Evolutionary Learning of Graph Layout Constraints from Examples

Toshiyuki MASUI
Software Research Laboratories
SHARP Corporation
2613-1 Ichinomoto-cho
Tenri, Nara 632, Japan
Tel: +81-7436-5-0987
E-mail: masui@shpcsl.sharp.co.jp

ABSTRACT

We propose a new evolutionary method of extracting user preferences from examples shown to an automatic graph layout system. Using stochastic methods such as simulated annealing and genetic algorithms, automatic layout systems can find a good layout using an evaluation function which can calculate how good a given layout is. However, the evaluation function is usually not known beforehand, and it might vary from user to user. In our system, users show the system several pairs of good and bad layout examples, and the system infers the evaluation function from the examples using genetic programming technique. After the evaluation function evolves to reflect the preferences of the user, it is used as a general evaluation function for laying out graphs. The same technique can be used for a wide range of adaptive user interface systems.

KEYWORDS: Graphic Object Layout, Graph Layout, Genetic Algorithms, Genetic Programming, Programming by Example, Adaptive User Interface

INTRODUCTION

One of the goals of user interface research is to create nice-looking pictures of data structures. All the research on data visualization and text formatting fit into this category. TEX, the famous text formatting system, is one of these systems. It lays out characters, words, paragraphs, and figures in a two-dimensional space, treating them as boxes and evaluating the layout using the "badness" value. Like most other visualization systems, the layout scheme is coded deep in the system and users cannot change it easily even when they do not like it. Although users can change the behavior of TEX slightly by changing the badness value and other parameters, they cannot make great changes. In many other layout systems, users can do almost nothing except accepting the system output as-is. Complex layout systems are usually hard to build and hard to modify.

There are two reasons why layout systems are hard to build. First, when the data structure is complex, the algorithm to lay it out under some criteria also becomes complex and developing such an algorithm is usually difficult. Second, the criteria themselves are usually not obvious either to the developer or to the users. There usually exist many conflicting criteria which cannot be satisfied at the same time.

One solution to the first problem is using stochastic optimization techniques like simulated annealing[10] and genetic algorithms(GA)[6][8]. Using these techniques, if an evaluation function which can tell the goodness of a layout is known, near-optimal layout can be computed after iterations of searching in the solution space. These techniques are widely used in VLSI layout systems, where the evaluation function (usually the size of the chip) is clear and time constraints are not severe [2][22][23].

To solve the second problem, by-example approaches[3] are promising. If the system can guess a user's intentions or preferences from the examples given by the user, users don't have to specify preferences directly to the system. Many researches have been working in this area. Myers[19] introduced a WYSIWYG editor which can create text formatting macros from user examples. With his system, for example, users can make the system infer the formatting macros for section titles just by drawing one sample section title. Hudson[9] showed a graphic layout system which can generalize the layout rules from a small number of examples. Since it is difficult for the system to infer proper layout rules from a small number of examples, the system generates many possible rules and shows the user how they work, and the user selects the right one from them. Miyashita's IMAGE system[18] can infer more complicated rules from a small number of examples, also interacting with the user. Although these systems can infer simple layout rules from examples, they consist of many heuristics and cannot be used for more complex layout tasks.

We propose taking a completely different approach of using evolutionary learning technique for constructing the layout evaluation function from examples. In our system, users show the system several pairs of good and bad layout ex-

Published in:
Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'94) (November 1994), ACM Press, pp. 103–108.

