A Parallel Stochastic Optimization Algorithm for Finding Mappings of the Rectilinear Minimal Crossing Problem

John T. Thorpe *
jtthorpe@cs.clemson.edu

Frederick C. Harris, Jr.
fredh@cs.clemson.edu

Department of Computer Science
Clemson University
Clemson, South Carolina 29634-1906

Abstract: Parallel processing has been a valuable tool for improving the performance of many algorithms. Solving intractable problems is an attractive application of parallel processing. Traditionally, exhaustive search techniques have been used to find solutions to NP-complete problems. However, the performance benefit of parallelization of exhaustive search algorithms can only provide linear speedup, which is typically of little use as problem complexity increases exponentially with problem size. Genetic algorithms can be useful tools to provide satisfactory results to such problems. This paper presents a genetic algorithm that uses parallel processing in a cooperative fashion to determine mappings for the rectilinear crossing problem. Results from this genetic algorithm are presented which contradict a conjecture that has been open for over 20 years regarding the minimal crossing number for rectilinear graphs.

keywords: Rectilinear Minimal Crossing, Parallel Genetic Algorithms

1 Introduction

Parallel processing has proven to be a valuable tool for increasing the performance of various algorithms. Intractable problems solved by exhaustive search techniques seem to resist the speedup typically brought about by parallelization. Algorithms for these problems, when run in parallel, typically only give a speedup directly proportional to the number of processors

*Current Address: AT&T Global Information Solutions, GIS WP&S, CDT, Liberty, SC 29657, jthorpe@admin.clemsonSC.NCR.COM

working in concert on the problem. This paper presents an alternative to traditional exhaustive search methods by using a variation on genetic algorithms [7, 4].

In the early 1970's, John Holland at the University of Michigan developed a heuristic search technique known as a genetic algorithm [7]. Because they are heuristic techniques, genetic algorithms are not guaranteed to find optimal solutions but rather to find "acceptable" solutions when searching complex systems. An acceptable result is defined by the problem at hand, and by how close to the optimal solution the user deems satisfactory. It is hoped that using genetic algorithms in parallel can produce results in a "reasonable" amount of time as good as or better than exhaustive search techniques.

The problem chosen to demonstrate the viability of using genetic algorithms in parallel is the rectilinear crossing problem. The rectilinear crossing problem is an extension of the minimal crossing problem as described by Garey and Johnson [6] as the GRAPH GENUS problem. Johnson [11] places the crossing problem in its own category as an NP-complete problem. In general, crossing problems attempt to find some optimal mapping of a graph such that the number of edge crossings (intersections) is minimized. First of all, it is necessary to define the criteria for a valid graph in order to determine what the crossing number of a valid graph is. Chartrand and Lesniak [3] state the definition this way: the *crossing number* of a graph G, denoted $\nu(G)$, is the minimum number of crossings (of its edges) among the drawings of G in the plane. Assumptions about the drawings are as follows:

(a) adjacent edges never cross – i.e. edges with a common vertex;
(b) two non-adjacent edges cross at most once;
(c) no edge crosses itself;
(d) no more than two edges cross at a point of the plane; and
(e) the (open) arc in the plane corresponding to an edge of the graph contains no vertex of the graph

The number of crossings produced by such a mapping gives a measure of how non-planar the graph is. Extending the constraints of the minimal crossing problem by requiring that edges of the graph must be straight lines creates the rectilinear crossing problem. So the question to be solved is: What is the crossing number of a complete graph G given that G is rectilinear and lies in a bounded planar region, and what mapping will provide a minimal crossing number for that graph?

For complete graphs that have ten or fewer vertices, the crossing numbers are known [8]. For larger graphs, no minimal crossing number has been verified. Formulas that postulate what the crossing numbers should

be and methods to produce mappings with that many crossings have been developed but not verified for their minimality [8, 1]. Verification has been difficult mainly due to the sheer number of edges and the difficulty of counting the crossings by hand. Additionally, isomorphic transformations of the solutions can compound the difficulty of identifying unique solutions.

