Proceedings of the 7th Oklahoma Symposium on Artificial Intelligence, November 18-19, 1993, Stillwater, OK, pp. 215-225.

Manipulating Subpopulations in Genetic Algorithms for Solving the k-way Graph Partitioning Problem*

D. Ansa Sekharan
Roger L. Wainwright

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

ABSTRACT

This paper explores the partitioning of the population pool in genetic algorithms (GA) into separate subpopulations of feasible and infeasible solutions, and the interaction on a regular basis of crossover operations among and within the subpopulations. The Graph Bisection Problem and the k-way Graph Partitioning Problems were chosen as representative optimization problems to apply our subpopulation strategies. We designed several 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. We also developed several new crossover operators to be used while manipulating feasible and infeasible solutions. All of our algorithms significantly and consistently outperformed the traditional GA in all of the test problems. This illustrates the importance of infeasible solutions as a source of good genetic material. Furthermore, results show that two of our new crossover operators consistently out performed the others.

1. INTRODUCTION

The Graph Bisectioning problem (GBP) partitions the vertices of an undirected weighted graph $G = (V,E)$, ($|V|$ is even) into two sets of equal size such that the weight of the edges between them is minimized. Such a partition is sometimes called a *min-cut* or a *minimum bisection*. Graph partitioning has important applications to VLSI such as floor planning, and module placement and routing [2,16]. Other applications include memory segmentation to minimize paging [14], and processor allocation.

Since the graph bisectioning problem has been shown to be NP-complete [11], there have been several heuristic algorithms developed to yield good approximations for the GBP problem. The Kernighan-Lin algorithm [14] is recognized as the classic approach to this problem. Recently, Saab and Rao [19] developed a series of algorithms for the GBP problem. Their adaptive heuristic technique outperformed the Kernighan-Lin algorithm in 70% of the cases they tested, while executing three times faster. Both the Kernighan-Lin and Saab-Rao techniques are greedy algorithms. That is, they begin with an initial solution, then a neighborhood of the current partition is searched for a new partition. If the cost is reduced, the current solution is replaced by the new solution. This process is repeated until no more improvement can be found. One danger of greedy algorithms is the possibility of falling into a local minimum. Usually it is difficult to get out of the local minimum, and furthermore, it may be far from the optimal solution.

Zhao *et al.* [24] developed a simulated annealing (SA) technique for the graph bisectioning problem. They theo-

* Research supported by OCAST Grants AR2-004 and ARO-038 and Sun Microsystems Inc.

rized that since the SA technique generally avoids local minimums, perhaps the SA approach will yield superior solutions to the Kernighan-Lin and Saab-Rao algorithms. Their results suggest this. They found their SA graph bisectioning algorithm obtained better results in nearly all of the cases they tested. Their test cases ranged from 50 to 500 nodes. Furthermore, the CPU times for their SA algorithms were less than the other two algorithms for 100 nodes or more.

The k-way graph partitioning problem (k-GPP) is also a combinatorial problem which has applications in the VLSI design of electrical circuits, mapping and many other areas of computer science. The k-way graph partitioning problem partitions the vertices of an undirected weighted graph $G = (V,E)$, into k disjoint subsets of vertices $V_1,\ldots,V_k$, such that the sum of the weights of the edges between the subsets is minimal, and the sizes of the subsets are as equal as possible. The subsets are called *partitions*, and the set of edges that must be removed to form the partitions is called a *cut*.

2. GENETIC ALGORITHMS

The genetic algorithm (GA), first described by Holland [12] in the early 1970's, is a robust search and optimization technique based on the principles of natural genetics and survival of the fittest. Genetic algorithms use the laws of natural selection and genetics to guide a non-deterministic search. In genetics, a set of chromosomes contain all of the genes (information) which form the "blueprint" for a species. In a genetic algorithm, a chromosome is a string which encodes a possible solution for a problem. The traditional genetic algorithm begins by creating an initial population of feasible solutions, and then recombines feasible solutions in such a way to guide the search to only the most promising areas of the state space.

Genetic algorithms search through the solution space by emulating biological selection and reproduction. Each new generation is created by a biased reproduction. That is, the more "fit" members of the population have a better chance of reproduction. 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 (chromosome). Each chromosome is given a measure of "fitness" by the fitness function, which is sometimes called the objective or evaluation function. The fitness function drives the population toward better solutions and is the most important part of the algorithm. The fitness function is what distinguishes one problem from another [15]. The fitness of a chromosome determines its ability to survive and reproduce offspring in the next generation. Since a fixed size population is maintained, the "least fit" or weakest chromosomes of the population are displaced by more fit chromosomes. Hence the population of chromosomes tends to mature to the optimal or near-optimal solutions with each new generation.

Genetic Algorithms are applicable to a wide variety of problems and have been very successful in obtaining near-optimal solutions to many different combinatorial optimization problems [1,3,4,6,7,8,17,23]. Genetic algorithm packages for a single processor have been available for several years. A steady-state GA such as GENITOR [22] and a generational GA such as GENESIS [10], and LibGA [5] which offers the ability to use both a generational or a steady-state approach are three example packages that are readily available. Davis, Goldberg and Rawling, provide an excellent in depth study of genetic algorithms [6,7,9,18].