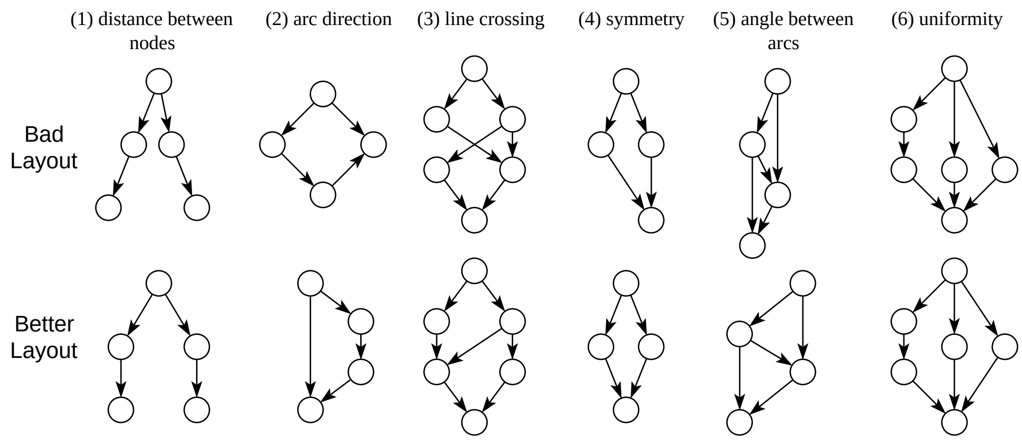


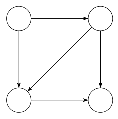
Figure 1: Constraints used in the layout of directed graphs.

amples, and the system infers the evaluation function using *genetic programming* technique[12], where a population of tree-structured evaluation functions "evolve" to a function which reflects the user's preferences, under many generations of Darwinian selection pressure. Once such an evaluation function is obtained, it is used as the user's own preference function to be used with stochastic layout systems such as [16]. In this paper, we show how this technique works, using the directed graph layout problem as an example.

DIRECTED GRAPH LAYOUT PROBLEM

Directed Graphs

A *directed graph* is a graph which consists of a set of nodes and a set of arcs. An arc is an ordered pair of nodes (n, m), where n is called the *tail* and m is called the *head*. Below is a directed graph with four nodes and five arcs.



When a directed graph has many nodes and arcs, it becomes very hard to lay out all of them so that the graph looks nice to humans. Many constraints can be defined to make the layout look nice. Some of them are listed in Table 1 and Figure 1.

1. There should be enough space between nodes.
2. The head of an arc should be below the tail.
3. Arc crossings should be avoided.
4. There should be as much symmetry as possible.
5. The angle between two arcs should not be too small.
6. Nodes should be placed uniformly in the region.

Table 1: Constraints for laying out a directed graph.

Algorithms for Directed Graph Layout

Many graphic layout algorithms for directed graphs have been proposed[24]. However, finding a layout which gives a minimal number of line crossing is an NP-hard problem, and many other problems are, too. So, most of the algorithms for laying out directed graphs use heuristics. For example, Eades and Sugiyama[5] introduced the following algorithm.

Step 1 Sort all the nodes according to the arc directions between nodes.
Step 2 Calculate the minimal number of "layers" from the top of the graph to the bottom.
Step 3 Assign every node to one of the layers so that there is no arc from a node in a layer to another node in the same layer. Nodes should be scattered uniformly.
Step 4 If there is an arc between nonadjacent layers, add dummy nodes between the head and the tail of the arc, and put them onto layers between them.
Step 5 Arrange nodes in each layer so that there are as few line crossings as possible.

Here, finding the best solution in step 2, 3, and 5 is NP-hard, and several heuristic methods are used in their algorithm.

In this kind of algorithmic solution to graph layout problems, the evaluation of the layout is coded implicitly in the algorithm and cannot easily be changed without totally modifying the algorithm.

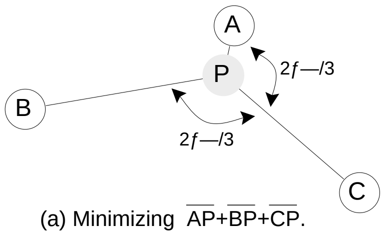
Using Stochastic Methods for Graph Layout Problems

Using genetic algorithms for the layout of graphs is becoming popular [7] [11] [16] [17] [20]. In these systems, the evaluation function for the layout is given explicitly, and the system designer can modify it fairly easily. However, getting an appropriate evaluation function is not an easy task. For example, in [16], the formula in Figure 2 is used as the evaluation function of the layout. (A small value reflects a good layout here.)