Once the crossing numbers for complete graphs have been determined, the problem can easily be generalized to apply to less than complete graphs. Examples of such graphs are circuit diagrams, communication networks, railroad track and highway networks, and other interconnection diagrams. Recent papers by Chang [2] and Wang, Lee, and Chang [12] have described circuit design in similar terms to the minimal crossing problem. Chang [2] proposed an algorithm for minimizing *vias* in multi-layer circuits, which is a transformation of the minimal crossing problem.

The purpose of this paper is to demonstrate the effectiveness of using genetic algorithms in concert with parallel processing in obtaining good solutions for large or intractable problems. In Section 2, previous work on the minimal crossing problem, NP-completeness, and genetic algorithms are presented. Section 3 explains the heuristic used to solve the problem, as well as describing the platform that the heuristic was implemented on. Results and analysis of the algorithm are presented in Section 4. Section 5 provides some conclusions and future work is presented in Section 6.

2 Previous Work

Minimal Crossing Problem

Twenty years ago, almost all questions about the minimal crossing problem were unanswered. Methods for generating mappings to provide graphs with conjectured minimal crossing numbers have been developed, and formulas that provide the expected minimal crossing number have been proposed [8, 1]. The majority of these formulas and methods have been verified by visual inspection for a small number of vertices. As the number of edges and vertices in the graphs grow in number, this method of verification becomes progressively more difficult. Even exhaustive searches by computer programs to find and verify mappings become prohibitively time consuming with increased problem size.

For graphs of size 10 and less, a simple formula provides the minimal crossing number for complete graphs [8, 3]:

$$\nu(K_n) = \frac{\lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \lfloor \frac{n-2}{2} \rfloor \lfloor \frac{n-3}{2} \rfloor}{4}$$

The minimal crossing number for fully connected rectilinear graphs also holds for this formula, with the notable exception of graphs with 8 vertices.

For K_8, the crossing number is 18; while for rectilinear graphs the crossing number is 19. As the crossing problem increases in size (11 vertices or more), a different formula appears to hold true:

$$\bar{\nu}(K_n) \leq \lfloor \frac{(7n^4 - 56n^3 + 128n^2 + 48n\lfloor \frac{(n-7)}{3} \rfloor + 108)}{432} \rfloor$$

Guy [8] proposed this formula in 1972, and has conjectured that this formula is actually an equality. In private communication however, Guy [9] noted that if equality does not hold for graphs with 12 vertices, then graphs with more vertices are also likely to have smaller crossing numbers than the formula suggests.

NP-Completeness

NP-complete problems are a class of decision problems that are considered to be intractable. In other words, solutions to these problems probably will not be found by using a polynomial time algorithm. Although it has not been proven whether or not NP-complete problems are truly intractable, it would appear that a major breakthrough will be necessary to solve them in polynomial time. The crossing number problem has been classified as an NP-complete problem that is a transformation of the OPTIMAL LINEAR ARRANGEMENT problem [6], and Johnson has cited a variant of the crossing problem that is related to the GRAPH GENUS problem [11]. For a more complete overview of NP-completeness, see [6], and for a list of known NP-completeness problems see [6], and Johnson's series of *NP-Completeness Columns* [11].

Genetic Algorithms

Genetic algorithms were first developed by John Holland at the University of Michigan in the early 1970's. Holland was interested in how an algorithm could simulate natural selection. The goals of Holland's research included explaining the adaptive processes of natural systems and then designing artificial systems software which would retain the important mechanisms of natural selection [7]. Thus, the power of genetic algorithms lies in their robustness, or ability to adapt, just as in natural systems.