3. GA APPLIED TO THE GBP PROBLEM

Consider the GBP problem with n nodes, where n is even. A Chromosome defining a possible partitioning is depicted as a bit string of length n. The ith bit of the chromosome corresponds to the ith node in the graph. A one bit means the associated node is included in one partition and a zero bit means the node is included in the other partition. There are obviously 2^n possible chromosomes. However, a feasible solution is defined as any chromosome where the number of one bits equals the number of zero bits, otherwise the partitioning is unequal in the number of nodes, hence illegal. The optimal solution is a feasible solution with the minimum cut.

In general, the traditional genetic algorithm implementation uses a single population pool. The population can contain both feasible and infeasible solutions. In some GA's only feasible solutions are considered; infeasible solutions are discarded and not placed in the pool. In some GAs infeasible solutions are given an extremely poor or non-existent fitness value, hence they rarely, if ever, participate in the genetic recombination operations. The traditional single pool GA could be made to manipulate infeasible solutions, however, 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 our previous paper involving the Set Covering Problem [20], 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 determined that infeasible solutions often contain an excellent source of genetic material that should not be overlooked. 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. Finally, 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 [20]. We developed nine different algorithms involving feasible and infeasible solutions and compared these algorithms against the traditional GA. Among the nine algorithms we tested, no one algorithm was significantly better than any of the others. However, all nine algorithms outperformed the traditional GA in all of our test cases solving the set covering problem. We concluded 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. In this paper we apply this strategy to the GBP and the k-GPP problems.

In our GA model, we represent a graph bisection of n nodes as a chromosome of n bits. Each bit is associated with a node. The nodes represented by zero bits form one partition, and the nodes represented by one bits form the other. Obviously, a feasible solution is represented by a chromosome with an equal number of one and zero bits. In our GA model, feasible and infeasible solutions are maintained in separate subpopulation pools of equal size of $ps/2$. The total population of size ps at each generation remains constant. We used GENITOR [22] which was modified for our particular application. GENITOR is a steady-state GA, generating one child at a time while removing the worst chromosome in the population.

The initial population is constructed by randomly generating each bit in each chromosome. Each resulting chromosome representing either a feasible or infeasible solution is then placed into the 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 $ps/2$ as future children are generated. The evaluation function for a chromosome in the feasible solution pool is the cost of the cut. We define *off* as | no. 1's - no. 0's | / 2. That is, if n = 100 and a chromosome has 46 1 bits then *off* = 4. The evaluation function for infeasible solutions is *off* * *const* + *cut*, where *const* is some constant larger than the sum of the weights of all of the edges in the graph, and *cut* is the cost of the partitioning. In this way the infeasible solutions are ranked first by *off* then secondly by the *cut* of the partition. For feasible solutions, *off* is zero. Hence feasible solutions are ranked by the cost of the *cut*, as expected. In all cases the mutation operation consists of exchanging two randomly selected bits 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 [20].

We have designed a class of algorithms for manipulating the feasible and infeasible population pools. An 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 $(95, 4, 1)$, for example, means that for any given crossover operator, there is a 95% chance that two chromosomes from the feasible pool will be selected, 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, oscillation algorithm (*OSC*) holds z constant at 1% and oscillates x from 75 to 98, then back down to 75, and so forth in increments and decrements of 1. The y percent is such that $x+y$ is always 99. The x, y, z values for

(*OSC*) were selected after testing numerous alternatives.

We also compared the difference in executing a problem using a bias selection from the infeasible pool, and a random selection. In our previous work [20] we noted there was no distinguishable difference whether chromosomes were selected for the infeasible solution randomly or using a selection bias. In this research, we tested the effect of random and bias selections from the infeasible pool on the GBP problem. In our notation, algorithm (*95, 4, 1*) - *Bias* means the chromosomes were selected for the infeasible solution using a selection bias, and algorithm (*95, 4, 1*) - *Random* means the chromosomes were selected for the infeasible solution randomly. A similar notation is used for the (*OSC*) algorithm.

Tables I through III show the results of testing various algorithms on a GBP problem of $n = 50$ nodes. We tested the following algorithms: (*95, 4, 1*) - *Random*, (*OSC*) - *Random*, (*95, 4, 1*) - *Bias* and (*OSC*) - *Bias*. We compared these results with the traditional single pool GA *(Tradl)*. The fitness function described in the previous section was used in all of the algorithms. We also tested the *Saab-Rao* algorithm. The *Saab-Rao* results are given in the title of each Table. All edge weights were randomly selected between one and four. The only difference between the GBP problems used in Tables I through III is the degree probability. Table I gives results of a 50 node GBP with degree probability of 0.2. Table II and Table III represent problems of size 50 with degree probabilities 0.3 and 0.4 respectively. The degree probability is the probability that any two nodes in the graph are connected. For a given degree probability, the specific edge connectivity was determined randomly.