3000 * (the number of arrows going upward) +
400 * (the number of arcs shorter than a constant C_1) +
300 * (the number of arc crossings) +
400 * (the number of angles between arcs which are smaller than a certain constant C_2)

Figure 2: The layout evaluation function used in [16].

This formula contains many magic numbers. A number of trial and error loops have been performed before getting these values. Changing the values and functions slightly will produce totally different results that are quite unpredictable. We show this by a simple example. If you want to put a node P at some place within a triangle ABC and use $\overline{AP} + \overline{BP} + \overline{CP}$ as the evaluation function to minimize, P should be at a point where $\angle APB = \angle BPC = \angle CPA = 2\pi/3$. If you use $\overline{AP}^2 + \overline{BP}^2 + \overline{CP}^2$ instead, P should be at the gravity center of ABC. (See Figure 3.) In this way, you cannot easily guess what kind of layout is produced from a given evaluation function.



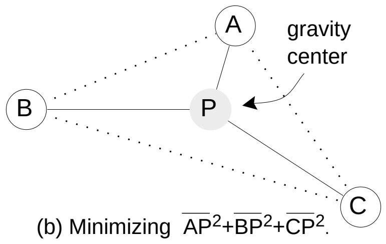

A
gravity center
B
P
C
(b) Minimizing $\overline{AP}^2+\overline{BP}^2+\overline{CP}^2$.

Figure 3: Two evaluation functions and resulting layout.

In [16], undesirable layouts can be modified interactively by the user and the inappropriateness of the evaluation function is not a big problem, but in other systems, users can do nothing but accept the resulting layout. In any case, if users can show their preference somehow and specify the evaluation function to the system, stochastic methods become much more appealing.

DEVELOPING THE LAYOUT EVALUATION FUNCTION THROUGH GENETIC PROGRAMMING

Genetic Programming

Genetic programming[12] is a technique to make randomly-generated programs “evolve” to a program which conforms to the specification given by the user, just like performing optimization in genetic algorithms. Programs are usually represented as trees, like the S-expressions of Lisp. The algorithm starts with many randomly-generated tree-shaped programs. First, all the programs are checked for differences from the given specification. If a program works closer to the specification, it will have a better chance of surviving to the next iteration, or generation, and if it performs badly, it will not survive to the next generation. After evaluating all the programs and selecting what will survive to the next generation, some pairs of the programs exchange their subtree (called the *crossover* operation) like shown in Figure 4, so that even better program can be generated. Also, some of

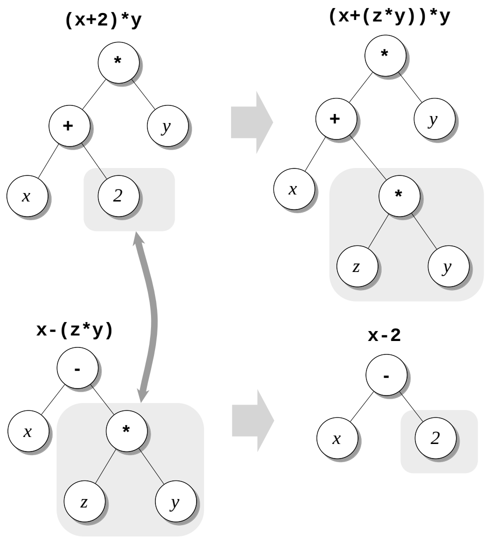


Figure 4: Crossover operation.