Genetic algorithms are heuristic search algorithms. Thus, the goal of a genetic algorithm is not to necessarily find the optimal solution, but to produce a "satisfactory" solution when searching a complex system. Random choice is used as a tool to guide a genetic algorithm as it searches. As a genetic algorithm produces new generations, better solutions may be discovered.

The structure of a genetic algorithm is based on natural selection. First, an initial population of feasible solutions is randomly generated. The initial population consists of chromosomes representing particular encodings of solutions. Reproduction takes place between members of the population, and a child is formed from a combination of the parent chromosomes. For each new child, an evaluation function is used to determine the fitness of that child. Whether or not the child becomes a member of the population depends on its fitness value. Each new child chromosome is compared against the worst member of the population, and the better one is kept in the population. By producing new generations in this manner, the population improves and the best member of the final population is the solution which is returned by the algorithm.

3 Method

Why use genetic algorithms implemented in parallel to find solutions to the rectilinear crossing problem? An initial solution might be to use an exhaustive search method to place vertices in a plane, and then evaluate the resulting graph. For large problem sizes(graphs of size 12 or more), this approach is obviously time consuming, and is not guaranteed to produce a "good" result within a reasonable amount of time.

An alternative to exhaustive search is a heuristic such as a genetic algorithm. The premise of the genetic algorithm is that by generating a large number of solutions (a population) and continually recombining the solutions, a "good" result will eventually be produced through "survival of the fittest" as good results replace "worse" results. A genetic algorithm working on a single processor may produce good results, but if multiple versions of the algorithm were to operate in parallel in some *cooperative fashion*, it is likely that the concurrent version would produce even better results than the single processor implementation.

The algorithm used for the rectilinear crossing problem is basically a genetic algorithm with some modifications that enhance its use on a parallel processing system. First, eight nodes of a parallel processing machine (iPSC/2) are allocated to run the genetic algorithms. Each node generates its own initial population, and begins executing the genetic algorithm. Each iteration of the genetic algorithm produces two children to evaluate and possibly insert into the node's population. Every so often, a "mutant" is generated and inserted into the population. Mutation is a technique to help prevent stagnation of the population. Once the genetic algorithm meets one of its convergence criteria (time limit, number of iterations, difference in the number of crossings between the best and the worst solutions), the algorithm halts and broadcasts its results to the host program. If all

nodes have converged to the same crossing number, the host stops and reports the results.

As previously noted, the algorithm used does not follow the traditional approach described by Goldberg [7] in which the *entire* population is replaced at each iteration. Instead, a combination of Davis' and Goldberg's approaches were used in which each genetic algorithm uses a Davis-like approach, [4], to insert a few new members into the population.

There is no strategy involved in creating initial populations. The graph vertices are randomly placed in a rectangular area, and the number of crossings for each resulting graph are determined. After the initial population is generated, the genetic algorithm selects two sets of parent "chromosomes" to recombine. One set is chosen via a simple linear bias, the other is chosen from a normal distribution of the best ten percent (10%) of the population.

Genetic algorithms were designed for use on sequential machines. In order to take advantage of parallel processing, a variation on traditional genetic algorithms can prove useful. By allowing some sort of cross-pollenation of chromosomes between genetic algorithms operating in parallel, information can be shared, and it is hoped that the interaction will improve performance of the genetic algorithms. This modification is logically consistent and exists in "real-world" genetics.

A pollenation rate, P, is set as a parameter to the program, and once every P iterations, each genetic algorithm sends a solution to the host program. The host then chooses the best solution and broadcasts it back to all the nodes to use in their recombination. This method of cross-pollenation has provided exceptionally better results then simply running eight genetic algorithms independently, an observation previously made by the authors [10]. One additional note: when cross-pollenation occurs too frequently, the populations tend to converge very quickly, and rarely do they produce a "good" result.

