A GENETIC PROGRAMMING APPROACH TO THE SPACE LAYOUT PLANNING PROBLEM

ROMUALD JAGIELSKI [*] AND JOHN S. GERO
Key Centre of Design Computing
Department of Architectural and Design Science
University of Sydney NSW 2006 Australia
email: {rom, john}@arch.usyd.edu.au

Abstract.
The space layout planning problem belongs to the class of NP-hard problems with a wide range of practical applications. Many algorithms have been developed in the past, however recently evolutionary techniques have emerged as an alternative approach to their solution. In this paper, a genetic programming approach, one variation of evolutionary computation, is discussed. A representation of the space layout planning problem suitable for genetic programming is presented along with some implementation details and results.

1. Introduction

Algorithmic solutions of spatial allocation problems were first developed more than thirty years ago. In this paper we discuss space layout planning formulated as a quadratic assignment problem. This is a difficult combinatorial optimisation problem of great importance, reaching beyond formal architectural design. Lately, such spatial allocation problems have again attracted attention, yielding solutions which employed genetic algorithms (Jo and Gero 1997). An overview and history of the automated layout problem of the pre-genetic period can be found in Liggett (1985).

The quadratic assignment problem can be defined in the following way: m distinct objects M = {1,2,...,m} are to be placed uniquely in n distinct sites N = {1,2,...,n}, where $m \leq n$. In other words, we want to find a one-to-one mapping of the set of facilities M into a set of locations N:

$$\rho : \{ M \rightarrow N \}, \qquad j = \rho(i), \ i \quad M, \ j \quad N$$

The quadratic assignment problem (Tate 1995) can be stated as the task of finding the minimal total cost Cost(A) of allocation A, usually with some constraints to be satisfied as well:

---

[*] on leave from Swinburne University of Technology, School of Computer Science and Software Engineering, *RJagielski@swin.edu.au*

$$\text{Cost}(A) = f_{i\,\rho(i)} + \sum_i \sum_j q_{ij}\; c_{i\,\rho(i)\,\rho(j)} \qquad (1)$$

where

$f_{i\,\rho(i)}$ fixed cost of assigning element i M to element j N,
q_{ij} intensity of interaction (traffic) between elements i, j M,
$c_{i\,\rho(i)\,\rho(j)}$ cost of interaction between elements i, j N, (often referred as the distance between i and j).

2. Introduction to Genetic Programming

Genetic (or evolutionary) computing is a powerful method of programming which relies on developing systems that demonstrate self-organization and adaptation in a similar, though simplified, manner to the way in which biological systems work. The best known technique are genetic algorithms(Goldberg 1989, Koza 1992, Michalewicz 1994). Koza has proposed a system which evolves Lisp computer programs. This method, *called genetic programming*, is extensively described in his book (Koza 1992). Genetic programming starts with an initial population of hundreds or thousands of randomly generated computer programs. Then using the Darwinian principle of survival and reproduction of the fittest, new offspring populations are created. The reproduction operation involves selecting from the current population programs (individuals) in proportion to their fitness and copying them to the next population. The best individuals survive, and finally optimal or near optimal programs are generated. Following is the pseudo-code for genetic programming:

Create an initial population of randomly generated programs
REPEAT
Execute each program in the population and evaluate its fitness
Create a new population of programs by
- reproduction
- crossover and/or
- mutation
UNTIL the termination condition is satisfied.

The solution is the best program that appeared in any generation (called the *best so far*). The individuals of the populations originally were so called S-expressions (or Lisp-expressions), but it is possible to develop genetic programming in any other language, with individuals in the form of syntax trees. The program used in this investigation, called GENPRO, has been written in C++ and uses C-like syntax trees to represent programs (individuals) and finally those programs constitute the outcome of genetic programming.

The initial population consists of programs which are randomly generated syntax trees. These programs are produced from a set of non-terminals (or functions):

$\mathbf{F} = \{f_1, f_2, \ldots, f_n\}$

and a set of terminals

$\mathbf{T} = \{t_1, t_2, \ldots, t_m\}$

Each terminal and nonterminal can be represented by a node on the syntax tree. If we think about them as functions, then terminals have arity zero (they do not have arguments, and in the tree they are always leaves). In a program, terminals are variables or constants. Non-terminals have one or more arguments, so in the tree they are always nodes with children. includes arithmetic and Boolean operators, mathematical functions, conditional and iterative operators and any other primitive functions predefined by the user.