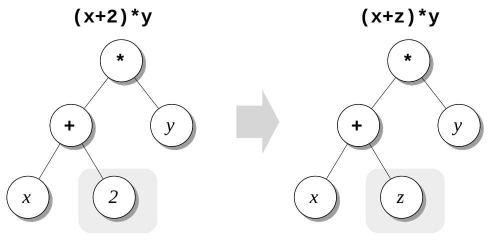


Figure 5: Mutation operation.

the programs substitute their subtree with other randomly-generated program tree (called the *mutation* operation) like shown in Figure 5. After many generations of these genetic operations, programs which works close to the specification will eventually emerge.

Getting User Preferences from Examples Using Genetic Programming

Using genetic programming technique, we can generate the layout evaluation function only from the examples given by the user. We give the genetic programming system pairs of good and bad layout examples. If a program yields a larger value for a good layout and a smaller value for a bad layout, there is a chance that the program can tell how good a layout is. If it works the same way with many example pairs, the chance becomes even greater. We use N graphs as examples and for each graph $i \in 1..N$, provide good layout G_i and bad layout B_i. To make an evaluation function f evolve, we use $E(f) = \sum_{i=1}^{N} p(f,i)$ as the evaluation function for f, where $p(f,i) = 1$ if $f(G_i) > f(B_i)$, and $p(f,i) = 0$, otherwise. If f yields a larger value for good layouts than bad layouts for all the given examples, $E(f)$ takes the value N. With many examples, chances are we can get a good evaluation function which truly reflects the user’s preferences.

EMPIRICAL RESULTS

We used only a small number of operators, arguments and constants to construct the evaluation function, although using control constructs like condition statements is also possible.

The arguments and operators used in the evaluation functions are listed in Figure 6.

Operators

add, sub, mul, div, abs,
sum, max, min, compare

Arguments

x/y coordinates of node locations
x/y directions of arcs
number of crossing arcs
node distances
minimal angle between arcs connected to a node

Figure 6: Operators and arguments used in layout evaluation functions.

At the beginning of the algorithm, evaluation functions are generated randomly using these operators, arguments and constants. We provided 22 ($= N$) example layout pairs, some of which are shown in Figure 7. For each evaluation function in the population, we computed $f(G_i)$ and $f(B_i)$ for each example i and computed $E(f)$ to be used for the evolution.

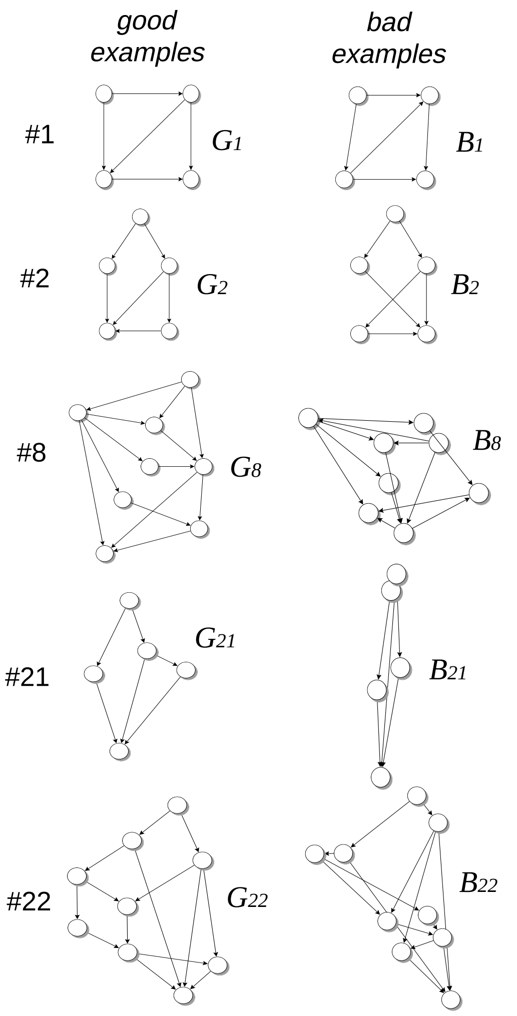


Figure 7: Good and bad example layout pairs given to the system.