The recombination technique used in this work is based on uniform order-based crossover presented by Davis [4]. According to this recombination, a chromosome is a bit string that represents the contribution of the parents to a child. A one in the bit string indicates that the vertex corresponding to that bit index will contribute to the construction of child 1, and a zero indicates that the vertex will contribute to child 2. This bit string is generated randomly for each generation with parent 1 receiving the bit string and parent 2 receiving it's complement. Hence, child 1 is composed of vertices marked with ones and child 2 is made of vertices marked with zeroes. This type of chromosome recombination is illustrated in Figure 1. These children are then evaluated by the cost function and inserted into the population if the value returned is better than that of the worst member of the population.

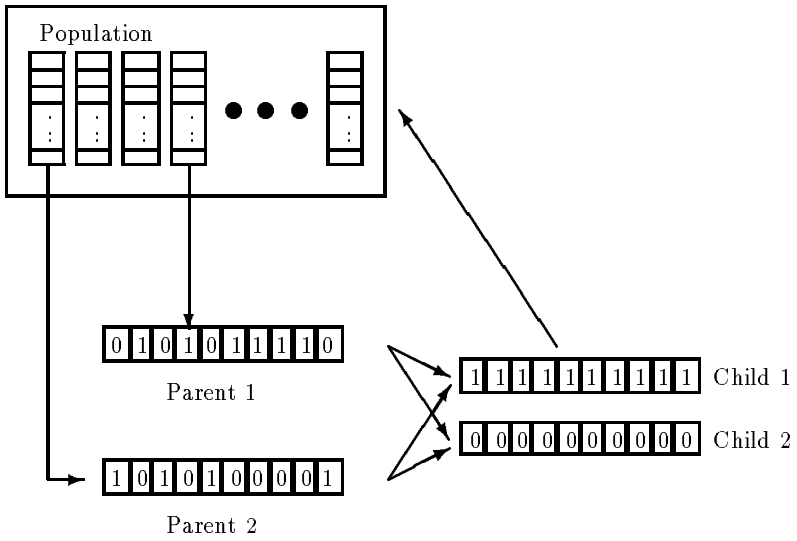


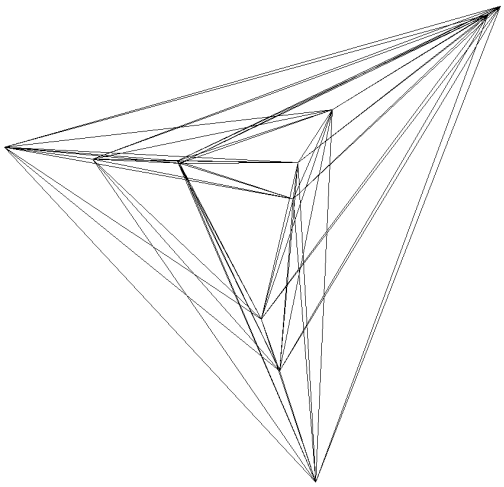
Figure 1: Recombination of Chromosomes

The cost function for the rectilinear crossing problem is fairly easy to describe. All edges are checked in a pairwise fashion such that no edge is compared to itself, and no pair of edges are compared more than once. If a pair of edges cross, then the crossing number is increased by one. Whenever the cost evaluation function detects colinear edges (something that violate the problem constraints) a large penalty is invoked. Solutions containing large penalty values tend to drop out of the population very quickly.

At first, verification was carried out by hand. As the problem sizes grew larger, simple manual verification became increasingly difficult. A simple graphics utility was written in the X-Windows environment to help verify solutions by plotting graphs of solutions and plotting intersection points.

