TimGA - A GENETIC ALGORITHM FOR DRAWING UNDIRECTED GRAPHS

Timo Eloranta and Erkki Mäkinen

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1996-10

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1996-10, DECEMBER 1996

TimGA - A GENETIC ALGORITHM FOR DRAWING UNDIRECTED GRAPHS

Timo Eloranta and Erkki Mäkinen

University of Tampere
Department of Computer Science
P.O. Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-4142-7
ISSN 0783-6910

Abstract: The problem of drawing graphs nicely contains several computationally intractable subproblems. Hence, it is natural to apply genetic algorithms to graph drawing. This paper introduces a genetic algorithm (TimGA) which nicely draws undirected graphs of moderate size. The aesthetic criteria used are the number of edge crossings, even distribution of nodes, and edge length deviation. Although TimGA usually works well, there are some unsolved problems related to the genetic crossover operation of graphs. Namely, our tests indicate that TimGA's search is mainly guided by the mutation operations.

Key Words and Phrases: genetic algorithm, graph drawing.
CR Categories: I.2.8, E.1, G.2.2

1. Introduction

The problem of drawing graphs nicely is completely solved only in some very special cases [7]. Irrespective of the aesthetic criteria used, the problem usually contains several computationally intractable subproblems [2]. This motivates the use of methods such as simulated annealing and genetic algorithms. For earlier works following this line of research, see e.g. [5,9,11,12,13].

This paper introduces a genetic algorithm TimGA (Timo's Genetic Algorithm) for drawing undirected graphs. TimGA owes some of its basic data structures to Groves et al.'s algorithm [9]. However, since undirected edges instead of directed ones are considered, most decisions differ from those made by Groves et al. TimGA outputs grid drawings with straight line edges.

We start by giving a short description of genetic algorithms. The general principle underlying genetic algorithms is that of maintaining a population of possible solutions, which are often called *chromosomes*. In our problem a population is a set of graph layouts. The population undergoes an evolutionary process which imitates the natural biological evolution. In each generation better chromosomes have greater possibilities to reproduce, while worse chromosomes have greater possibilities to die and to be replaced by new individuals. To distinguish between "good" and "bad" chromosomes we need an evaluation function. In graph drawing the evaluation function depends on the aesthetic criteria used; our evaluation function is discussed in greater detail in the next chapter.

The general structure of a genetic algorithm is as follows:

```
procedure ga
begin
    t:= 0;
    create the initial population P0;
    evaluate the initial population;
    while not Termination-condition do
        t:= t + 1;
        select individuals to be reproduced;
        recombine (i.e. apply genetic operations to create the new population Pt);
        evaluate(Pt)
    od
end;
```

There are several parameters to be fixed. First, we have to decide how to represent the set of possible solutions. In "pure" genetic algorithms only bit string representations were allowed, but we allow any representation that makes efficient computation possible. Hence, in the terminology of [14] TimGA is an "evolution program". Second, we have to choose an initial population. We use initial populations created by random selection. Third, we have to design the genetic operations which alter the composition of children during reproduction. The two basic genetic operations are the *mutation* operation and the *crossover* operation. Mutation is an unary operation which increases the variability of the population by making pointwise changes in the representation of the individuals. Crossover combines the features of two parents to form two new individuals by swapping corresponding segments of parents' representations. It turns out that the main problem in genetic graph drawing algorithms is to find efficient crossover operations. TimGA's genetic operations are introduced in the next chapter.

In what follows we assume that the reader is familiar with the basics of genetic algorithms and graph theory as given e.g. in [14] and [10], respectively.

2. Selection and the evaluation function

TimGA draws graphs in an $N \times N$ matrix. Each node is located in a square of the matrix and all edges are drawn as straight lines. To represent a graph with n nodes and m edges we use a $2 \times n$ matrix to indicate the positions of the nodes and a $2 \times m$ matrix to indicate the edges by storing pairs of nodes. The corresponding end points are then found from the node matrix. Figure 1 shows a simple example of the representation used. Groves et al. [9] have used similar representation for nodes.

Nodes

n	1	2	3	4	5	6	7	8
x	1	4	6	3	2	3	5	4
y	4	6	3	1	3	5	4	2

1	2	3	4	5	6	7	8
5	6	7	8	6	7	8	5

Edges

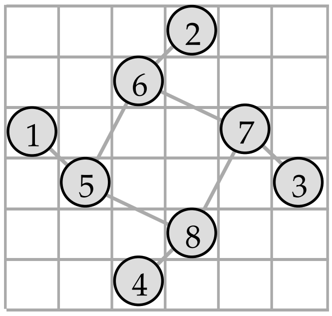


Figure 1. The representation of a sample graph.