The evolution of the best and average performances of the evaluation functions are shown in Figure 8. The population, crossover ratio and mutation ratio used in this example is 600, 0.8, 0.005, respectively. Using appropriate values for crossover ratio and mutation ratio is important. In this example, this combination yield good results.

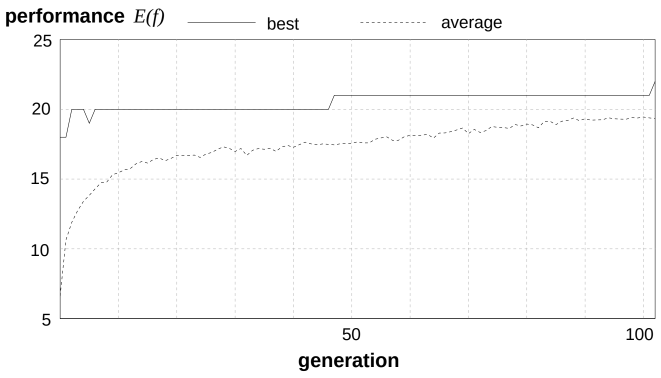


Figure 8: Best and average performance $E(f)$.

At generation 1 with randomly-generated evaluation functions, the average performance is as low as 7 and the best performance is 18. As the computation goes on, evaluation functions evolve, and at generation 102, an evaluation function f_b which calculates larger values for good layout examples for all the input examples ($f(G_i) > f(B_i)$ for all $i \in 1..22$) is found. f_b, represented like a Lisp program, is shown in Figure 9.

```
(ADD (SUB (ADD (MUL (MUL (MUL (ADD (ADD (ADD SUM(diry)
SUM(minangle)) (ADD 44.00 69.00)) (MUL 43.00 MIN(diry)))
5.00) (ADD (ABS MAX(minangle) MIN(dist)) (ADD (ABS 74.00
MIN(dirx)) (ABS 15.00 SUM(locx))))) SUM(minangle)) (MUL 12.00
(CMP (DIV 57.00 MIN(locx)) (CMP 94.00 MIN(intersec))))) (DIV
(ABS (MUL (SUB SUM(locy) 27.00) (CMP 28.00 65.00)) 62.00)
SUM(dirx))) (CMP (ABS (DIV 67.00 SUM(locy)) (CMP (ABS
(ABS 73.00 (CMP 67.00 SUM(dist))) MIN(intersec)) (ABS MIN(dist)
MIN(diry)))) MIN(diry)))
```

Figure 9: Computed layout evaluation function f_b.

Using the function f_b as the evaluation function of our GA-based automatic graph layout system[16], we could get the layouts shown in Figure 10. Although only the number of nodes and link information are given to the layout system, it automatically produces nice layouts just using f_b as the evaluation function. This means that although f_b looks quite complex, it reflects the preferences of the user shown implicitly in the examples. For example, most of the arcs in Figure 10 are going downward, reflecting the fact that most arcs in the good examples in Figure 7 are going downward.

DISCUSSIONS

The Path to Really Adaptive User Interface

As the user provides more examples, the performance of our system improves. This means that for creating adaptive user interfaces [1] [21], evolutionary approaches seem quite promising. Adaptive user interface systems have been