4 Results and Analysis

Some of the parameters used for this presentation came from experiments done in previous work by one of the authors [10]. Whether they are actually "good" parameters for this particular problem has yet to be explored. For instance, in all of the experiments run, the mutation rate was held at 10% and the linear bias at 2.5. Parameters specific to this instance of the minimal crossing problem include the boundaries of the planar region occupied by the graph. The size of the area was limited to 1000×1000 units (an arbitrary decision). As noted in Table 1, there was some minor exper-



(894, 15)	(544, 339)	(610, 189)	(205, 272)
(582, 815)	(490, 541)	(521, 628)	(552, 278)
(873, 29)	(348, 275)	(352, 279)	(56, 252)

Figure 2: Complete Graph with 12 Vertices and 155 crossings

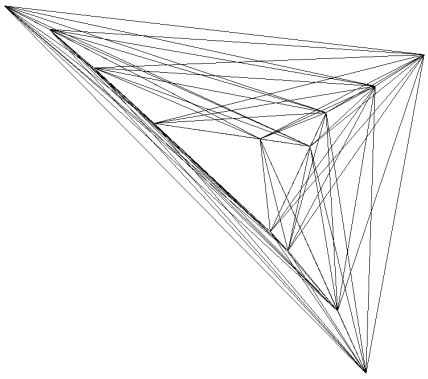
imentation with the pollenation rate. This experimentation showed that more frequent pollenation led to faster convergence and sporadic results. These preliminary results are probably not enough to justify the choice of pollenation every 150 generations.

Table 1 lists out the parameters and results for various executions of the parallel genetic algorithm. Guy's formula [8] produces the number in the Expected column, and output of the parallel genetic algorithm appears in the Result column. This column displays some rather interesting numbers for graphs of size 12 and 13. For these graphs the crossing numbers produced by the heuristic are lower than the minimum value proposed by Guy in 1972 [8]. These values have been verified by hand, and the graphs that correspond to those numbers are shown in Figures 2 and 3.

Another interesting feature of the genetic algorithms used is the number of iterations taken to produce a result. There appears to be no real correlation between the "goodness" of a solution and how long it takes a genetic algorithm to produce it. Intuition tells us that the longer a genetic algorithm executes, the better its results should be through "survival of the fittest." However, as previously stated, genetic algorithms are heuristic search techniques, and are not guaranteed to find an optimal solution. Much of the disparity between the results of each execution lies in the use

Size	Pop	Pollen	Expected	Result	Generations	Time.
3	150	150	0	0	1	0 sec
4	150	150	0	0	127	1 sec
5	150	150	1	1	247	5 sec
6	150	150	3	3	2,338	47 sec
7	150	150	9	9	5,250	148 sec
7	150	150	9	9	5,007	145 sec
10	100	400	60	62	134,409	3 hrs
10	100	400	60	62	143,609	4 hrs
11	150	100	102	120	196	2 min
11	150	5	102	124	252	2 min
11	150	5	102	102	7,770	15 min
11	150	150	102	102	13,299	15 min
11	100	400	102	104	14,009	15 min
11	150	150	102	102	152,550	2 hrs
12	150	150	156	157	138,599	2 hrs
12	150	150	156	156	194,151	3 hrs
12	150	150	156	**155**	228,739	3 hrs
13	150	150	231	231	139,711	3 hrs
13	150	150	231	231	189,809	3 hrs
13	150	150	231	**229**	196,639	3 hrs
14	150	150	328	344	89,125	3 hrs
14	150	150	328	334	131,250	3 hrs
14	150	150	328	333	147,999	4 hrs
15	150	150	453	465	97,679	3 hrs
15	150	150	453	469	126,099	4 hrs
16	150	150	612	638	77,639	3 hrs
16	150	150	612	636	81,099	3 hrs
17	150	150	808	874	45,289	3 hrs
17	150	150	808	864	81,769	4 hrs
18	150	150	1,047	1,220	57,809	4 hrs
18	150	150	1,047	1,172	59,829	4 hrs
19	150	150	1,338	1,562	32,449	3 hrs
19	150	150	1,338	1,568	43,939	4 hrs
20	150	150	1,683	2,054	22,349	3 hrs
20	150	150	1,683	2,018	35,619	4 hrs