The initial population and subsequent operations involve random generation of symbols which, sooner or later, create a tree not having any conventional meaning or containing an illegal operation. Therefore the system must have a closure property. The closure property ensures that any combination of symbols and data types is always legal. That is why many of the operators have to be redefined. For example, the normal division will be replaced by a special division, which allows division by zero. The initial population of programs is converted to a new population by the genetic operators of reproduction, crossover and mutation. Reproduction is an asexual operation, ie, it requires only a single individual. For this operation a tree is selected from the current population according to a criterion based on fitness, and is copied into the new population. Typically fitness-proportionate reproduction is used (the biased roulette wheel). Recently the tournament selection has proven to be popular (a small group of individuals is selected randomly and then the fittest individual in the group is determined and reproduced). Crossover is a sexual operation and requires two individuals. Two programs (parents) are chosen with the probability proportional to their fitnesses. A crossover point at each tree is selected randomly, and the fragments of the trees which follow the crossover points are swapped. Mutation is a small random change in a program. For example one node of a tree can be replaced by a randomly generated different one. Usually mutation is used sparingly.

The process of transforming the current population by means of reproduction, crossover and mutation is repeated. As a termination condition, the number of generations can be chosen and the best program that appeared in any generation *(the best so far* individual) is the solution. Alternatively the program may be terminated when the fitness reaches a certain level (for example a sufficiently small error). These automatically produced programs are incomprehensible to humans, but are legitimate C functions and can be included without any modification to an ANSI C or C++ program.

3. An Office Layout Problem

We will apply the genetic programming approach the problem of an office layout planning (Liggett 1985). There are a number of office departments to be placed in a four-level terraced building. The building is divided into 17 zones, as it is shown in Figure 1 (letters A, B, C, ... denote zones). The whole area of the building is divided into smaller units (modules - each letter in Figure 1 represent one module), thus the size of each zone can be expressed as a number of modules (Table 1). Some zones have a fixed allocation

(here zone 1- executive office, and zones 18 and 19 - the public access). We have in total 280 modules to which 17 departments are to be allocated. The departments require 277 modules in total. For practical reasons we will add three "dummy" departments of one module size each, and in this way the total required area equals the total available area (that is 280 modules). Now, having a fixed order of assigning (as in Figure 2), our space layout planning problem has been reduced to finding a sequence of departments d_1, d_2,...,d_{20}, which are allocated correspondingly to zones in that order. Since we don't consider any constraints here (for example, that each department should be located on one floor only), each permutation of 20 departments is a feasible solution. Our task, however, is to find such a sequence of allocations, which minimises the costs defined by formula (1). In this problem the fixed cost $f_{i\,\rho(i)}$ is ignored, the cost of interaction (travel) $c_{i\,\rho(i)\,\rho(j)}$ is specified in Table 1 and intensity of interaction between departments q_{ij} is given in Table 2. Since the departments can be split between different zones, the calculation of the total cost has to be performed as a sum of costs of all 280 modules.

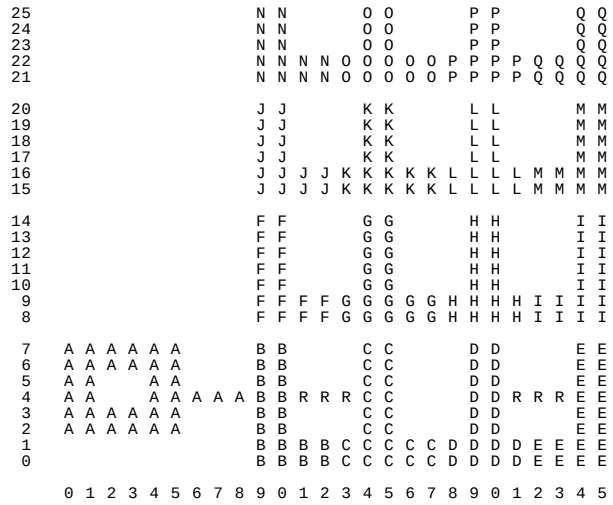

Figure 1. Graphical representation of zone definition.

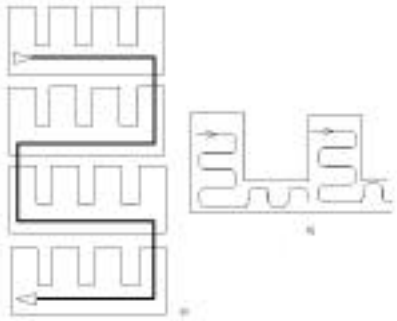

Figure 2. Order of the departments assignment (a) over floors of the building, (b) over a floor.

