GENETIC ALGORITHMS FOR DRAWING BIPARTITE GRAPHS

Erkki Mäkinen and Mika Sieranta

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1994-1

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1994-1, JANUARY 1994

GENETIC ALGORITHMS FOR DRAWING BIPARTITE GRAPHS

Erkki Mäkinen and Mika Sieranta

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

ISBN 951-44-3497-8
ISSN 0783-6910

GENETIC ALGORITHMS FOR DRAWING BIPARTITE GRAPHS

Erkki Mäkinen[#] and Mika Sieranta
University of Tampere, Department of Computer Science
P.O. Box 607, FIN-33101 Tampere, Finland

Abstract. This paper introduces genetic algorithms for the level permutation problem (LPP). The problem is to minimize the number of edge crossings in a bipartite graph when the order of vertices in one of the two vertex subsets is fixed. We show that genetic algorithms outperform the previously known heuristics especially when applied to low density graphs. Values for various parameters of genetic LPP algorithms are tested.

CR Categories: I.2.8, E.1, G.2.2

Key Words and Phrases: bipartite graph, graph drawing, genetic algorithm.

1. Introduction

Drawing a graph with as few edge crossings as possible is a well-known problem of graph theory [1, 7, 14]. Watkins [16] was the first who restricted the edge crossing minimization problem to the case of bipartite graphs. Recently, this problem has found several applications in various graph drawing systems [2], most notably in those based on the Sugiyama's algorithm [13]. Similar problems are also considered in the connection with VLSI design [8].

Let $G = (V, E)$ be a bipartite graph, where $V = L \cup U$, $L \cap U = \emptyset$. We can define two drawing problems as follows. Assume that the vertices in U appear in some order on the horizontal line $y = \alpha$, $\alpha > 0$. Similarly, the vertices in L appear in some order on the horizontal line $y = 0$. The number of edge crossings depends only on the order of vertices in U and L. In the *bipartite drawing problem* (BDP) our task is to find linear orders f_U: $U \rightarrow \{1,2,..., U \}$ and f_L: $L \rightarrow \{1,2,..., L \}$ of the vertex subsets minimizing the number of edge crossings in the corresponding drawing. For the other problem, we assume that the order of the vertices in U is fixed. Hence, our task is to find the linear order f_L: $L \rightarrow \{1,2,..., L \}$ minimizing the number of edge crossings in the corresponding drawing. This problem is referred to as the *level permutation problem* (LPP).

[#] To whom all correspondence should be addressed (e-mail: em@cs.uta.fi).

Both BDP and LPP are known to be NP-complete [7, 4]. Hence, we have to use heuristic algorithms for solving them. Various heuristics for LPP are introduced in the literature [3, 5, 10, 13, 15]. We can use LPP heuristics also when solving BDP. Namely, we can in turn fix one of the two subsets, and repeatedly apply heuristics for LPP. In this paper we restrict ourselves to LPP.

Most of the known heuristics for LPP are fast and some of them often produce fair drawings. However, in some applications it might be advantageous to use more time for solving LPP in order to obtain a drawing with fewer edge crossings than it is possible to get with the fast heuristics. To this end May and Szkatula [11] have applied simulated annealing to BDP. We continue this line of research and apply genetic algorithms to LPP.

The purpose of this paper is twofold. First, we try to find better solutions for LPP. And second, we test the applicability of different implementation details of genetic algorithms in a typical combinatorial problem.

The rest of the paper is organized as follows. Section 2 introduces some of the known heuristics for LPP. In section 3 we shortly present the main properties of genetic algorithms following [12]. In section 4 we describe our test runs, and section 5 shortly summarizes our findings.