Table 1: Parameters and Results from Parallel Genetic Algorithms



(541, 519)	(639, 311)	(611, 370)	(811, 210)
(337, 330)	(74, 125)	(725, 265)	(659, 657)
(709, 766)	(153, 166)	(231, 233)	(523, 357)
(572, 552)			

Figure 3: Complete Graph with 13 Vertices and 229 crossings

of randomness as a tool to guide the search. Some of the disparity is a result of the error checking and cost evaluation functions required by the minimal crossing problem. When a child's chromosome is generated, no two vertices in that chromosome can be the same. Whenever equal vertices are found, the child is discarded. If the child passes error checking, it goes through the cost evaluation function, which is a fairly expensive operation. Once evaluated, then the child *may* be inserted into the population–another expensive operation.

Cost evaluation is expensive due to the method that selects edges to test. Each edge is chosen in a pairwise fashion by vertices and tested against all other edges. A simple algorithm to do this is briefly outlined in Figure 4, and it is highly likely that there are better methods for evaluating the graph cost. The cost of this evaluation is characterized by the following formula:

$$\frac{n(n-1)(n-2)(n-3)}{2} \quad = \quad \frac{n^4 - 7n^3 + 12n^2 - 6n}{2}$$

Obviously cost evaluation is an $O(n^4)$ operation. In practice however, this cost evaluation does not grow at the expected rate of an $O(n^4)$ operation. This slow growth rate is due to the speed of the CPU which introduces

```
function cost()

  for i := 1 to (number of vertices - 1) do
    for j := (i + 1) to number of vertices do
      for k := 1 to (number of vertices - 1) do
        for m := (k + 1) to number of vertices do
          if ((edge(i,j) != edge(k,m)) AND
              not_yet_tested(i,j,k,m)) {
               total_cost := total_cost +
                             cost(edge(i,j),edge(k,m));
          }
        end /* for m */
      end /* for k */
    end /* for j */
  end /* for i */
  return(total_cost)
end /* function cost */
```

Figure 4: Cost Evaluation

a large constant that divides the entire function. This division appears to provide a lower order function for the limited values of n that we are pursuing. This savings is clearly illustrated by the time it takes to generate and test the initial population. Figure 5 shows a plot of initialization time versus problem size and compares it to n^4, n^3, and n^2. From this plot, it appears the checking mechanism's performance lies somewhere between $O(n^2)$ and $O(n^3)$ for the graph sizes considered.

5 Conclusion

The marriage of parallel processing and genetic algorithms seems to be a reasonably effective tool for obtaining results for intractable problems. The method presented is a somewhat crude tool; however, with refinement, it has the potential of becoming an even more valuable problem solving heuristic. As there was no comparison between the presented algorithm and an equivalent exhaustive search technique, few, if any, conclusions can be made about the superiority of one method over the other. The performance of a traditional exhaustive search method is known to have a speedup linearly proportional to the number of processors working in concert on the problem. Genetic algorithms do not provide any speedup; however, the results

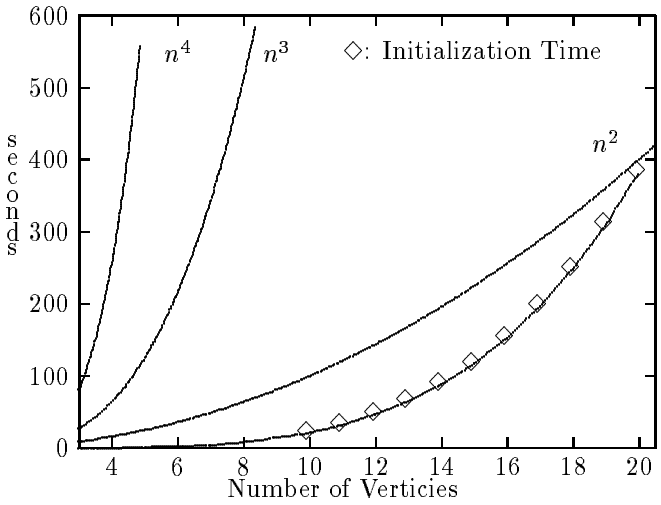


Figure 5: Initialization Time versus Number of Vertices

generated by this implementation encourage the conclusion that genetic algorithms are a valuable search tool.