0 =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1 =	0	0	4	9	21	27	9	10	23	28	10	11	24	29	11	12	25	30	6	24
2 =	0	4	0	5	17	23	5	6	19	24	6	7	20	25	7	8	21	26	2	20
3 =	0	9	5	0	13	18	6	5	18	23	7	6	19	24	8	7	20	25	2	15
4 =	0	21	17	13	0	5	23	18	5	6	24	19	6	7	25	20	7	8	15	2
5 =	0	27	23	18	5	0	24	19	6	5	25	20	7	6	26	21	8	7	20	2
6 =	0	9	5	6	23	24	0	5	18	23	5	6	19	24	6	7	20	25	3	21
7 =	0	10	6	5	18	19	5	0	13	18	6	5	18	23	7	6	19	24	3	16
8 =	0	23	19	18	5	6	18	13	0	9	23	18	5	6	24	19	6	7	16	3
9 =	0	28	24	23	6	5	23	18	9	0	24	19	6	5	25	20	7	6	21	3
10 =	0	10	6	7	24	25	5	6	23	24	0	5	18	23	5	6	19	24	4	22
11 =	0	11	7	6	19	20	6	5	18	19	5	0	13	18	6	5	18	23	4	17
12 =	0	24	20	19	6	7	19	18	5	6	18	13	0	5	23	18	5	6	22	4
13 =	0	29	25	24	7	4	24	23	6	5	23	18	5	0	24	19	6	5	17	4
14 =	0	11	7	8	25	26	6	7	24	25	5	6	23	24	0	5	18	23	5	23
15 =	0	12	8	7	20	21	7	6	19	20	6	5	18	19	5	0	13	18	5	18
16 =	0	25	21	20	7	8	20	19	6	7	19	18	5	6	18	13	0	5	18	5
17 =	0	30	26	25	8	7	25	24	7	6	24	23	6	5	23	18	5	0	23	5
18 =	0	6	2	2	15	20	3	3	16	21	4	4	22	27	5	5	18	23	0	18
19 =	0	24	20	15	2	2	21	16	3	3	22	17	4	4	23	18	5	5	18	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Table 1. Cost of interaction (matrix D).

0 -	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1 -	0	0	10	10	5	5	5	0	0	10	0	0	0	0	0	0	5	0	0	0	10
2 -	0	10	0	10	5	5	5	0	0	10	0	0	0	0	0	0	5	0	0	0	0
3 -	0	10	10	0	10	5	5	0	0	10	0	0	0	0	0	0	5	0	0	0	0
4 -	0	5	5	10	0	10	5	0	0	10	0	0	0	0	0	0	5	0	0	0	0
5 -	0	5	5	5	10	0	5	0	0	10	0	0	0	0	0	0	5	0	0	0	0
6 -	0	5	5	5	5	5	0	0	0	10	0	0	0	0	0	0	5	0	0	0	10
7 -	0	0	0	0	0	0	0	0	10	0	0	0	0	0	10	0	5	0	0	0	0
8 -	0	0	0	0	0	0	0	10	0	0	0	0	0	0	10	0	5	0	0	0	0
9 -	0	10	10	10	10	10	10	0	0	0	0	0	0	0	0	0	5	0	0	0	0
10 -	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	10
11 -	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	5	0	0	0	0
12 -	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	5	0	0	0	0
13 -	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	5	0	0	0	0
14 -	0	0	0	0	0	0	0	10	10	0	0	0	0	0	0	0	5	0	0	0	10
15 -	0	0	0	0	0	0	0	0	0	0	0	10	10	10	0	0	5	0	0	0	0
16 -	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	0	0	0	0	0	10
17 -	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18 -	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19 -	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20 -	0	10	0	0	0	0	10	0	0	0	10	0	0	0	10	0	10	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Table 2. Intensity of interaction between departments (matrix I).