2. Heuristics for drawing bipartite graphs

Let $G = (L \cup U, E)$ be a bipartite graph where the vertices in U are ordered according to the linear order f_U. We recall some heuristics for LPP, i.e. for ordering the vertices in L while keeping U fixed. There are two basic categories of LPP heuristics. *Context-free* heuristics assign the same approximate value to each vertex irrespective of the other vertices of the graph in question. The solution is then obtained by ordering the vertices according to these values. On the other hand, *context-sensitive* heuristics first determine the exact numbers of edge crossings connected to the relative orders of the vertex pairs in question. Based on these numbers context-sensitive heuristics then try to find a linear order which minimizes the number of edge crossings.

We start with two context-free heuristics. The *barycenter heuristic* (BC) [13] orders the vertices according to the average (arithmetic mean) of the positions of the vertices incident with them. Figure 1 shows a sample graph with 10 edge crossings. Let av(u) stand for the average of the positions incident with u. In the case of Figure 1 we have av(a) = 4, av(b) = 2.3, av(c) = 3 and av(d) = 3.5. Figure 2 shows the graph of Figure 1 with L permuted according to the BC heuristic. The number of edge crossings is now only five. The BC heuristic is called *averaging* in [3].

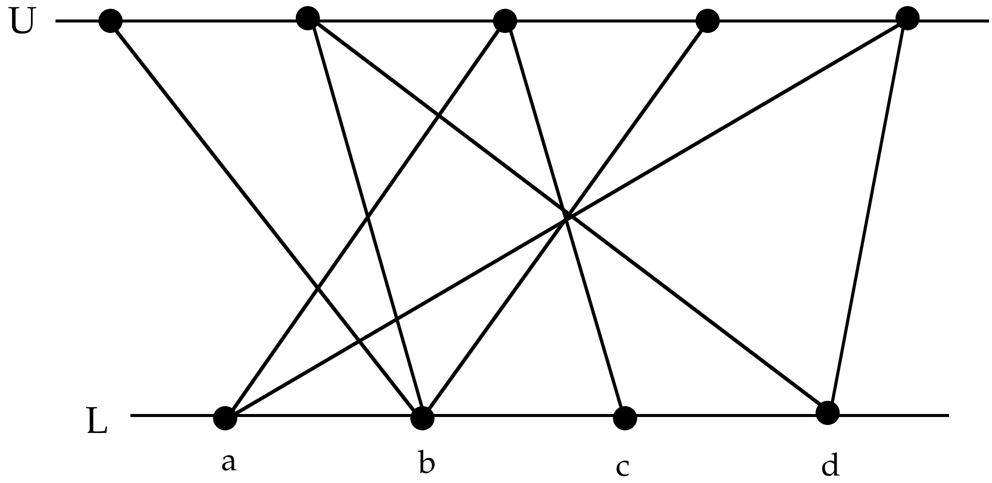

Figure 1. A sample graph.

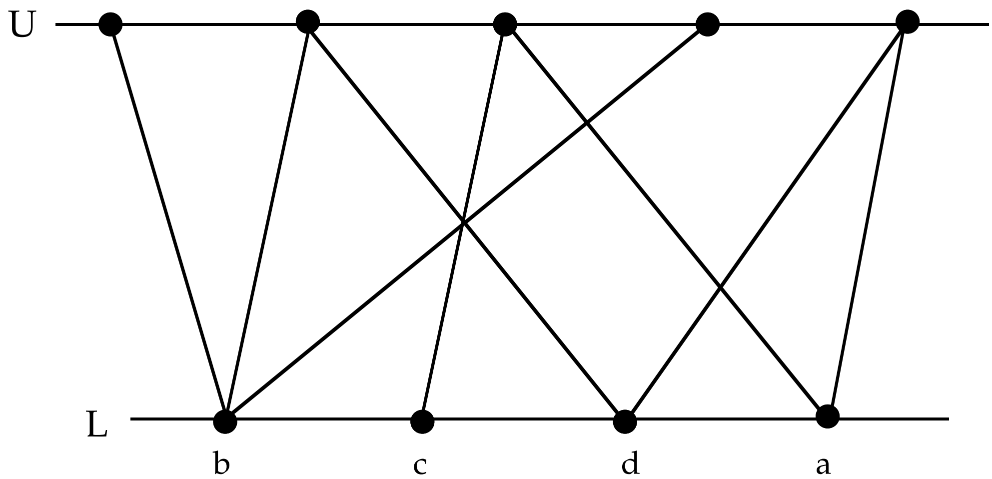

Figure 2. The graph of Figure 1 drawn by the BC heuristic.