In each GBP problem we used the uniform crossover operator. Results of the uniform crossover are shown at the top of each table. The population size when the uniform crossover was used was 400. We also tested a multi-point crossover operator, for various values of m ranging from the standard single and double point crossover operators to 5, 10, 15, ...40 point crossover operators. We also tested two random point crossover operators, one randomly ranging from 1..50 and the other randomly ranging from 15..35. The population size for each of the m-point crossover examples was 600.

4. RESULTS FOR THE GBP PROBLEM

Results from Tables I through Table III are consistent. Clearly the Saab-Rao algorithm is an excellent heuristic algorithm for the GBP problem yielding excellent results. Our four genetic algorithms as a group performed comparatively well to the Saab-Rao algorithm yielding the same results in many instances, and a superior result in one case. Results indicate there was no one crossover technique that consistently performed better than any of the others. Nor was there any significant indication that random selection was any better or worse than bias selection. One overwhelming result, however, was that our four GA algorithms, which took the infeasible solutions into account, consistently performed better than the traditional genetic algorithm. In only two instances out of 39 cases in Tables I through III did the traditional GA match the best performance of the other four genetic algorithms. Furthermore, there was no instance in which the traditional GA was superior to any of the other four genetic algorithms, which made use of infeasible solutions.

5. GA APPLIED TO THE k-GPP PROBLEM

The main thrust of this paper is the k-GPP problem. The k-GPP problem has a much larger state space than the GBP problem for the same number of nodes. Furthermore, there is no special heuristic for the k-GPP problem like the Saab-Rao algorithm is for the GBP problem. Jones and Beltramo [13] developed a genetic algorithm for solving partitioning problems. Their work centered on techniques for partitioning a sequence of numbers into k piles of numbers of equal sum. This is sometimes called the "equal piles problem". This problem is similar to the k-GPP problem and may be of interest to the reader.

We adopted the chromosome encoding scheme for the k-GPP problem suggested by Von Laszewski [21]. A k-GPP problem with n nodes (n is a multiple of k) is represented by a chromosome of length n, where each gene represents a specific node. Each gene of the chromosome may take on any integer value from 1..k. For example, $n = 12$ and $k = 3$, a possible chromosome representation could be (1 3 3 2 1 2 3 2 1 3 1 2) or perhaps (1 3 3 2 3 2 3 2 2 3 1 2). Notice the first chromosome has an equal number of one's, two's and three's, representing an equal partitioning of the 12 nodes into three groups. This represents a feasible solution. The second chromosome does not depict an equal partitioning of the 12 nodes into three groups, thus representing an infeasible solution. Von Laszewski's genetic algorithm for the k-way graph partitioning problem does not take into account infeasible solutions.

In the k-GPP problem, the evaluation function for a chromosome in the feasible solution pool is the cost of the cut, just like the GBP problem. We define *off* as the minimum number of genes that must be altered to convert an infeasible solution into a feasible solution. For example, the value for *off* for the second chromosome in the above example is two. The evaluation function for infeasible solutions is *off* * *const* + *cut*, where *const* is some constant larger than the sum of the weights of all of the edges in the graph, and *cut* is the cost of the partitioning. In this way the infeasible solutions are ranked first by *off* then secondly by the *cut* of their partition. For feasible solutions, *off* is zero. Hence feasible solutions are ranked by the cost of the *cut*, as expected. In all cases the mutation operation consists of exchanging two randomly selected integers in a string. We used several different crossover operators for our GA implementation of the k-GPP problem. These include *Structural*, *Modified Uniform*, *Modified Asexual*, and *Asexual Uniform*. All of the crossover operators are described below.

Von Laszewski [21] suggests an intelligent structural crossover operator for the k-way graph partitioning problem. This operator copies whole partitions into an offspring to avoid destroying valuable information that has already been gained. We call this the *Structural* crossover operator. Consider the following example for n = 16, and k = 4:

Parent 1: (1 3 3 3 1 3 4 4 **2 2** 4 4 **2 2** 1 1)

Parent 2: (1 1 3 3 1 2 3 3 **1 2** 4 4 **4 2** 2 4)

One of the partitions is randomly selected from Parent 1, for example partition 2. Partition 2 from Parent 1 is overlaid on top of Parent 2. The values involved are shown in bold. This produces

Child: (1 1 3 3 1 2 3 3 **2 2** 4 4 **2 2** 2 4),

which is infeasible. This is corrected by noting **1** and **4** values were destroyed in Parent 2, and now the Child has two extra **2** values. Hence two **2** values (other than the ones copied from Parent 1) are randomly selected and altered to a **1** and a **4** producing the final Child:

Child: (1 1 3 3 1 **1** 3 3 2 2 4 4 2 2 **4** 4),

which is feasible. The *Modified Uniform* crossover is an extension of the uniform crossover operator. The first child is produced as a result of the uniform crossover, and placed into the feasible or infeasible pool whichever is appropriate. A second child is also produced using the uniform crossover operator. However, if this is infeasible it is fixed so that it becomes feasible and then placed into the feasible pool. This guarantees at least one feasible solution from the crossover operator, which otherwise might be very rare. Note our experiments using the uniform crossover operator (unmodified) produced extremely poor results, and was not considered.