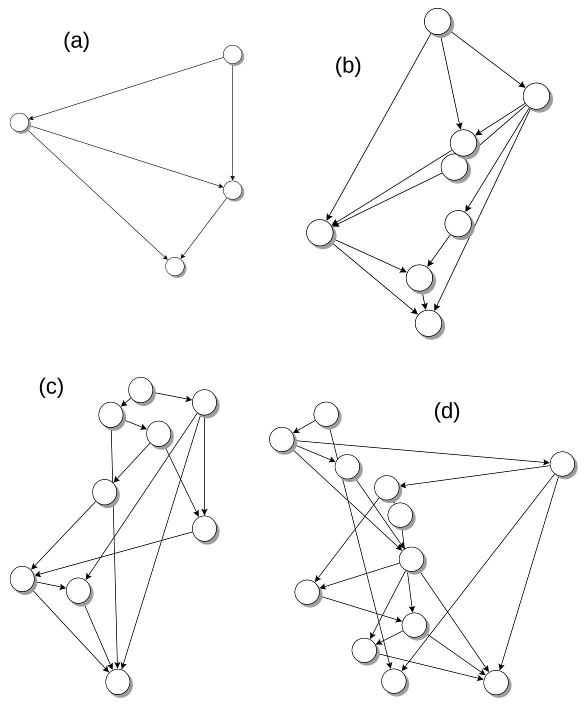


Figure 10: Layout results using f_b as the evaluation function.

considered to be difficult to construct, and most of the systems proposed so far were based on rigid application knowledge with a small amout of adaptivity, controlled by the user models which are also rigid. However, recent research in machine learning and artificial life[14] suggests that various techniques can be applied to make interface systems learn or evolve to fit to each user's needs. For example, Maes proposes "learning interface agents" [13] [15] which gradually learn to assist the individual user by observing the user's actions, getting user feedback, and explicit training, based on memory-based learning approach. If a system can not only learn, but evolve to a more powerful learning system, it can be a very powerful framework for adaptive interfaces.

Is Genetic Programming Really Effective?

Some people doubt if genetic programming is nothing but a random search. In our example shown in the last section, we reached the desired evaluation function after 102 generations with a population size of 600. This means we found the evaluation function after about 60,000 trials. On the other hand, if we generate the evaluation function totally randomly until we find a function f where $E(f) = 22$, we have to perform $2^{22} \simeq 4{,}000{,}000$ trials on the average before finding it. So at least for this layout problem, genetic programming technique is actually working much better than simple random searches.

Interactive Technique for Showing Examples

Although showing examples is easier than writing an entire layout program or layout rules, providing many examples is still a heavy burden to users. Some by-example systems including Hudson's system[9] and IMAGE[18] try to reduce the number of examples to be shown to the system by interactively telling the system the right direction of the inference. Similar ideas can also be applied to our system. After a layout evaluation function is computed from a small number of examples, we can use it for the layout of another example, check the result, and if the result is not good enough, use it as another bad layout example and compute the evaluation function again with the accompanying better layout example. In fact, B_{21} in Figure 7 is the output of an automatic layout system using the evaluation function computed by examples #1 to #20, and the B_{22} is created from examples #1 to #21.

Another possibility is to make the system create examples automatically like Hudson's system[9], instead of receiving examples from the users. For example, if the user iteratively selects the best layout from the examples generated by the system, the system might eventually learn this user's preferences. This is a well-known technique in creating artistic pictures and virtual insects[1] using evolutionary techniques. We did not take this approach because selecting the best layout from many random-looking layouts seems much more difficult than comparing only two layouts and deciding which is better.

Preventing Over-Adaptation and Complexity

The f_b shown in Figure 9 is quite complex and involves many meaningless elements. This is because we did not take into account the simpleness of the evaluation function when making them evolve. This problem can be solved by adding the simpleness factor into $E(f)$, although it takes a longer time to get a simpler evaluation function which does the same thing as a complex one.

Processing Speed

The biggest problem with our system is that it takes a very long time (several minutes in the example above) to get a layout evaluation function in the current implementation using Objective-C running on a 68040 NeXTstation. However, just like constraint solvers are widely used today while they were too slow to be used for interaction ten years ago, we believe that evolutionary techniques will become one of the main techniques for adaptive interfaces, since they are so simple, robust and useful.

CONCLUSIONS

Although we are still in the process of trying and evaluating various evolutionary techniques including genetic programming, we believe that it is a promising approach to adaptive user interfaces and other by-example systems. As the next step, we are preparing to apply this technique to other application areas such as information retrieval systems.