The *median heuristic* [5] permutes the vertices according to the medians of the positions. If a vertex in L is incident with an even number of vertices we use the upper median. Let me(u) stand for the median of the positions incident with u. In the case of Figure 1 we have me(a) = 5, me(b) = 2, me(c) = 3 and me(d) = 5. The resulting number of edge crossings now depends on how we break tie between me(a) and me(d). If the tie is broken correctly, we obtain the drawing shown in Figure 2. Otherwise, the resulting drawing has six edge crossings.

Tests have shown that the BC heuristic usually performs slightly better than the median heuristic, although only in the median heuristic the relative error done has a constant upper bound [5, 6, 9].

As an example of *context-sensitive* heuristics we consider splitting [3] which follows the divide-and-conquer strategy. We start by giving some notations. Let u and v be vertices in L. The number of edge crossings caused by the edges incident with u with the edges incident with v depends only on the relative order of u and v. Let c_{uv} denote the number of these crossings if u is on the left of v. The splitting heuristic works as follows. Arbitrarily choose a vertex u (called the *pivot*) and divide the vertex set into two subsets $L(u) = \{ v \quad c_{uv} \leq c_{vu} \}$ and $H(u) = \{ v \quad c_{uv} > c_{vu} \}$. The heuristic is then recursively applied to L(u) and H(u) until a linear order is found. The number of recursive calls and hence, the time needed to apply the splitting heuristic, depends on the choice of pivots. In the worst case we need $\Theta(n)$ calls to handle a set of n vertices. In any case, the splitting heuristic always takes much more time than the BC heuristic.

The *density* of bipartite graphs has its influence to the success of the heuristics (and also to the success of genetic algorithms as we shall see later on). The complete bipartite graph $K_{m,n}$ has density 100 %. The density per cent of an arbitrary bipartite graph G is defined by the ratio between the numbers of edges in G and in the corresponding complete bipartite graph $K_{m,n}$.

Although in some special cases the BC heuristic can be "fooled" to make arbitrarily large relative errors (for details, see [5, 6, 10]), it usually works better than the other fast heuristics (see [10] for a comparision of the heuristics). In the rest of the paper we compare the performance of genetic algorithms to that of the BC heuristic.

3. Genetic algorithms

The general principle underlying genetic algorithms is that of maintaining a population of possible solutions. (In LPP a population is a set of permutations of the vertices in L.) The population undergoes an evolutionary process which imitates the natural biological evolution. In each generation better solutions have greater possibilities to reproduce, while worse solutions have greater possibilities to die and to be replaced by new individuals. To distinguish between "good" and "bad" solutions we need an evaluation function. (In LPP we simply count the number of edge crossings related to a permutation.)

The general structure of genetic algorithms 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 implementation details 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 [12] our algorithms are "evolution programs". (In LPP it is natural to represent the set of possible solutions as permutations, i.e. as integer vectors.) Second, we have to choose an initial population. (In our tests the initial population is always created by random selection.) Third, we have to design the genetic operations which alter the composition of children during reproduction. In this paper we use two basic genetic operations, 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. The genetic operations used in LPP are discussed in the following section. We also need to specify values for different parameters such as population size, mutation rate, and so on.

In what follows we assume that the reader is familiar with the basics of genetic algorithms as given e.g. in [12].