One of the crucial points of a genetic algorithm is the method of selecting chromosomes to the genetic operations. TimGA uses the *linear normalization* suggested by Davis [6] together with *elitism*. The linear normalization works as follows. The chromosomes are sorted in decreasing order by their evaluation function values. The best chromosome gets a certain constant value (e.g. 100) and the other chromosomes get stepwise decreasing constant values (e.g. 98, 96, 94,...). Chromosomes are then selected to the genetic operations proportionally to the values so obtained. Depending on the length of the step (the difference between the consecutive constant values; two in the above example), this method can be parametrized to give a desired emphasis to the best chromosomes. TimGA allows the user to set the length of the step. By default, TimGA uses elitist selection, i.e., the best chromosome is always chosen as such to the next generation.

The aesthetic criteria used are imported to genetic graph drawing algorithms in the form of the evaluation function (also called the fitness function). TimGA tries to minimize the number of edge crossings, to distribute the nodes evenly over the drawing area, and to minimize the deviation of edge lengths.

The positive terms (to be maximized) in the evaluation function are

- Minimum Node Distance Sum (Min. Node Dist. Sum): The distance of each node from its nearest neighbour is measured, and the distances are added up. The bigger the sum the more evenly the nodes are usually distributed over the drawing area.
- Minimum Node Distance (Nbr of Nodes $*$ Min. Node Distance^2): This term helps in distributing the nodes. The square of minimum node distance is multiplied by the number of nodes.

The negative terms (to be minimized) in the evaluation function are

• Edge Length Deviation: The length of each edge is measured and compared to the "optimal" edge length, which is little more than the minimum edge length found from the present layout.

• Edge Crossings: The number of edge crossings is multiplied by the size of the drawing grid. (The grid is always a square.)

TimGA spends most of its computation time in evaluating the chromosomes. One of the problematic issues is the counting of the number of edge crossings. There is a well-known method based on cross productions to check whether two line segments intersect [4, pp. 889-890]. More advanced methods are introduced by Bentley and Ottmann [1] and Chazelle and Edelsbrunner [3]. Unfortunately, the method of Chazelle and Edelsbrunner, though asymptotically time optimal, is too complicated for the present application. On the other hand, the Bentley and Ottman's algorithm is too slow. Thus, we have to use a method of our own for counting the number of edge crossing. We keep track of the movements of the nodes, and update the number of edge crossings only when a node is moved. This method outperforms the Bentley and Ottman's algorithm in the present situation.

3. The genetic operations

The crossover operation transforms two chromosomes into two new chromosomes. TimGA has two types of crossover operations. *RectCrossover* works as follows. First it randomly chooses a rectangle from the drawing area of the parent chromosomes. Then a rectangle of equal size is chosen from the drawing area of the child chromosomes. The parent chromosomes exchange the positions of the nodes inside the chosen rectangles. The rest of the nodes are kept unchanged, if possible. A sample RectCrossover is shown in Figure 2.

The sample RectCrossover operation of Figure 2 uses rectangles of size 3×3; these are painted grey in the figure. The parents change the positions of the nodes 3 and 6 (from Parent-1) and nodes 1 and 3 (form Parent-2). The nodes 1 and 3 keep their relative positions in the grey area when it is moved from Parent-2 to Child-1. Moreover, since the chosen rectangle in Child-1 is empty, the rest of the nodes in Child-1 can keep their old positions, i.e. the positions they have in Parent-1. On the other hand, in Child-2 there are two nodes in the chosen 3×3 rectangle (nodes 2 and 7). These must be moved outside the area. The first possible place is the square where the corresponding node is in the other parent. Since node 2 of Parent-1 is in the square (2,5), this is the new position of the node in Child-2. This method does not work with node 7, since the square (8,4) is