The traditional asexual crossover operator randomly selects two locations within a chromosome and exchanges values producing one child. Our *Modified Asexual* is carried out for the X, Y and Z crossovers and produces one or two children. The first child is generated using the traditional asexual operator. This child is placed into the proper pool. If, however the first child was an infeasible solution, then a second child is generated by "fixing" the first child to make it feasible. This ensures propagation of infeasible solutions and the generation of new feasible solutions.

We define the *Asexual Uniform* crossover operator as follows. For an X-Crossover the traditional asexual crossover operator is used. However, for Y-Crossover and Z-Crossover operations, the traditional uniform crossover operator is used. In all cases, the child is placed into the proper pool. If, however the child was an infeasible solution, then a second child is generated by "fixing" the first child to make it feasible. In this regard, it is similar to the *Modified Asexual* crossover operator.

5. RESULTS OF THE k-GPP PROBLEM

We generated several datasets to test our GA implementation of the k-GPP problem. Datasets were randomly generated for n = 60 and 120, for k = 3..5, and for n = 240, for k = 3 and 4. Each edge was given unit weight, and the degree probability for each node was randomly generated in the range 2..n/2. We tested the following algorithms: Traditional GA, *(95,4,1) - Random* and *OSC - Random*. We ran each of these three algorithms using each of our four crossover techniques, *Structural*, *Modified Uniform*, *Modified Asexual*, and *Asexual Uniform* for each of the datasets.

Figure 1 through Figure 3 shows the results of the 3-way graph partitioning genetic algorithms for n = 60, 120 and 240, respectively. The *Asexual Uniform* crossover is defined only for infeasible solutions, thus the traditional GA was not run using this crossover. Notice the overall poor performance in all cases when the *Structural* crossover operator was used. In general the *(95,4,1) - Random* and *OSC - Random* algorithms, which make use of the infeasible pool, performed better than the traditional GA algorithm. Overall, the best algorithm was *OSC - Random* using either the *Modified Asexual*, or *Asexual Uniform* crossover operator.

Figure 4 through Figure 6 shows the results of the 4-way graph partitioning genetic algorithms for n = 60, 120 and 240, respectively. Notice again the overall poor performance in all cases when the *Structural* crossover operator was used. In general the traditional GA performed better when the *Structural* and *Modified Uniform* crossover operators were used. However, the overall best algorithms appeared to be *(95,4,1) Random* and *OSC - Random* using either the *Modified Asexual*, or *Asexual Uniform* crossover operator.

Figure 7 and Figure 8 show the results of the 5-way graph partitioning genetic algorithms for n = 60 and 120, respectively. Notice again the overall poor performance in these two cases when the *Structural* crossover operator was used. In general the traditional GA performed better when the *Structural* and *Modified Uniform* crossover operators were used. This was not the best results, however. Again, the overall best algorithms appeared to be *(95,4,1) Random* and *OSC - Random* using either the *Modified Asexual*, or *Asexual Uniform* crossover operator.

6. CONCLUSIONS

The Saab-Rao algorithm is an excellent heuristic algorithm for the GBP problem. The four genetic algorithms we developed that used both feasible and infeasible pools compared reasonably well to the Saab-Rao algorithm. The performance of the traditional GA, however, for the GBP was the worse of all the algorithms we tested. The overall best algorithms for the k-way graph partitioning problem are the *(95,4,1) Random* and *OSC - Random* that we developed using either of our *Modified Asexual*, or *Asexual Uniform* crossover operators. The *Structural* and *Modified Uniform* crossover operators in general are not effective for this problem. The performance of the traditional GA was hindered by not taking advantage of an infeasible pool. This research strongly suggests if a problem generates infeasible solutions, then it may be extremely important to the success of the algorithm to include infeasible solutions in the crossover operations on some regular basis.

ACKNOWLEDGEMENTS

This research has been partially supported by OCAST Grants AR2-004 and ARO-038. The authors wish to acknowledge the support of Sun Microsystems Inc. The authors also wish to thank K. Thulasiraman [24] for supplying the authors with code for the Saab-Rao algorithms.

REFERENCES

[1] J.L. Blanton and R.L. Wainwright, "Multiple Vehicle Routing with Time and Capacity Constraints using Genetic Algorithms", in S. Forrest, ed., *Genetic Algorithms: Proceedings of the fifth International Conference (GA93)*, Morgan Kaufmann, San Mateo, CA.

[2] S. Bhatt and F. Leighton, "A Framework for Solving VLSI graph Problems", *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 300-343, Apr., 1984.

[3] 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.

[4] A.L. Corcoran and R.L. Wainwright, "A Genetic Algorithm for Packing in Three Dimensions", *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, 1992, pp. 1021-1030, ACM Press.