4. Test runs

In this section we describe our test runs. Our aim is to fix the parameters needed in genetic algorithms. In our tests we use bipartite graphs $G = (L \cup U, E)$ where $|L| = |U|$, i.e. the size of the two vertex subsets is equal. This is called the *size* of G.

4.1. Genetic operations

As already mentioned, we represent the possible solutions of LPP as permutations. In the genetic algorithm literature, similar representations are used e.g. for the possible tours in the travelling salesperson problem. To define the mutation operation, consider a permutation $(x_1, x_2,..., x_n)$. For each integer in the permutation, we generate a random real number from the range [0..1]. If the random number is not greater than the mutation rate (to be fixed later) we perform a mutation operation as follows. Suppose the condition is triggered at point x_i. We generate a random integer j, $j + i$, from the range [1..n]. This integer tells the new position of x_i. If $j > i$, we move the items x_k, $k = i + 1, ..., j$, one step to the left. Similarly, if $j < i$, we move the items x_k, $k = j, ..., i - 1$, one step to the right.

The crossover operation transforms two individuals into two new individuals as follows. Let $(x_1, x_2,..., x_n)$ and $(y_1, y_2,..., y_n)$ be the parents to be transformed. First, we generate two random integers, say i and j, $i < j$, from the range [1..n]. The new individuals are obtained by taking the segments $x_i,..., x_j$ and $y_i,..., y_j$ from the parents and the rest of the permutation from the other parent preserving the relative order of the inserted integers. As an example, consider the case where parents are (1, 2, 3, 4, 5, 6) and (2, 4, 5, 1, 6, 3). Suppose further that we have $i = 3$ and $j = 5$. The segment from the first parent is (3, 4, 5) and the segment from the latter one is (5, 1, 6). The segment (3, 4, 5) is augmented to be an individual by inserting 2 and 1 to the beginning and 6 to the rear. Hence, we obtain the offspring (2, 1, 3, 4, 5, 6). The segment (5, 1, 6) is augmented by integers 2, 3, and 4 in that order. The resulting offspring is (2, 3, 5, 1, 6, 4).

Several sampling methods for selecting individuals for reproduction are introduced in the literature (see e.g. [12]). We use the following elitist policy to reproduce the population: The best individual is always selected as such to the new population. The rest of the present population is divided into two subpopulations A ("the good ones") and B ("the bad ones") by using our evaluation function. Hence, A (resp. B) contains the individuals whose number of edge crossings is below (resp. above) the average of the whole population. New individuals are now created by performing crossover productions in which one parent is randomly selected from A and the other one is randomly selected from the whole population. Our preliminary tests indicated that this policy outperforms other policies tested. Especially, genetic algorithms using this policy converge faster than other versions.

4.2. Population size

Population size is an important parameter of a genetic algorithm. Population size should be large enough in order to give an unbiased view of the search space. On the other hand, too large population size makes the algorithm computationally intractable. We tested the effect of population size by counting the number of generations (i.e. iterations of our main loop) needed to find an individual which has the same number of edge crossings as the solution obtained by the BC heuristic.

Figure 3 shows the number of iterations needed to reach the BC result and the number of iterations needed to find the best result of the test. The number of iterations needed first decreases when the population size increases. The population size 50 is a turning point after which hardly any improvement happens. Our tests suggest that 50 is good population size. (In these tests we used constant graph density (20%), graph size (15), and mutation rate (0.03)).