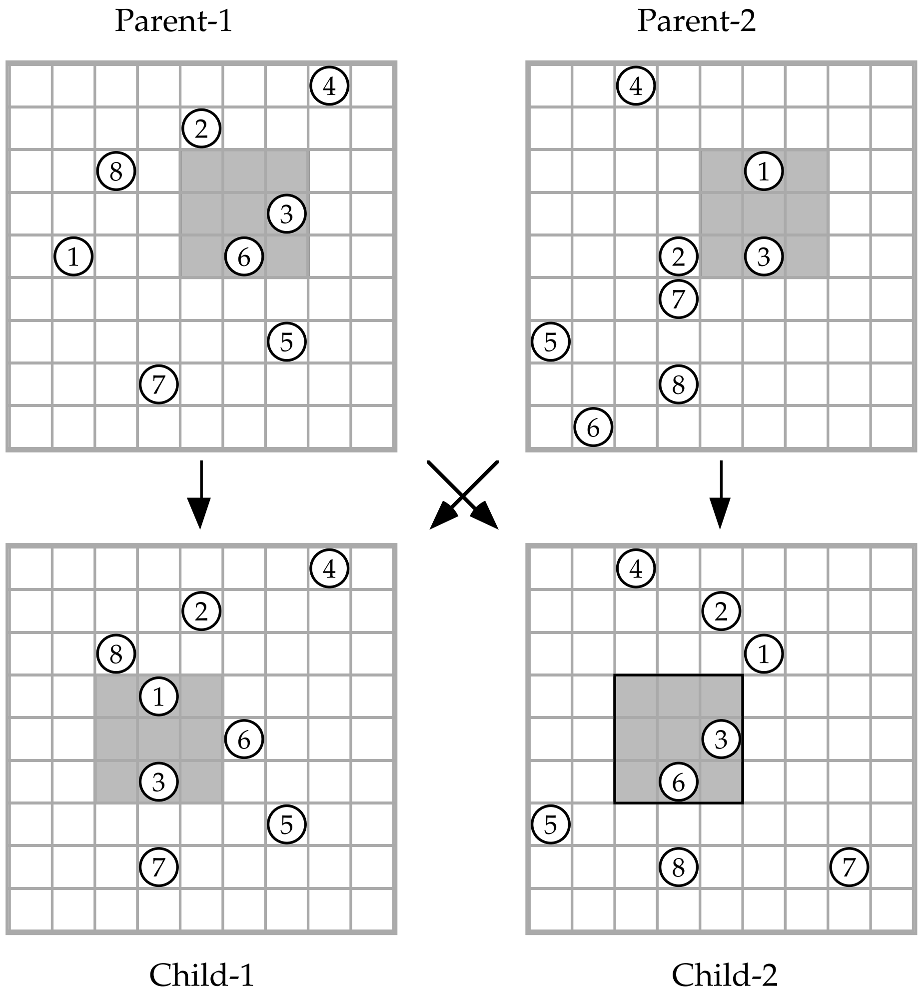


Figure 2. A sample RectCrossover.

already occupied by node 8. So, we have to place node 7 to an randomly chosen free square (8,8). RectCrossover closely resembles the Cont-Crossover operation of [9].

The other crossover operation in TimGA is called *ThreeNodeCrossover*. A connected subgraph consisting of three nodes is chosen. The parents then exchange the positions of the three nodes in question. If some of the new positions are already occupied, the nodes in question are kept unchanged. A sample ThreeNodeCrossover is shown in Figure 3.

Groves et al. [9] introduced about a dozen different mutation operations. In our tests we have used 16 different mutations of which 11 are from [9] and the five rest are new. Our tests indicate that mutation operations applied to edges usually have better performance than those applied to nodes. The following eight mutation operations performed best in our tests:

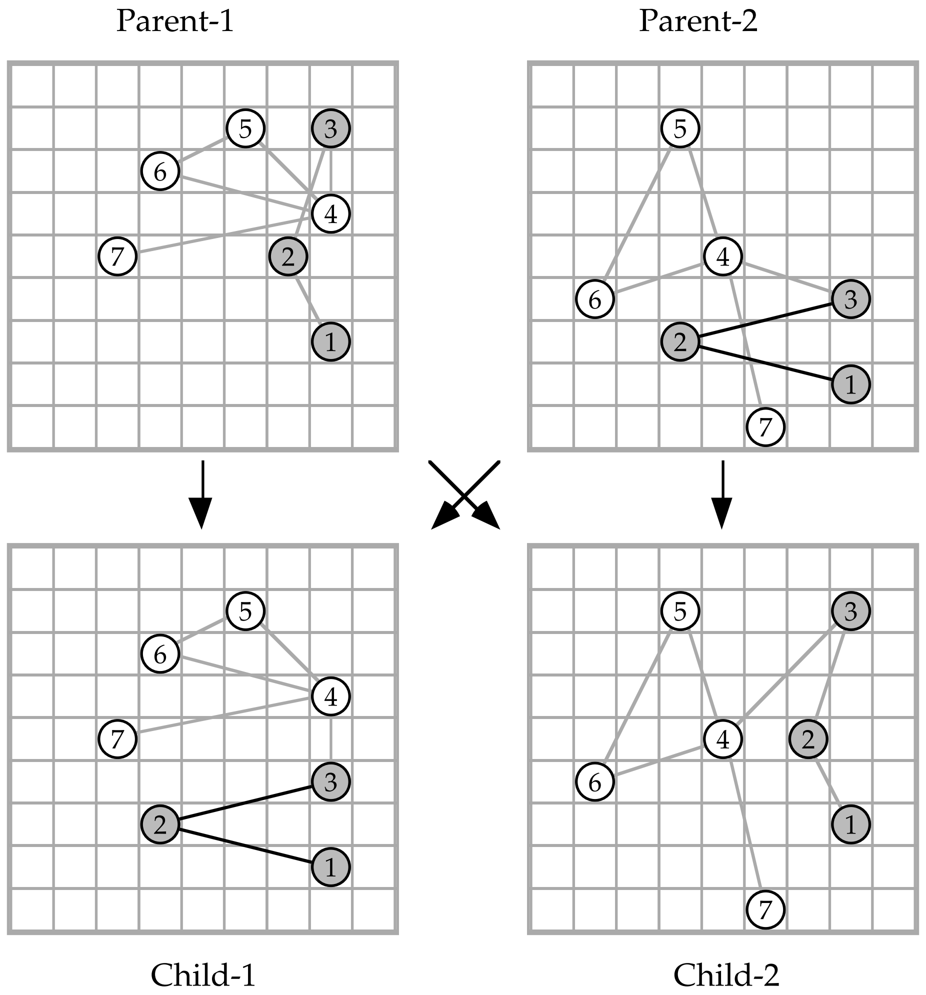


Figure 3. A sample ThreeNodeCrossover.

• SingleMutate: Choose a random node and move it to a random empty square [9].

• SmallMutate: Choose randomly two squares from the drawing area such that at least one of them contains a node. If both contain a node, exchange the nodes. If only one of them contains a node, then move the node from the present location to the empty square [9].

• LargeContMutate: Choose two areas of equal size and shape from the grid. Exchange the contents of the chosen areas [9].

• EdgeMutation-1: Choose a random edge and move it to a random new position.

• EdgeMutation-2: Like EdgeMutation-1, but the length and angle of the edge is kept unchanged, if possible.

• TinyEdgeMove: Like EdgeMutation-2, but the edge is moved only at most one square both horizontally and vertically.

• TwoEdgeMutation: Like EdgeMutation-2, but two edges incident with a same node are moved.

• TinyMutate: Like SingleMutate, but the node is moved only at most one square both horizontally and vertically.

The probability of using a certain mutation type depends on its performance in our tests. The operations introduced above have the following relative probabilities (the bigger the probability the better performance in our tests):

TwoEdgeMutation	12/65
EdgeMuation-2	10/65
SingleMutate	10/65
EdgeMutation-1	5/65
LargeContMutate	5/65
SmallMutate	5/65
TinyMoveEdge	5/65
TinyMutate	5/65.

Moreover, eight additional mutation operations introduced in [9] are used with relative probability 1/65. Note that the mutation operations clearly have different roles: some of them are more suitable for tentative searching and some others for fine tuning.

5. Parameters

This chapter deals with the test runs which were done to fix the various parameters of TimGA. We did over 4000 runs using mainly the following test graphs:

- a cycle with 48 edges
- a triangular grid with 28 nodes and 63 edges
- a complete binary tree with 63 nodes.

The number of edge crossings was the only criterion used in evaluating the results. This naturally follows from straightforwardness of measuring the criterion in question. We believe that despite of the small number of test graphs used, the results can be generalized also to other graphs of approximately the same size.

The size of the grid. What is the optimal size of the drawing area for our test graphs? This was tested for grids from 10×10 to 70×70. The optimum size was 40×40, and this size was used in all the tests to be reported. There were only small differences between all the grid sizes from 20×20 to 70×70; grids smaller than 10×10 were clearly inferior (for obvious reasons).

The size of population. Population size should be large enough to give an unbiased view of the search space. On the other hand, too large population size makes the algorithm inefficient, if not intractable. Surprisingly, TimGA seems to works best with very small populations. Figure 4 shows the number of edge crossings with different population sizes after the running time of 15 seconds on a Power Macintosh with our complete tree test graph. (All the tests were executed on a 100 MHz Power Macintosh.)