[5] A.L. Corcoran and R.L. Wainwright, "LibGA: A User-friendly Workbench for Order-based Genetic Algorithm Research", *Proceedings of the 1993 ACM/SIGAPP Sympo-*

sium on Applied Computing, pp. 111-118, 1993, ACM Press.

[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] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[12] J.H. Holland "Adaptation in Natural and Artificial Systems", Ann Arbor: The University of Michigan Press, 1975.

[13] D.R. Jones and M.A. Beltramo, "Solving Partitioning Problems with Genetic Algorithms", *Proceedings of the Fourth International Conference on Genetic Algorithms,* pp. 442-449, Morgan Kaufmann, 1989.

[14] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning graphs", *The Bell Technical Journal*, vol. 49, pp. 291-307, Feb., 1970.

[15] L.R. Knight and R.L. Wainwright, "HYPERGEN: A Distributed Genetic Algorithm on a Hypercube", *Proceedings of the 1992 Scalable High Performance Computing Conference, SHPCC'92,* Williamsburg, Va., April 26-29, 1992.

[16] D. La Potin and S. Directer, "Mason: A Global Floorplanning Approach for VLSI Design", *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 477-489, Oct., 1986.

[17] P.P. Mutalik, L.R. Knight, J.L. Blanton and R.L. Wainwright, "Solving Combinatorial Optimization Problems Using Parallel Simulated Annealing and Parallel Genetic Algorithms", *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, pp. 1031-1038, 1992, ACM Press.

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

[19] Y.G. Saab and V.B. Rao, "Fast Effective Heuristics for the Graph Bisectioning Problem", *IEEE Transaction on Computer-Aided Design*, vol. 9, no. 1, pp. 91-98, January, 1990.

[20] D.A. Sekharan and R.L. Wainwright, "Manipulating Subpopulations of Feasible and Infeasible Solutions in Genetic Algorithms", *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pp. 118-125, 1993, ACM Press.

[21] G. von Laszewski, "Intelligent Structural Operators for the k-way Graph Partitioning Problem", *Proceedings of the Fourth International Conference on Genetic Algorithms,* June, 1991, pp. 45-52.

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

[23] D. Whitley, 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.

[24] Y.C. Zhao, L. Tao, K. Thulasiraman and M.N.S. Swamy, "An Efficient Simulated Annealing for Graph Bisectioning", *Proceedings of the 1991 IEEE/ACM/SIGAPP Symposium on Applied Computing*, pp. 65-68, 1993, IEEE Press.

	Algorithm				
	(95,4,1) Random	OSC Random	(95,4,1) Bias	OSC Bias	Tradl
Uniform	194	172	169	169	200
M-point					
1	214	198	224	218	229
2	240	236	227	210	229
5	204	191	199	181	185
10	173	169	200	191	176
15	169	185	176	193	183
20	172	175	196	183	175
25	177	169	192	169	181
30	172	175	169	172	179
35	169	179	176	175	183
40	169	173	205	172	176
RAND [1..50]	180	177	195	194	196
RAND [15..35]	180	186	182	172	178

Table I: GBP Results for 50 Nodes, Deg. Prob. = 0.2, Uniform Pool Size = 600, M-point Pool Size = 400 Saab_Rao Cost = 169

	Algorithm				
	(95,4,1) Random	OSC Random	(95,4,1) Bias	OSC Bias	Tradl
Uniform	297	292	298	297	302
M-point					
1	342	351	341	335	335
2	345	343	341	333	347
5	326	327	292	303	297
10	301	317	306	305	316
15	300	311	299	305	302
20	299	299	299	298	308
25	297	296	307	304	309
30	296	299	297	299	308
35	300	309	299	308	306
40	295	297	299	297	299
RAND [1..50]	298	299	310	322	305
RAND [15..35]	298	296	299	295	301

Table II: GBP Results for 50 Nodes, Deg. Prob. = 0.3, Uniform Pool Size = 600, M-point Pool Size = 400 Saab_Rao Cost = 293

	Algorithm				
	(95,4,1) Random	OSC Random	(95,4,1) Bias	OSC Bias	Tradl
Uniform	426	432	426	426	426
M-point					
1	440	447	472	469	490
2	493	479	521	459	483
5	429	429	443	426	447
10	435	442	459	448	450
15	426	434	432	439	435
20	426	426	431	437	431
25	432	435	426	437	431
30	430	428	435	426	444
35	428	433	432	426	430
40	426	426	428	436	426
RAND [1..50]	438	426	431	426	437
RAND [15..35]	426	426	435	426	437

Table III: GBP Results for 50 Nodes, Deg. Prob. = 0.4, Uniform Pool Size = 600, M-point Pool Size = 400 Saab_Rao Cost = 426