List of zones and their sizes(in modules)

no	symbol	size	no	symbol	size	no	symbol	size
1	A	39	7	G	20	13	M	16
2	B	20	8	H	18	14	N	14
3	C	22	9	I	18	15	O	16
4	D	20	10	J	16	16	P	14
5	E	20	11	K	18	17	Q	14
6	F	18	12	L	16	18	R	6

List of departments and their sizes(in modules)

no	symbol	size	no	symbol	size	no	symbol	size
1	a	2	8	h	7	15	o	31
2	b	2	9	i	6	16	p	61
3	c	8	10	j	12	17	q	1
4	d	8	11	k	53	18	r	1
5	e	15	12	l	10	19	s	1
6	f	13	13	m	16	20	t	6
7	g	15	14	n	18			

Table 3. List of zones and departments of the office layout planning problem

4. Genetic Programming Solution to the Office Layout Problem

A solution to the space layout planning problem is a sequence of numbers (or a vector), representing allocations of the particular facilities, and in this form is very convenient for to genetic algorithms. A string of bits or numbers is a natural representation for genetic algorithms but is not the case for genetic programming. Genetic programming produces solutions in a form of programs, or more precisely, in a form of syntax trees. Therefore, applying genetic programming to the problem of space layout planning, we assume that the expected solution is a sequence of terminals read in order from a syntax tree during the evaluation process. The following code is a sample solution produced by genetic programming (an exact output from GENPRO):

```
GPvalue =
( IFG( (Act((d),((j))), (i), ( Dis((k), ( IFG( ( IFG( (e), ( Dis((e), ( Act((j)
, (q)) )) ), (i)) ), (q), (h)) )) )) )
; // end of the GPvalue
```

An equivalent tree is shown in Figure 3. In this case we used the set of terminals: **T** = {a, b,...,s}, and the set of functions: **F** = { *IFG, Act, Dis* }. The terminals denote the departments (as specified in Table 3).

The functions *IFG, Dis, Act* have been chosen arbitrarily, with the underlaying aim to build parse trees rich in information. *IFG* is an *if-greater-then* function, which implements conditional branching on the trees. Functions *Dis* and *Act* are in fact Boolean

functions, and are used as arguments for *IFG*. The meaning of these functions is given by the following pseudo-code:

```
        IFG(z, y, z)        // if-then-else function
Evaluate z (z can be a terminal or a function IFG, Dis or Act)
Allocate (z)
IF z > 0
THEN
        Allocate( x)
        Return x
ELSE
        Allocate( y)
        Return y

        Act(x, y):          // data-dependent function (intensity of interaction in matrix I)
IF I(x, y) > 0
THEN
        Allocate( x )
        Allocate( y)
        Return x
ELSE
        Allocate( y)
        Allocate( x)
        Return 0

        Dis(x, y):          // data-dependent function (distance in matrix D)
IF D(x, y) > 0
THEN
        Allocate( x)
        Allocate( y)
        Return x
ELSE
        Allocate( y)
        Allocate( x)
        Return 0
```

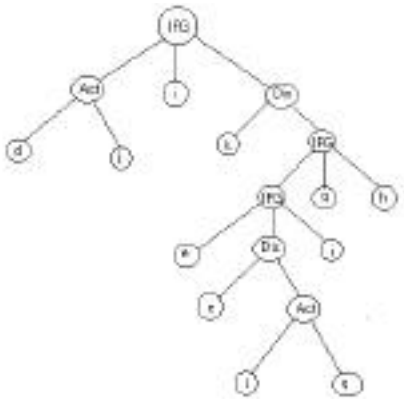

Figure 3. A sample parse tree.

The function Allocate(*x*) allocates *x,* if *x* is a department that hasn't been yet allocated, to the currently available space. A successful allocation of a department *x* returns the value *x*>0, an unsuccessful allocation returns 0.

These set of functions has been selected intuitively. The only way to confirm the validity is experimentally. During the process of evaluation any repetitions are ignored. The evaluation of the tree from Figure 3, for example, will result in sequence: A = { *j, d, k, e, q*}. For each sequence the total cost is calculated according to the formula (1) and this cost is used as a measure of fitness (with this assumption solutions with smaller fitness are better). Solutions such as the one above, which is an example of a sequence that doesn't allocate space for all the departments, remain in the population, however a very high cost is attached to them.