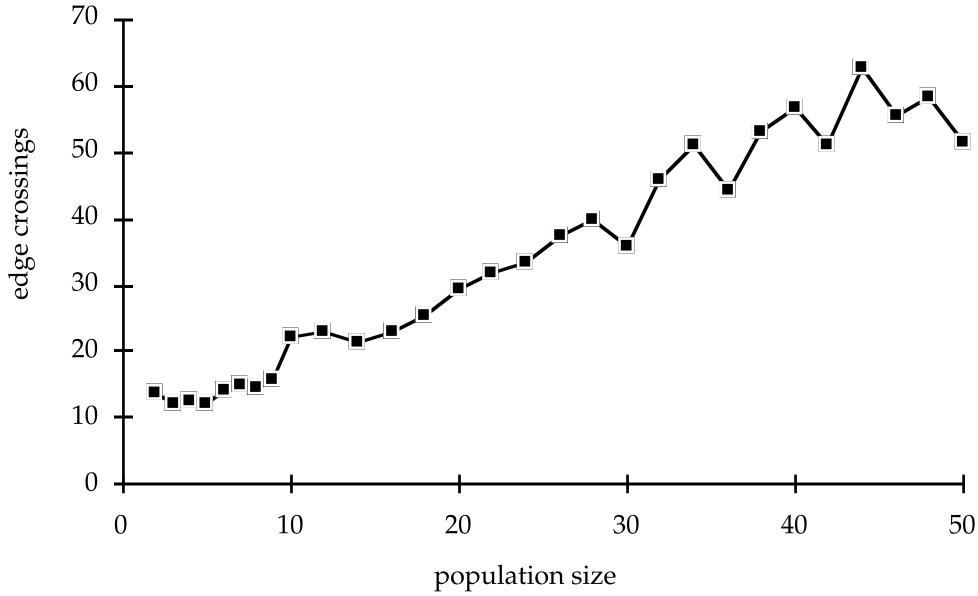


Figure 4. The number of edge crossings as a function of the population size.

These results suggest that the population size should not exceed 10. We use the population size 10 in the rest of our tests. Such a small population size might not fit the Schema Theorem, the Building Block Hypothesis [14], and other theoretical principles of genetic algorithms. However small populations give us the best results! We interpret this phenomenon so that the crossover operations used are unable to sift the good properties (called schemata in [14]) of the chromosomes from parents to children, and the search is mainly guided by the mutation operations.

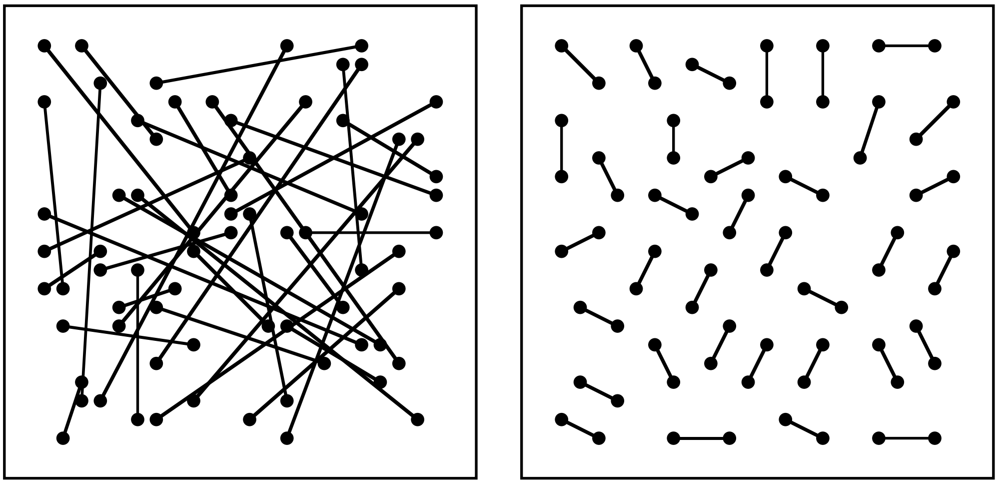
Selection. Our tests advice to use large steps in the linear normalization. This means that the best chromosomes are strongly favoured. This can be considered as a further evidence for the fact that our crossover operations do not help. (Michalewicz [14, p. 57] has noted that the use of selection methods neglecting the actual relative differences between the fitness of chromes is also against the theoretical basis of genetic algorithms. The linear normalization is one of these methods.)

Crossover and mutation rates. As already mentioned, our crossover operations seem to have no positive effect to the search process. On the other hand, increasing the mutation rate makes the search more efficient all the way to the level 40 - 45%. Still increasing the mutation rate over 45 % again makes the results worse. The crossover rate 5 % and mutation rate 45 % are TimGA's default values.