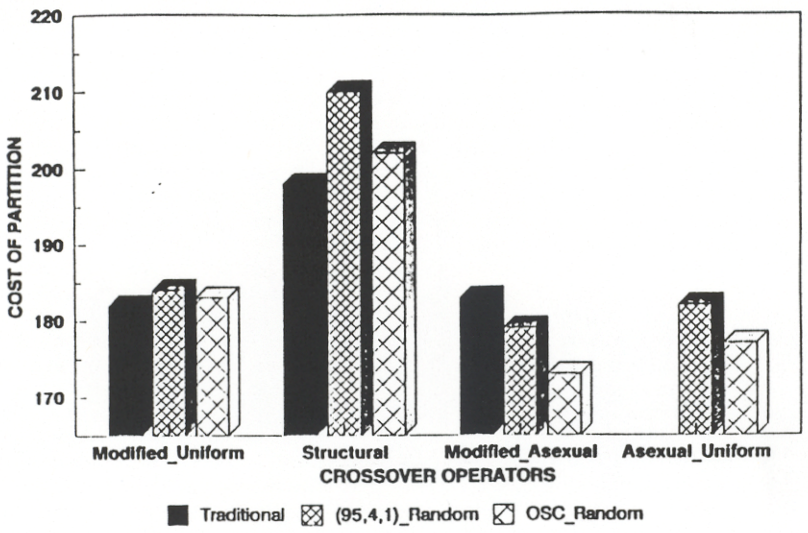

FIGURE 1: 3-WAY GRAPH PARTITION, 60 NODES, POOL SIZE = 100

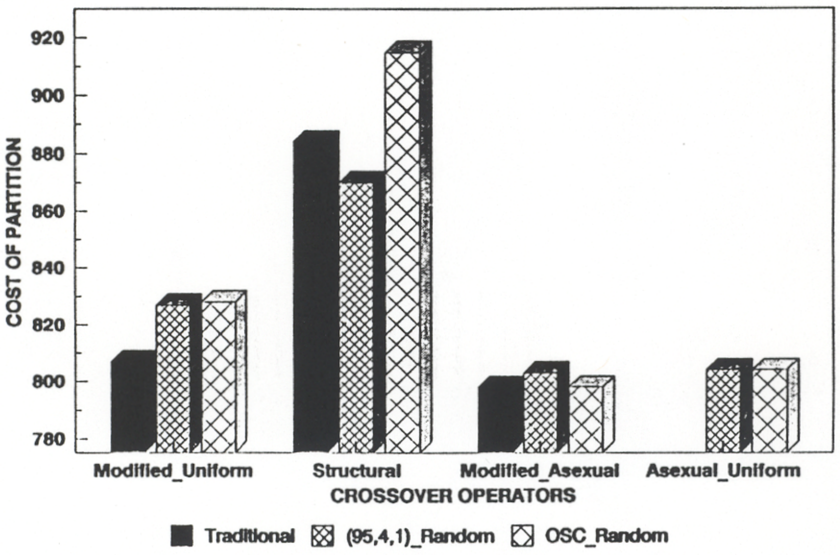

FIGURE 2: 3-WAY GRAPH PARTITION, 120 NODES, POOL SIZE = 200

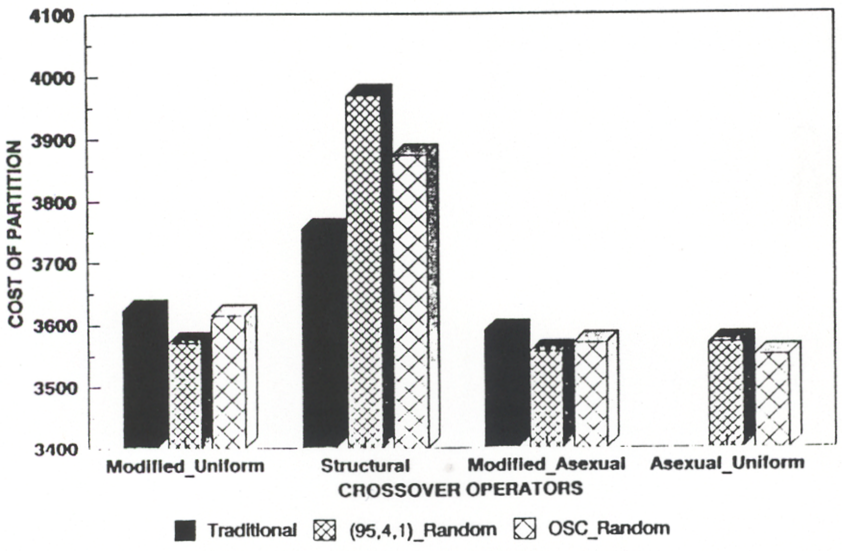