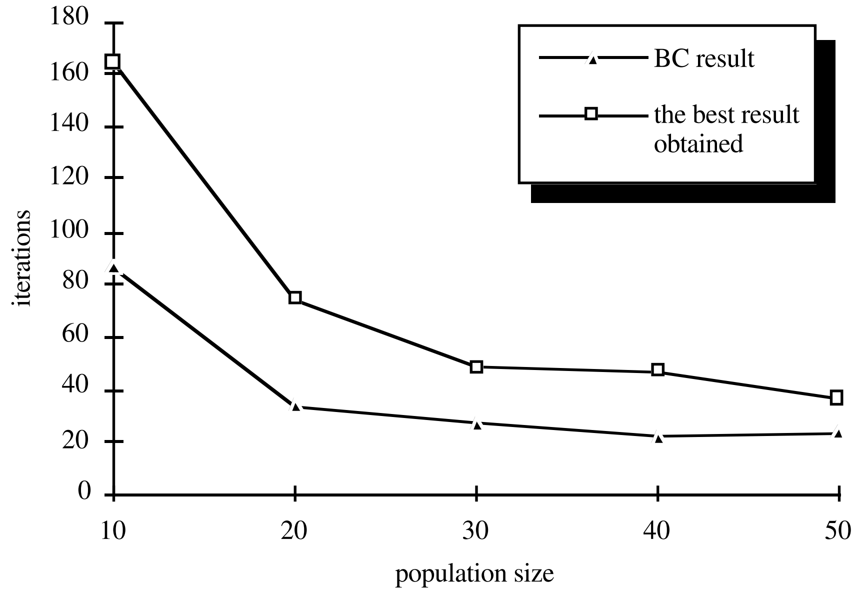

Figure 3. The number of iterations needed as a function of population size.

A striking phenomenon appeared during this test: the optimum was also found by using very small populations which had only a few individuals. The number of iterations needed was, of course, much larger than with population size 50. However, algorithms with very small populations can hardly be considered as genetic algorithms. Schema Theorem, Building Block Hypothesis [12], and other theoretical principles of genetic algorithms has no relevance in extremely small populations.

4.3. Mutation rate

The intuition behind mutations is that they increase the variability of population. Naturally, there is again a trade-off situation: if mutation rate becomes too large, the algorithm wanders aimlessly in the search space.

Figure 4 shows the results of our tests concerning mutation rate. The curve indicates the average difference in per cents between the final results of the genetic algorithm and the BC heuristic when mutation rate varies from 0.00 to 0.30. The average difference was calculated from the values obtained in 1000 runs for each mutation rate tested. The rates tested are shown on the x-axis. The other parameters were fixed as follows: graph density = 30%, graph size = 10, and population size = 20.

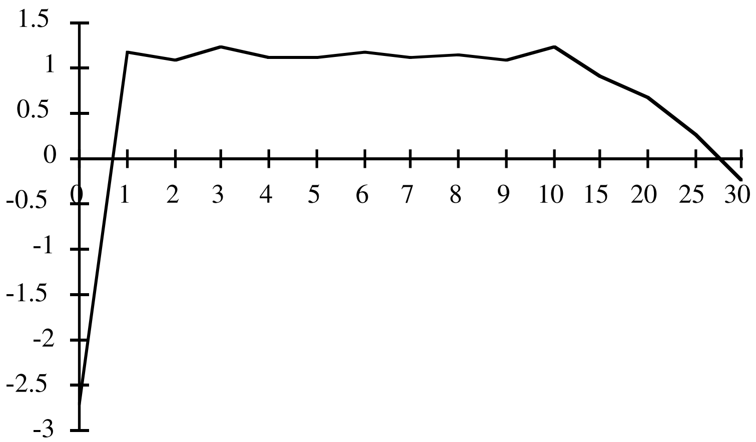

Figure 4. The difference between the results of the genetic algorithm and the BC heuristic in per cents when mutation rate varies from 0.00 to 0.30.

This test shows that mutation rates from 0.01 to 0.10 give approximately the same difference. Especially, they are better than the rate 0.00 which actually made the genetic algorithm to perform worse than the BC heuristic; i.e. mutations real help. Mutation rate should never exceed 0.10. If mutation rate is larger than 0.25, the BC heuristic again outperforms genetic algorithms.