6. Example layouts

In this chapter we present the results of applying TimGA to some typical graphs. All the drawings (and their computation times) reported in this chapter are produced on a Power Macintosh. The computation times given in this chapter are not averaged over several runs as was done in the results reported in the previous chapters. This means that randomly selected initial populations may distort the results.

Our first example demonstrates the aesthetic criteria used. In Figure 5(a) a set of separate edges is shown. From this input TimGA outputs the drawing shown in Figure 5(b). There are no edge crossings, the edges are distributed evenly over the drawing area, and the edges are of about the same length. The drawing of Figure 5(b) was created in 20 seconds; eliminating all the edge crossings took about a second.



(a) (b)

Figure 5. A sample input and the corresponding output.

Figure 6 shows how TimGA tends to draw a cycle. Since evaluation function has no component aiming at symmetric drawings, the nodes of the output graph are not drawn in a round shape.

Figure 7 demonstrates the effect of the grid size. The same graph (the cubic graph) is drawn using the grid sizes 12 × 12 (Figure 7(a)) and 20 × 20 (Figure 7(b)). The Figure 7(b) suffers from the tendency of drawing graphs with edges of equal length. This tendency is more easily realized in a grid with more squares.

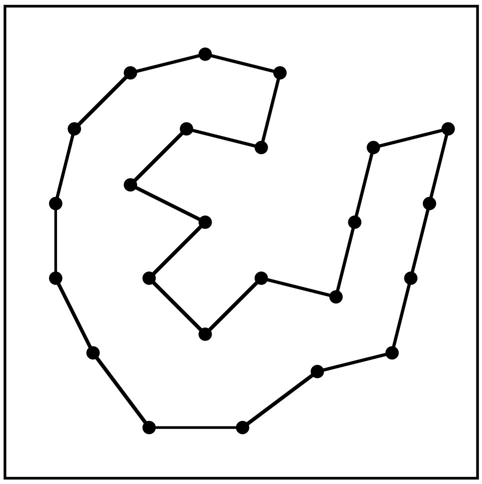
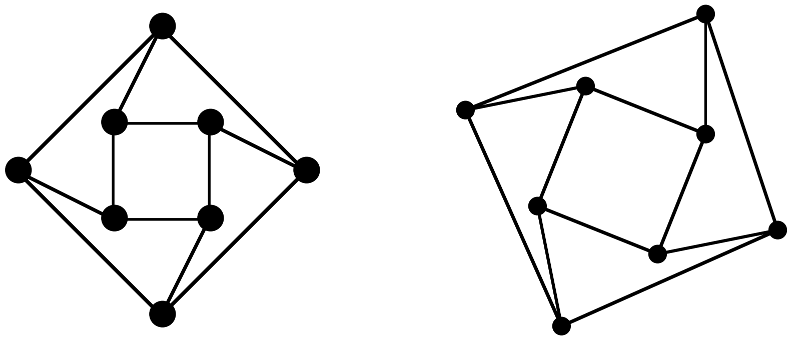


Figure 6. An output for a cycle.



(a) (b)

Figure 7. The effect of the grid size.

Figure 8 shows three drawings for square grid graphs of different sizes. The graph of Figure 8(a) is drawn in a drawing area of size 22 × 22, while the other two are drawn in a drawing area of size 40 × 40. Figure 8(a) was produced in 8 seconds using less than 5000 generations. Figure 8(b) took almost 90 seconds although the result is not completely symmetric. Even worse is the situation with Figure 8(c): after the running time of 10 minutes TimGA was still unable to find a planar drawing. The evaluation function does not "understand" that moving the top right node of the grid graph upwards would only temporarily cause more edge crossings.

Figure 9 shows a nice drawing of a triangular grid graph with 35 nodes and 135 edges. The computation time was about three and a half minutes (5400 generations).