FIGURE 3 : 3-WAY GRAPH PARTITION, 240 NODES, POOL SIZE = 300

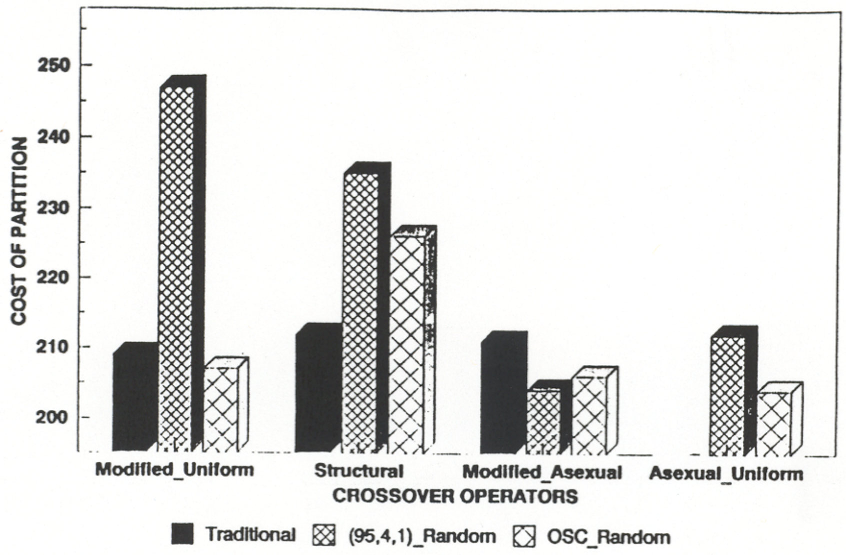

FIGURE 4 : 4-WAY PARTITION, 60 NODES, POOL SIZE = 200

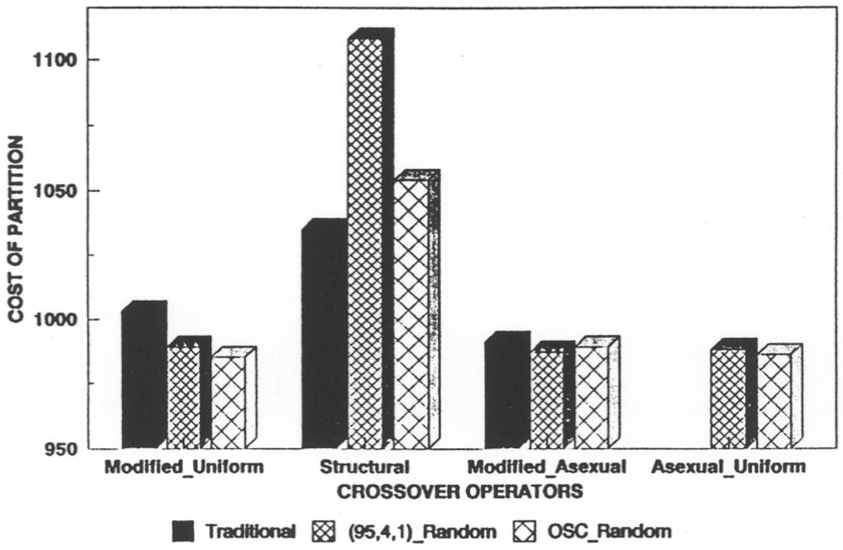

FIGURE 5 : 4-WAY PARTITION, 120 NODES, POOL SIZE = 300

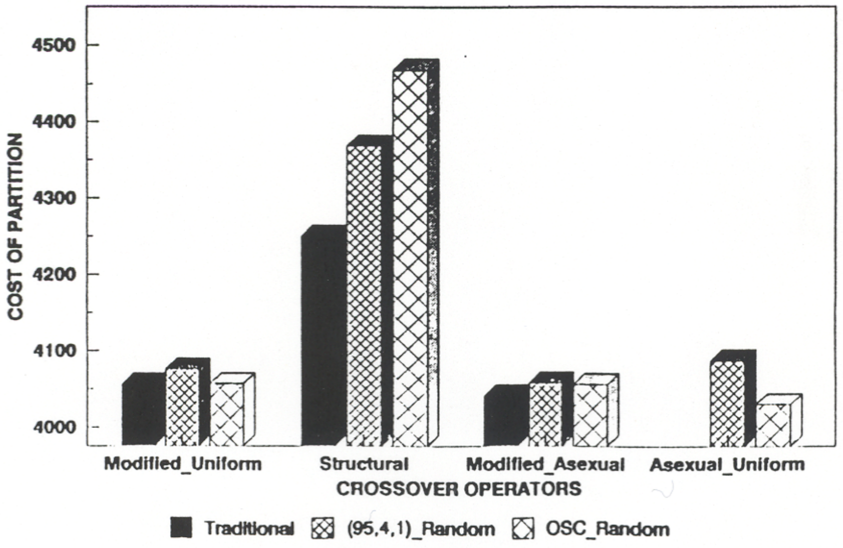

