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.

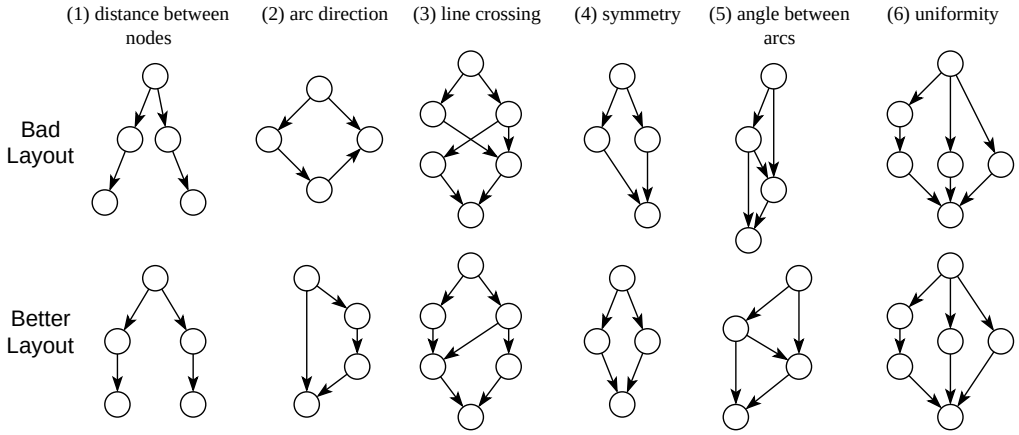


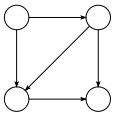
Figure 1: Constraints used in the layout of directed graphs.

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.

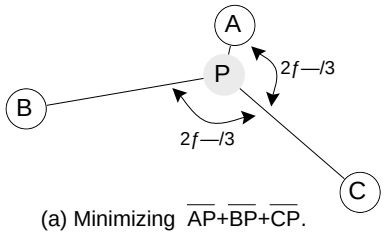
Using Stochastic Methods for Graph Layout Problems

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.)

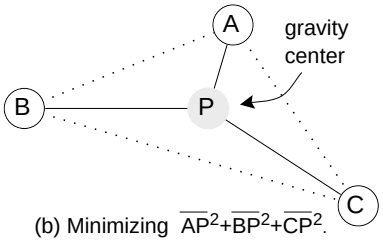
3000 * (the number of arrows going upward) +
400 * (the number of arcs shorter than a constant C_1) +
300 * (the number of arc crossings) +
400 * (the number of angles between arcs which are smaller than a certain constant C_2)

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) Minimizing $\overline{AP}+\overline{BP}+\overline{CP}$.



(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

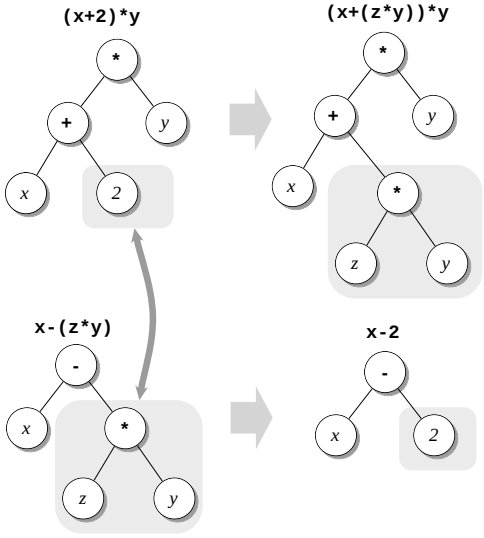


Figure 4: Crossover operation.

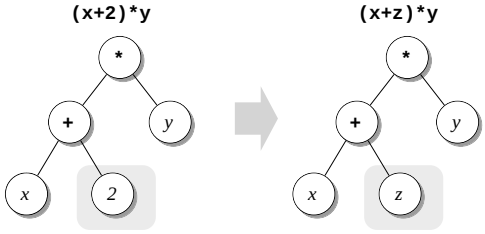


Figure 5: Mutation operation.

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.