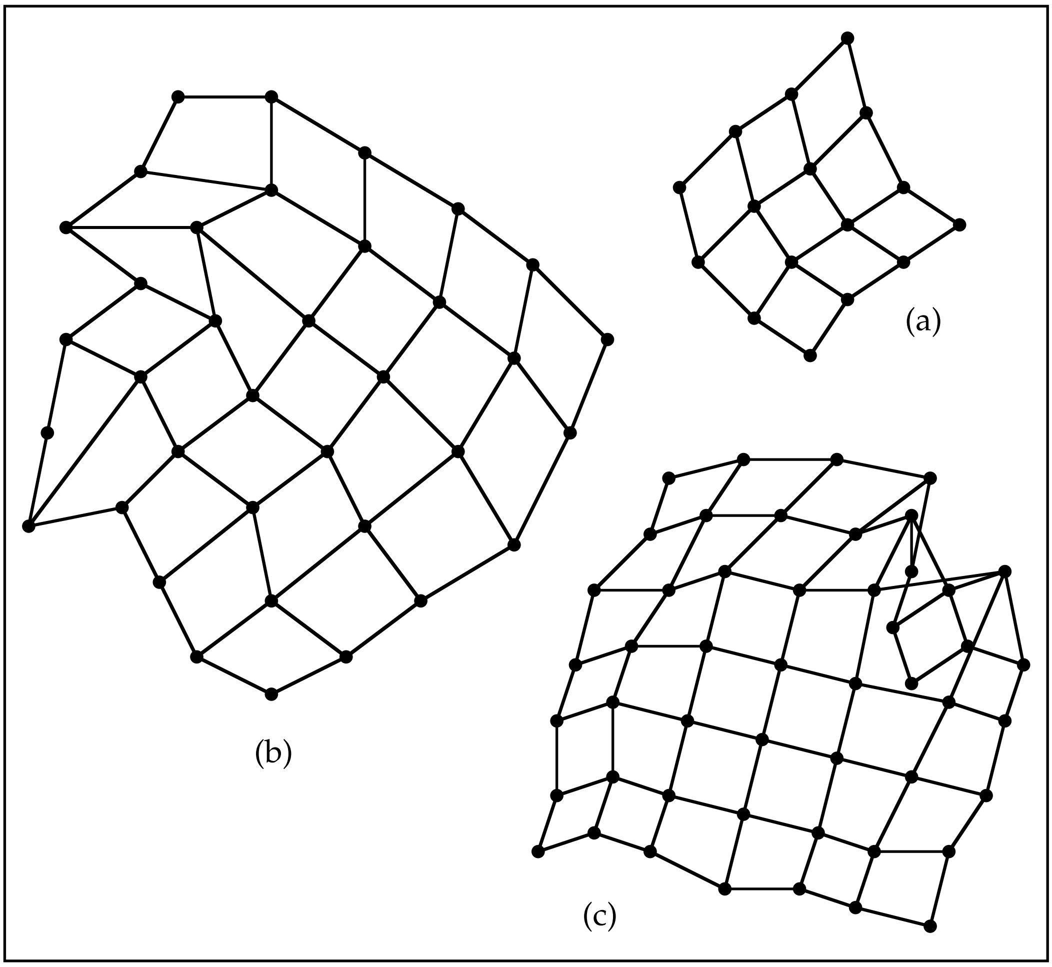


Figure 8. Three sample drawings of grid graphs.

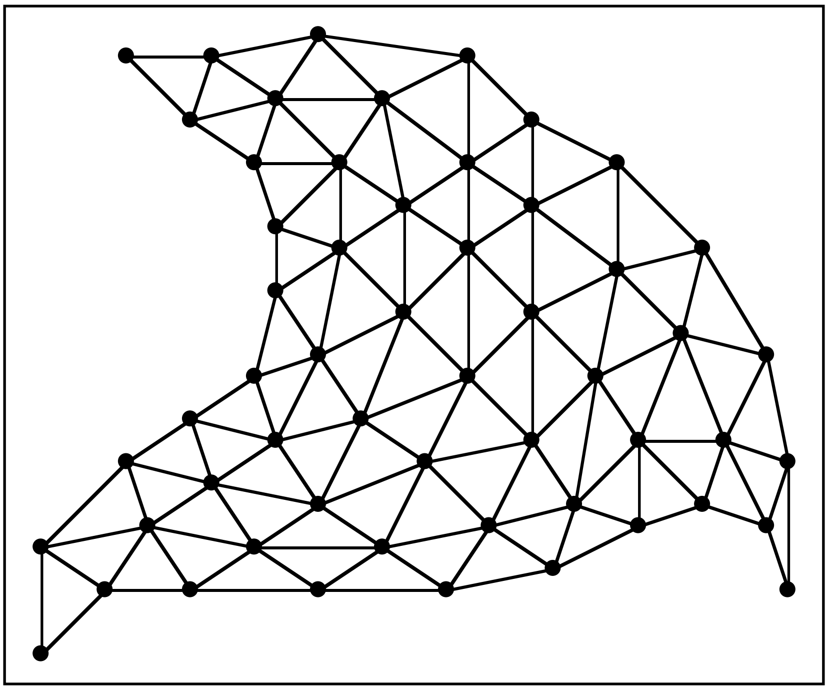


Figure 9. A layout for a triangular grid graph.