Guy's boundary formula [8] was an outgrowth of his construction method which placed the graph vertices in a triangular pattern. The final mappings provided by the genetic algorithm are very close to the results one would expect using Guy's method. These results increase the confidence in the usefulness and correctness of the genetic algorithm presented. The fact that the genetic algorithm obtained results for K_{12} and K_{13} that were better than those conjectured by Guy [8] is very exciting. The results we have produced show that what was conjectured as an equality more than 20 years ago is merely a loose upper bound which can possibly be tightened in the future.

6 Future Work

As with any project, extensions and modifications can be made to increase its utility and/or its performance. As it stands, this project is limited to the rectilinear crossing problem. With minor modification, the program could handle more general cases of rectilinear graphs, or with more effort, it could explore the general instances of the crossing problem.

A further, and perhaps more significant experiment would be to make use of the general characteristics of "good" solutions. For instance, the

results produced by the program have distinct characteristics which could be exploited in generating the initial population(s). Research into the representation of the solution should also be performed. A genetic algorithm is only as good as its solution representation [10]. It is possible that the representation used for this project is not the best available, but without further investigation, this point is unclear.

Genetic algorithms are designed to provide acceptable results within a reasonable amount of time. Improving the performance of the genetic algorithm requires examination of the sorting methods, random number generation, communication, and the cost evaluation function. One improvement to the implementation would be to sort the vertices of the graph to help reduce error-checking time. Another area of investigation for improving performance is to show the effects of various parameters on convergence time.

At the moment, the current implementation of the parallel genetic algorithm is a fairly crude tool. However, even this simple heuristic seems to provide excellent results. Further refinement and experimentation needs to be done in order to create a more robust program. Additionally, development of an exhaustive search method to solve the rectilinear crossing problem would provide a reasonable standard to compare the results of the genetic algorithm with. Such a standard for comparison would prove invaluable to determining the desirability of using one heuristic over another.

Remarks

In private Communication which began after this paper was submitted, Geoffrey Exoo[5] informed us of some very nice results for the Rectilinear Minimal Crossing problem which improve upon some of the results we have presented, and go well beyond the size of problems we have considered.

References

[1] J. Blažek and M. Koman. A minimal problem concerning complete plane graphs. In M. Fiedler, editor, *Theory of Graphs and its Applications*, pages 113–117. Academic Press, New York, 1964.

[2] K.C. Chang. Efficient algorithms for layer assignment problems. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 6(1):67–71, January 1987.

[3] G. Chartrand and L. Lesniak. *Graphs and Digraphs*. Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, CA, 2nd. edition, 1986.

[4] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[5] G. Exoo. Private Communication, results to be submitted for publication soon.

[6] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1982.

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

[8] R.K. Guy. A minimal problem concerning complete plane graphs. In Y. Alavi, D.R. Lick, and A.T. White, editors, *Graph Theory and Applications*, pages 111–124, Berlin, 1972. Springer-Verlag.

[9] R.K. Guy, November 24, 1992. Private Communication with R.P. Pargas.

[10] J. Hines, J.T. Thorpe, and K.B. Winiecki. Solving quadratic assignment problems through parallel genetic algorithms. Unpublished Tech Report, Clemson University, May 1992.

[11] D.S. Johnson. The NP-Completeness column: an ongoing guide. *Journal of Algorithms*, 3(1):89–99, March 1982.

[12] Y. Wang, R.C.T. Lee, and J.S. Chang. The number of intersections between two rectangular paths. *IEEE Trans. on Computers*, 38(11):1564–1571, November 1989.