We also recorded the number of iterations needed to find out the best result. The curve in Figure 5 shows that when mutation rate is 0.00 the algorithm converges very fast. Unfortunately, it ends up to a local minimum (as shown in Figure 4). The results with mutation rates from 0.01 to 0.10 were about the same in Figure 4. Figure 5 now shows that the number of iterations needed takes its minumum with mutation rate 0.03. Hence, we suggest mutation rate 0.03.

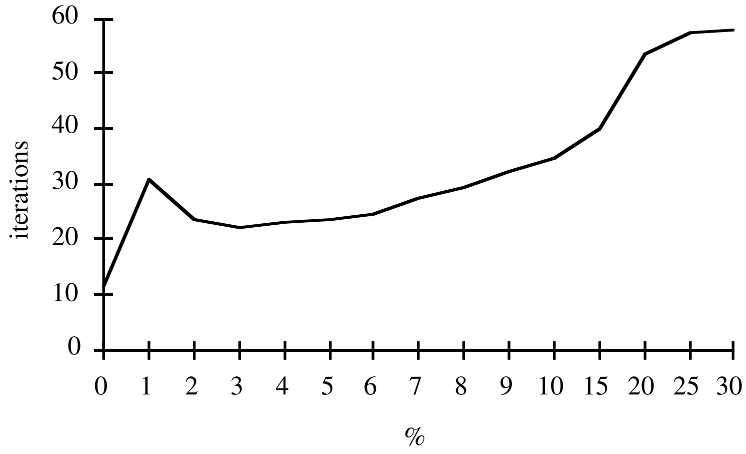

Figure 5. The number of iterations needed to find out the best result when mutation rate varies from 0.00 to 0.30.

4.4. Graph density

The BC heuristic and other known heuristics for LPP have the property of making larger relative error with low density graphs than with high density graphs [10]. We have also tested the effect of graph density to our genetic algorithms. We first counted the number of generations needed to reach the BC result. Figure 6 shows our results: more generations are needed with high density graphs than with low density graphs.

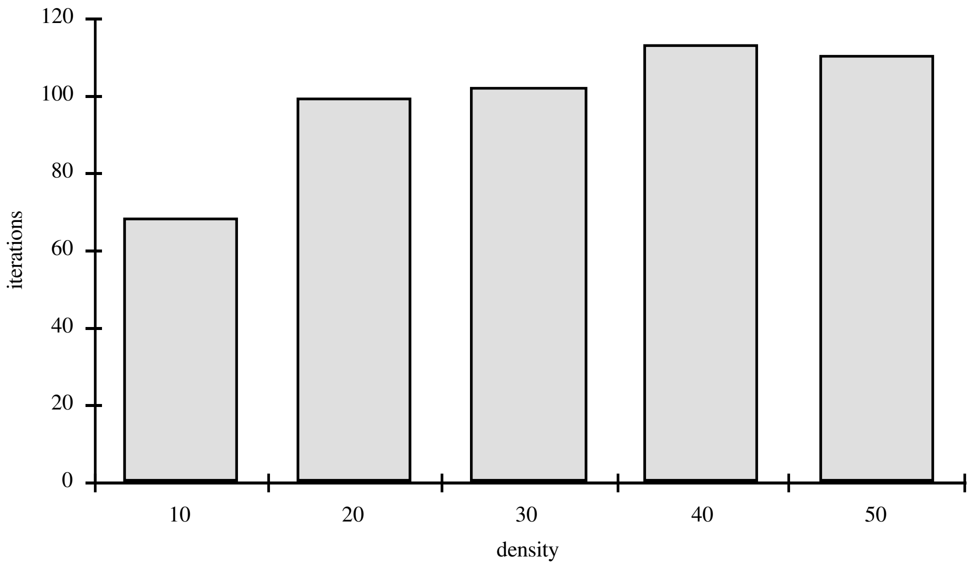

Figure 6. The number of iterations needed to reach the BC result when graph density varies from 10 % to 50 % (graph size = 15; mutation rate = 0.03).