We end this chapter with some remarks concerning the Edge Crossing Problem (ECP). Given an undirected graph G, ECP is the problem of determining the minimum number of edge crossings (denoted by $\nu(G)$) among layouts of G. ECP is known to be NP-complete [8]. The following approximation is known for the crossing number of complete bipartite graphs [10, p. 123]

$$\nu(K_{m,n}) \leq \lfloor\frac{m}{2}\rfloor \lfloor\frac{m-1}{2}\rfloor \lfloor\frac{n}{2}\rfloor \lfloor\frac{n-1}{2}\rfloor.$$

TimGA easily reaches the above bound for graphs $K_{m,m}$, where $m \leq 12$. Figure 10 shows a drawing for $K_{8,8}$.

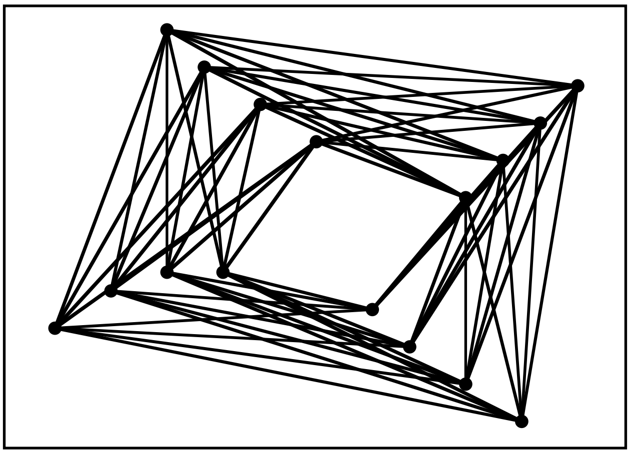


Figure 10. A layout for $K_{8,8}$.

7. Conclusions

TimGA nicely draws most graphs of moderate size. However, it suffers from the lack of a proper crossover operation which would speed up TimGA's computations by decreasing the number of generations needed.

TimGA is available with full source code in ftp://cs.uta.fi/pub/TimGA/.

References

[1] J. L. Bentley and T. A. Ottmann, Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.* **C-28** (1979), 643-647.

[2] F. J. Brandenburg, Nice drawings of graphs and trees are computationally hard. Tech. report MIP-8820, Fakultät für Mathematik und Informatik, Univ. Passau, 1988.

[3] B. Chazelle and H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane. *J. ACM* **39**, 1 (1992), 1-54.

[4] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*. The MIT Press, 1990.

[5] R. Davidson and D. Harel, Drawing graphs nicely using simulated annealing. The Weizmann Institute of Science, Dept. of Applied Mathematics and Computer Science, Revised version of Tech. Report CS-89-13, July 1993 (to appear in *Comm. ACM*).

[6] L. Davis, A genetic algorithms tutorial. *Handbook of Genetic Algorithms*, L. Davis (ed.), Van Nostrand Reinhold, 1991, 1-101.

[7] G. Di Battista, P. Eades, R. Tamassia and I. G. Tollis, Annotated bibliography on graph drawing algorithms. *Comput. Geom. Theory Appl.* **4** (1994), 235-282.

[8] M. R. Garey and D. S. Johnson, Crossing number is NP-complete. *SIAM J. Alg. Disc. Meth.* **4**, 3 (September 1983), 312-316.

[9] L. Groves, Z. Michalewicz, P. Elia and C. Janikow, Genetic algorithms for drawing directed graphs. *Proceedings of the Fifth International Symposium on Methodologies for Intelligent Systems*, Elsevier North-Holland, 1990, 268-276.

[10] F. Harary, *Graph Theory*. Addison-Wesley, 1969.

[11] C. Kosak, J. Marks, and S. Shieber, A parallel genetic algorithm for network-diagram layout. *Proc. 4th Int. Conf. on Genetic Algorithms*, 1991.

[12] E. Mäkinen and M. Sieranta, Genetic algorithms for drawing bipartite graphs. *Intern. J. Computer Math.* **53** (1994), 157-166.

[13] A. Márkus, Experiments with genetic algorithms for displaying graphs. *Proceedings of the 1991 IEEE Workshop on Visual Languages*, IEEE Computer Society Press, 1991, 62-67.

[14] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1992.