REFERENCES

1. Browne, D., Totterdell, P., and Norman, M., Eds. *Adaptive User Interfaces*. Academic Press, London, 1990.

2. Cohoon, J. P., and Paris, W. D. Genetic placement. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1986), pp. 422–425.

[1]The BioMorph system introduced in [4] is an evolutionary shape-breeding system under the user's selection pressure. It shows the user nine variations of insect-like object generated from an encoded string, and the user can iteratively select one of the variations to make the string evolve and produce his favorite shape.

3. Cypher, A., Ed. *Watch What I Do – Programming by Demonstration*. The MIT Press, Cambridge, MA 02142, 1993.

4. Dawkins, R. *The Blind Watchmaker*. W. W. Norton & Company, 1985.

5. Eades, P., and Sugiyama, K. How to draw a directed graph. *Journal of Information Processing 13*, 4 (1990), 424–437.

6. Goldberg, D. E. *Genetic Algorithm in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

7. Groves, L. J., Michalewicz, Z., Elia, P. V., and Janikow, C. Z. Genetic algorithms for drawing directed graphs. In *Methodologies for Intelligent Systems 5, Proceedings of the Fifth International Symposium* (October 1990), Z. W. Ras, M. Zemankova, and M. L. Emrich, Eds., North-Holland, pp. 268–276.

8. Holland, J. H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

9. Hudson, S. E., and Hsi, C.-N. A synergistic approach to specifying simple number independent layouts by example. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems (CHI'93)* (April 1993), Addison-Wesley, pp. 285–292.

10. Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. *Science*, 220 (1983), 671–680.

11. Kosak, C., Marks, J., and Shieber, S. A parallel genetic algorithm for network-diagram layout. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (UCSD, California, August 1991), Morgan Kaufmann Publishers, pp. 458–465.

12. Koza, J. R. *Genetic Programming*. The MIT Press, Cambridge, MA, 1992.

13. Kozierok, R., and Maes, P. A learning interface agent for scheduling meetings. In *Proceedings of the 1993 International Workshop on Intelligent User Interfaces* (January 1993), ACM Press, pp. 81–88.

14. Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S., Eds. *Artificial Life II*, vol. X of *Santa Fe Institute Studies in the Sciences of Complexity*. Addison-Wesley, 1991.

15. Maes, P. Learning interface agents. In *Proceedings of the 1994 Friend21 International Symposium on Next Generation Human Interface* (February 1994).

16. Masui, T. Graphic object layout with interactive genetic algorithms. In *Proceedings of the 1992 IEEE Workshop on Visual Languages* (September 1992), IEEE Computer Society Press, pp. 74–80.

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

18. Miyashita, K., Matsuoka, S., Takahashi, S., and Yonezawa, A. Interactive generation of graphical user interfaces by multiple visual examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '94)* (November 1994), ACM Press.

19. Myers, B. A. Text formatting by demonstration. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'91)* (April 1991), Addison-Wesley, pp. 251–256.

20. Pazel, D. P. A graphical interface for evaluating a genetic algorithm for graph layout. Tech. Rep. RC14348, IBM Research Division, T.J. Watson Research Center, 1989.

21. Schneider-Hufschmidt, M., Ed. *Adaptive User Interface*. North-Holland, Amsterdam, 1993.

22. Shahookar, K., and Mazumder, P. A genetic approach to standard cell placement using meta-genetic parameter optimization. *IEEE Transaction on Computer-Aided Design 9*, 5 (May 1990), 500–511.

23. Shahookar, K., and Mazumder, P. VLSI cell placement techniques. *ACM Computing Surveys 23*, 2 (June 1991), 143–220.

24. Tamassia, R., and Eades, P. Algorithms for drawing graphs : an annotated bibliography. Tech. Rep. CS-89-09, Brown University, Department of Computer Science, October 1989.