Second, we tested the effect of graph density to the quality of the final results. As in the previous subsection, we have compared the best result obtained by genetic algorithms and the BC result. Figure 7 shows that genetic algorithms perform clearly better than the BC heuristic on low density graphs, but on high density graphs the difference is quite small. The average difference on graphs with density 10 % is more than 2 %, but on graphs with density 50 % the two methods produce almost the same result.

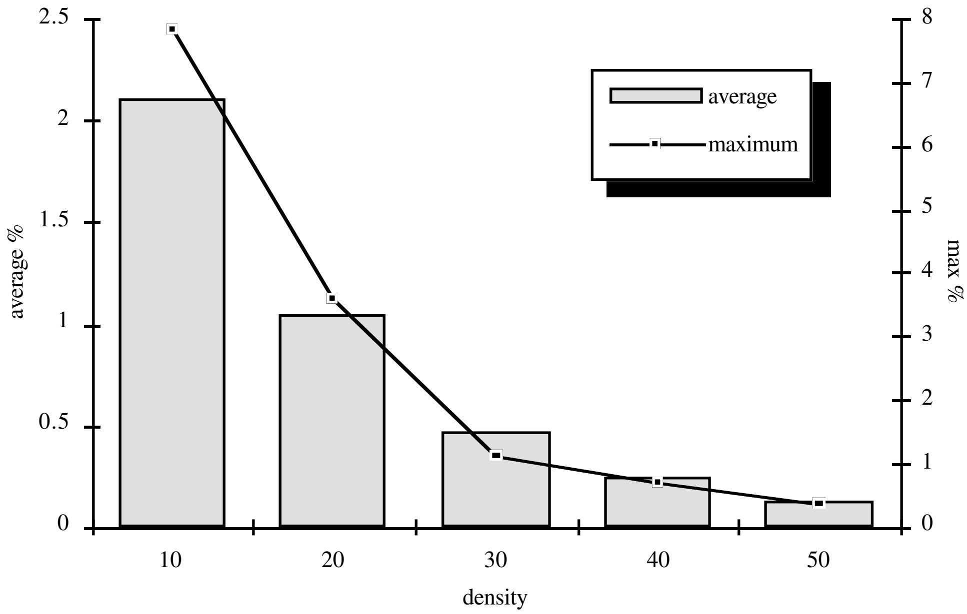

Figure 7. The average (the left scale) and maximum (the right scale) differences between the results on graphs with different density (graph size = 15; mutation rate = 0.03).

4.5. BC vs. genetic algorithms

Previous subsections have already compared the results of genetic algorithms and the results of the BC heuristic against each others. If we use the best parameters the genetic algorithm always performs at least as good as the BC heuristic. The genetic algorithm is clearly better than the BC heuristic especially when applied to low density graphs. The difference between the results is then about 2%. Note that the BC heuristic usually performs very favourably and that 2% usually means tens of edge crossings when graph size is 20 or more.

There is a natural reason to compare genetic algorithms against the BC heuristic and not against the optimum. Namely, the intractability of LPP makes it impossible to find the optimum when the graph size n becomes larger. For small values of n ($n \leq 9$), it is possible to solve all instances of LPP and compare the performances of the BC heuristic and our genetic algorithm against the optimum. Table I shows the portions of correct results (in per cents) produced by the two methods. Table I shows that our genetic algorithm always finds the optimum when n is small.

n	BC	GA
4	100	100
5	99	100
6	95	100
7	83	100
8	73	100
9	60	100

Table I. Correct performances in per cents of the BC heuristic and the genetic algorithm when n = 4,..., 9.

It is of interest to look closer the case n = 9. Table II shows (in per cents) the numbers of instances for which the BC heuristic makes different absolute errors.

Error	0	1	2	3	4	5	6	7	8
Instances	60	22	9	4	4	0	0	0	1

Table II. Per cents of instances with different absolute errors done by the BC heuristic when n = 9.