The best solution was obtain in generation 157 with the fitness 2185.5 and the sequence of allocations:

A = {j, k, g, r, h, p, f, n, l, o, m, q, c, a, i, b, d, e, s }

Figure 4 shows the solution graphically. The setting for the genetic program were as follows: population size 1200, tournament selection with a group size of 5, probability of mutation of the terminals 0.02, one point crossover. Figure 5 shows the best solutions developed in the process of evolution.

```
25                  j j       k k       k k       k k
24                  j j       k k       k k       k k
23                  j j       k k       k k       k k
22                  j j j k k k k k k k k k k k k k k
21                  j j j k k k k k k k k k k k k k k

20                  p p       p p       h h       k k
19                  p p       p p       h r       k k
18                  p p       p p       h g       k k
17                  p p       p p       h g       k g
16                  p p p p p p p p p p h g g g g g g
15                  p p p p p p p p p p h g g g g g g
```

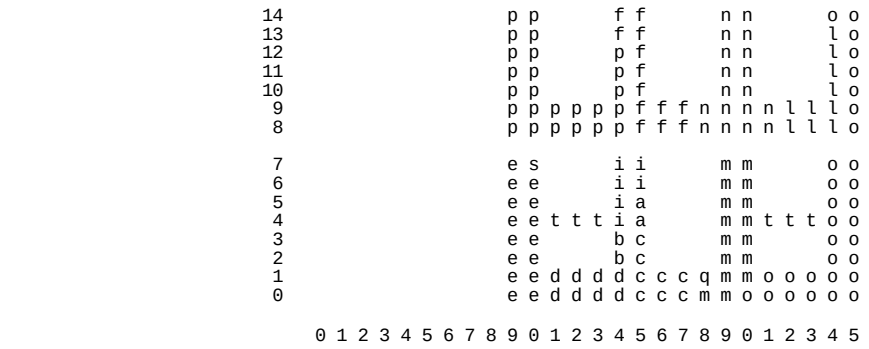

Figure 4. The best solution obtained for the office layout.

These results are better than those obtained previously using a classical approach where the lowest cost obtained was 2405.0 (Liggett 1985, and using genetic algorithms where the lowest cost evolved was 2254.5 (Jo and Gero 1997). These two solutions were re-evaluated by our system, to avoid differences due to small implementation dissimilarities. Direct comparison with Liggett's result has limited value, since we allowed splitting of a department between floors.

Solution	Cost	Generation
e h g o m r s p n j l k f d b i a c q	2346.5	39
e h g o m r s p n j l k q f d b i a c	2330.5	41
e h g o m r p n j l s k q f d b i a c	2322.5	42
e h g o l r s p n j q m k f d b i a c	2321.0	44
j k l r s p f m o n g h c b a i q d e	2302.0	50
j k l r h p f m o n s g c b a i d q e	2288.0	67
j k n r s p f m o l g h c b a i d q e	2280.0	68
j k g r h p f m o n l s c b a i q d e	2261.5	85
j k g r h p f m o n l s c b a i d e q	2249.5	86
j k m r h p f g o l s n c a i b d q e	2241.5	144
j k g r h p f n o l m q c a i b d e s	2192.5	145
j k g r h p f n l o m q c a i b d e s	2185.5	157

Figure 5. The final twelve *best so far* solutions.

5. Discussion

Evolutionary computing can be successfully used for such NP-hard problems as space allocation planning where it shows a clear and demonstrable improvement path. In this paper we have demonstrated that genetic programming can be used to model the space layout planning problem and produce results equal to those obtained by genetic algorithms and gives consistently better results than the best known heuristics from

previously published research. As with many such general solution methodologies there is still considerable skill required in formulating the space layout planning problem appropriately for solution using genetic programming. This is the primary disadvantage of these general solution methods.

The space layout planning problem is an example of problems which belong to the class of NP-hard problems in terms of their complexity. A problem of the size we have solved in this paper is already not trivial. The advantage of the genetic programming approach is that it is largely independent of the size of the search space unlike many other approaches. As the size of the problem gets larger the quality of the solution generated may drop but the robustness of the method is unchanged.

References

Goldberg, D.E. (1989) Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading.

Jo, J. H. and Gero J.S. (1997) Space layout planning using an evolutionary approach, Artificial Intelligence in Engineering, (to appear).

Koza, J. (1992) Genetic Programming, MIT Press, Cambridge, MA.

Liggett, R. S. (1985) Optimal spatial arrangement as a quadratic assignment problem, in J. S. Gero (ed), Design Optimization, Academic Press, New York, pp. 1-40.

Michalewicz, Z. (1994) Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag, Berlin.

Tate, D. M. (1995) A genetic approach to the quadratic assignment problem, Computers Opns Res. **22**, 1, 73-83.

This paper is a copy of: Jagielski, R. and Gero, J. S. (1997). A genetic programming approach to the space layout planning problem, *in* R. Junge (ed.), *CAAD Futures 1997,* Kluwer, Dordrecht, pp. 875-884.