)	(90,9)	(85,14)	(80,19)	(75,24)	(70,29)	(65,34)	OSC1	OSC2	Tradnl
1	50	49	53	49	49	50	49	50	49	55
2	57	60	57	57	57	57	58	61	63	57
3	54	54	55	56	54	54	54	53	54	55
4	71	70	74	76	72	72	72	69	70	72
5	74	78	78	78	76	74	77	69	74	78
6	76	77	76	77	76	71	71	72	71	72
7	58	56	59	54	56	56	62	59	55	59
8	44	42	42	42	44	42	42	42	44	48
9	56	55	60	54	58	59	58	57	55	56
10	76	77	77	76	71	75	76	76	77	79
Best Result	1/10	2/10	2/10	5/10	3/10	3/10	3/10	4/10	2/10	1/10
Beat Tradnl	9/10	8/10	7/10	7/10	8/10	9/10	7/10	8/10	9/10	--

Table V: Results of 12% Fill 100x100 SCP. Pool Size = 400
Uniform Crossover, Bias Infeasible Selection

Test Case	Algorithm									
	(95,4)	(90,9)	(85,14)	(80,19)	(75,24)	(70,29)	(65,34)	OSC1	OSC2	Tradnl
1	49	51	49	49	50	51	49	49	49	55
2	61	58	59	55	59	55	56	57	58	57
3	56	60	55	63	54	53	54	56	57	58
4	76	72	70	72	72	70	70	70	70	72
5	72	78	74	77	69	79	75	69	74	78
6	79	75	71	71	71	72	78	76	76	72
7	58	55	57	58	58	59	58	59	58	59
8	42	43	42	43	43	43	42	45	42	48
9	56	52	61	59	54	58	54	52	54	56
10	75	80	79	77	77	75	75	79	71	79
Best Result	2/10	2/10	4/10	3/10	2/10	3/10	3/10	4/10	4/10	0/10
Beat Tradnl	6/10	6/10	8/10	8/10	9/10	8/10	9/10	8/10	8/10	--

Table VI: Results of 12% Fill 100x100 SCP. Pool Size = 400
Uniform Crossover, Random Infeasible Selection

Test Case	Algorithm									
	(95,4)	(90,9)	(85,14)	(80,19)	(75,24)	(70,29)	(65,34)	OSC1	OSC2	Tradnl
1	28	28	29	28	28	28	29	29	30	28
2	42	46	46	46	44	41	42	44	46	45
3	36	36	36	38	34	36	36	34	36	39
4	52	50	50	50	50	50	51	50	53	50
5	58	65	58	61	60	57	60	57	58	63
6	31	30	30	30	30	31	30	30	30	30
7	35	38	36	34	35	35	35	35	35	40
8	33	34	34	34	33	33	33	34	32	35
9	41	37	41	41	41	37	37	37	37	41
10	50	52	50	51	51	48	48	50	49	51
Best Result	1/10	4/10	2/10	4/10	4/10	6/10	3/10	5/10	3/10	3/10
Beat Tradnl	8/10	7/10	8/10	9/10	10/10	9/10	8/10	9/10	7/10	--

Table VII: Results of 16% Fill 100x100 SCP. Pool Size = 400
Uniform Crossover, Bias Infeasible Selection

Test Case	Algorithm									
	(95,4)	(90,9)	(85,14)	(80,19)	(75,24)	(70,29)	(65,34)	OSC1	OSC2	Tradnl
1	29	29	28	28	28	28	28	28	29	28
2	45	44	42	44	46	41	44	43	44	45
3	38	36	34	36	36	34	38	36	36	39
4	52	50	50	50	50	50	50	50	50	50
5	59	58	57	57	57	60	58	60	57	63
6	30	30	30	30	30	30	30	30	31	30
7	37	35	35	35	35	35	36	36	35	40
8	32	33	32	32	33	37	32	33	34	35
9	41	37	37	40	39	37	37	37	37	41
10	50	50	51	48	53	48	48	50	48	51
Best Result	2/10	4/10	8/10	7/10	5/10	8/10	6/10	4/10	5/10	3/10
Beat Tradnl	8/10	9/10	10/10	10/10	8/10	9/10	10/10	10/10	8/10	--

Table VIII: Results of 16% Fill 100x100 SCP. Pool Size = 400
Uniform Crossover, Random Infeasible Selection

	Percent Fill 100x100 SCP			
	5	8	12	16
% Bias is Better	32.2	27.8	38.9	23.3
% Random is Better	57.8	57.8	42.2	40.0
% Same	10.0	14.4	18.9	36.7
% Initial Infeasible	98.7	77.0	12.4	1.0

Table IX: Bias versus Random Infeasible Selection