Table II indicates the general trend that the BC heuristic usually performs well. Hence, it is not reasonable to expect large relative improvements when using the genetic algorithm.

In order to compare the genetic algorithm and the BC heuristic with larger values of n, we randomly generated 100 bipartite graphs of size 20 and of density 30 %. In these tests the number of generations was at least 100, and new generations were created until there were 50 consecutive generations with no improvement in the result. The average difference between the genetic algorithm and the BC heuristic was about 0.5 % or 11 edge crossings. With graph size 30, the observed difference 0.3 % means almost 50 edge crossings. We believe that the result of our genetic algorithm is very close to the optimum when graph size is in the range [20-50]. Unfortunately, we cannot prove this claim because of the intractability of LPP.

When n increases the total number of individuals created during the generations increases, too. However, the total number of possible orders of vertices increases very much faster. The relative portion of possible permutations handled in the populations decreases dramatically. When n = 7, we have to create 2-3% of the search space in order to reach the BC results. On the other hand, when n = 15, the BC results is reached by handling about $0.2 \cdot 10^{-5}$ % of the search space, and when n = 20, only about $0.2 \cdot 10^{-11}$% of the search space have to be created.

5. Concluding remarks

We have found out that genetic algorithms can be used to sharpen the results obtained by previously known heuristics for LPP. We suggest the following parameters for genetic LPP algorithms: the elitist sampling mechanism described in subsection 4.1, mutation rate 0.03, and population size 50.

References

[1] D.Bienstock, Some provably hard crossing number problems. *Proc. 8th Annual ACM Symp. on Computational Geometry*, 1990, 253-260.

[2] G.Di Battista, P.Eades, R.Tamassia and I.G.Tollis, Algorithms for graph drawing: an annotated bibliography. Manuscript, June 1993.

[3] P.Eades and D.Kelly, Heuristics for drawing 2-layered networks. *Ars Combin.* **21-A** (1986), 89-98.

[4] P.Eades, B.McKay and N.Wormald, An NP-complete crossing number problem for bipartite graphs. Technical Report No. 60, Dept. of Computer Science, University of Queensland, St. Lucia, April 1985.

[5] P.Eades and N.Wormald, The median heuristic for drawing 2-layered networks. Technical Report No. 69, Dept. of Computer Science, University of Queensland, St. Lucia, May 1986.

[6] P.Eades and N.Wormald, Edge crossings in drawings of bipartite graphs. Tech. Report No. 108, Dept. of Computer Science, University of Queensland, St. Lucia, March 1989.

[7] M.R.Garey and D.S.Johnson, Crossing number is NP-complete. *SIAM J. Algebraic Discrete Methods* **4** (1983), 312-316.

[8] M.Koebe and J.Knöchel, On the block alignment proble. *J. Inf. Process. Cybern.* **EIK 26** (1990), 377-387.

[9] E.Mäkinen, A note on the median heuristic for drawing bipartite graphs. *Fund. Inform.* **XII**, 4 (1989), 563-569.

[10] E.Mäkinen, Experiments on drawing 2-level hierarchical graphs. *Intern. J. Comput. Math.* **36** (1990) 175-181.

[11] M.May and K.Szkatula, On the bipartite crossing number. *Control Cybernet.* **17**, 1 (1988), 85-98.

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

[13] K.Sugiyama, S.Tagawa and M.Toda, Methods for visual understanding of hierarchical system structures. *IEEE Trans. Syst. Man Cybern.* **SMC-11** (1981), 109-125.

[14] W.T.Tutte, Toward a theory of crossing numbers. *J. Combin. Theory* **8** (1970), 45-53.

[15] J.N.Warfield, Crossing theory and hierarchy mapping. *IEEE Trans. Syst. Man Cybern.* **SMC-7** (1977), 505-523.

[16] M.E.Watkins, A special crossing number for bipartite graphs: a research problem. *Ann. New York Acad. Sci.* **175** (1970), 405-410.