FIGURE 6 : 4-WAY PARTITION, 240 NODES, POOL SIZE = 400

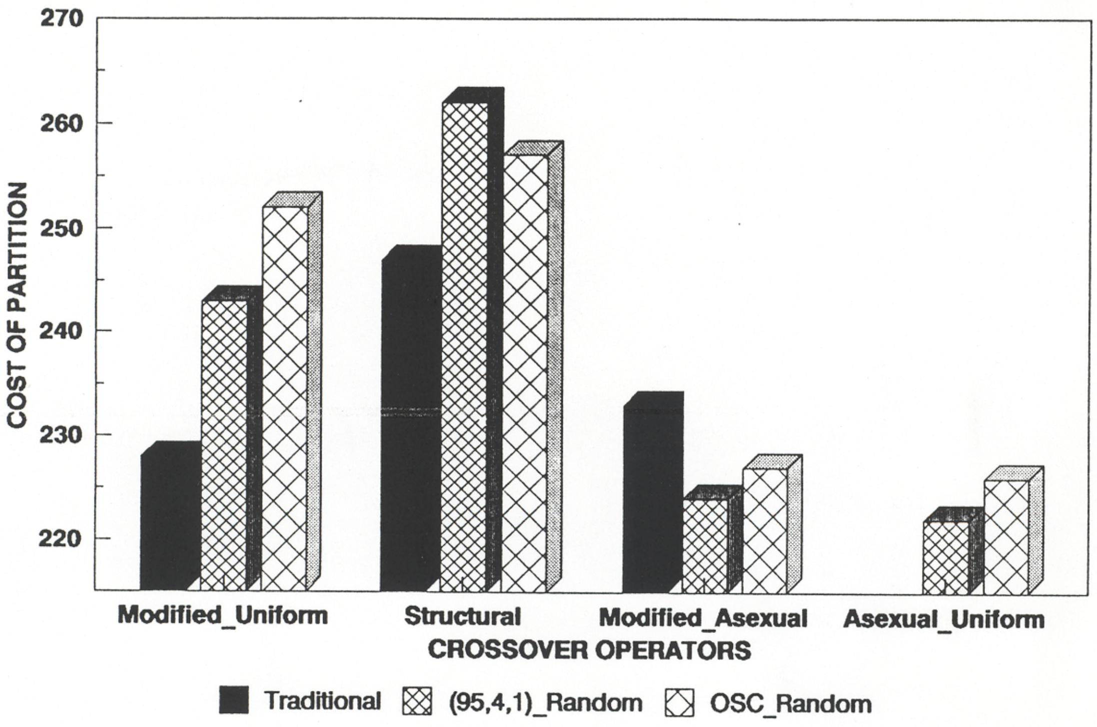

FIGURE 7 : 5-WAY PARTITION, 60 NODES, POOL SIZE = 50

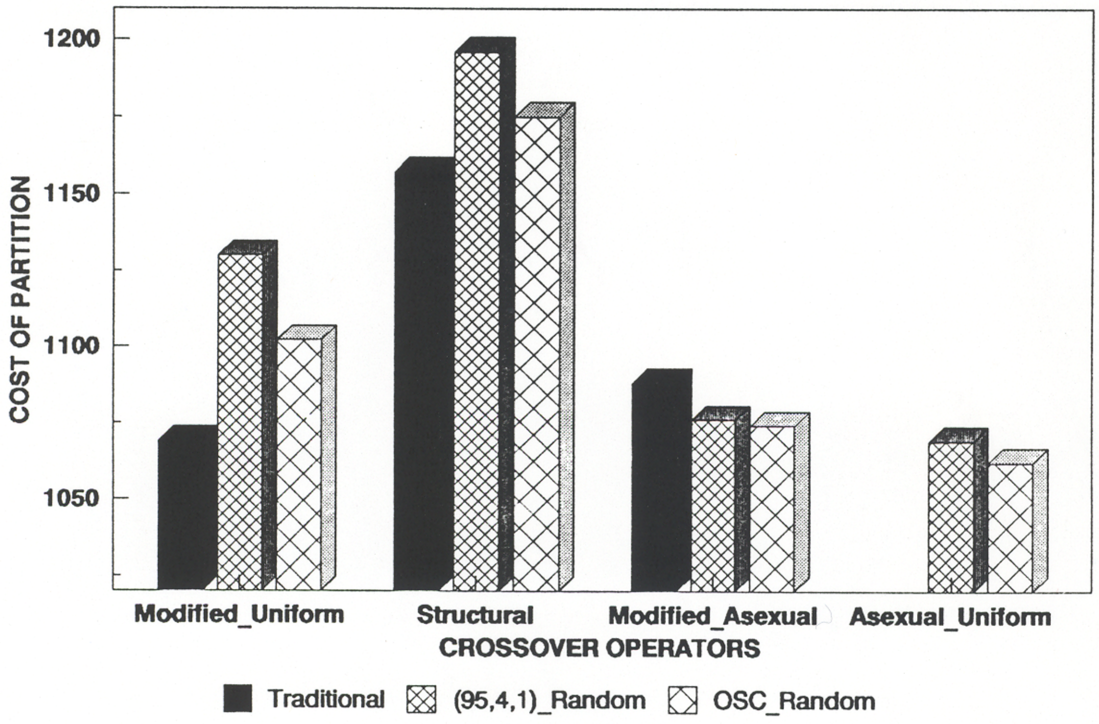

FIGURE 8 : 5-WAY PARTITION, 120 NODES, POOL SIZE = 75