Breeding Normalized Postfix Expressions for the Facility Layout Problem

Christine L. Valenzuela * Pearl Y. Wang †

* Department of Computer Science, Cardiff University
PO Box 916, Cardiff CF24 3XF, United Kingdom
Email: christine@cf.cs.ac.uk

† Department of Computer Science MS4A5, George Mason University
Fairfax, VA 22030-4444, USA
Email: pwang@cs.gmu.edu

1 Introduction

This is a preliminary study in which we use a genetic algorithm (GA) to breed normalized postfix expressions for solving the facility layout problem (FLP). Our technique, initially developed for VLSI floorplanning, simultaneously places rectangles onto a planar site and optimizes area utilization by altering the shapes of facilities that have fixed area but flexible height and width dimensions. We present results for eight small benchmark problems from the literature and obtain improvements on the previously published results for some of the problems. Our method has already proven successful on VLSI problems with many more rectangles than the FLP problems we use in the present study. Identifying suitable test data will enable us to extend to larger problems for the FLP.

1.1 Preamble

The FLP is an NP–Hard combinatorial optimization problem in which a collection of facilities are placed onto a planar site. Each facility has a required area and there is an interconnection cost for each pair of facilities. Traditional methods for solving the FLP include quadratic assignment techniques, integer programming, and graph theoretical methods. (See Kusiak and Heragu [5] for a survey.) More recently, various metaheuristic techniques have been applied, and a variety of layout representations have been developed. One promising representation is the slicing tree structure (STS) due to Otten [8]. STSs provide a method for representing a *guillotine pattern* or *slicing floorplan* which can be constructed by cutting a large rectangle into smaller rectangles using only vertical and horizontal edge–to–edge cuts. STSs were the chosen representation of Kado *et al* [3, 4] for their study on genetic algorithms for the FLP. Garces-Perez *et al* [2] also used STSs for their genetic programming approach to the FLP. Our genetic algorithm is applied to the benchmark problems used by these researchers and a comparative assessment is presented.

2 The Facility Layout Problem

Solving the Facility Layout Problem involves placing the facilities in the Euclidean plane so that they do not overlap with each other; at the same time, an objective function is to be minimized. The objective function below was introduced in Tam and Li [9] and is based on Hooke's Law:

$$\text{minimize } flow = 0.50 \sum_{i<j} w_{i,j} d_{i,j}^2$$

where $w_{i,j}$ is the traffic between facilities i and j, and $d_{i,j}$ is the Euclidean distance between them.

Eight FLP benchmark problems are introduced in [9] and include a variety of flexible and rigid facilities. Details of the smallest of the benchmark data sets, TL91-5, are shown in Table 1. Column two of the table specifies the traffic matrix between the five facilities. For facility i, the j^{th} column of the matrix specifies $w_{i,j}$, the traffic between i and j. Column three gives the area for each of the facilities and the fourth column specifies the range of Height/Width dimensions or *aspect ratios* which are allowed. Finally, the last column indicates that the facilities have *free orientation*, i.e. they can be rotated through 90 deg.

Facility	Traffic	Area	Aspect Ratio Range	Orientation
1	0 5 2 4 1	24	0.80–1.00	Free
2	5 0 3 0 2	16	0.75–1.15	Free
3	2 3 0 0 0	36	0.60–1.85	Free
4	4 0 0 0 5	8	0.30–1.10	Free
5	1 2 0 5 0	21	0.90–1.18	Free

Table 1: TL91-5 Benchmark Data

3 Slicing Trees and Normalized Postfix Expressions

Our postfix expressions utilize the operators '$*$' and '$+$' which represent vertical and horizontal cuts, respectively. The integers $1 \ldots n$ represent the n facilities in a problem. Postfix expressions provide a convenient linear representation system for STSs, and *normalized* postfix expressions have the further advantage of representing each layout uniquely (see Wong, Leong and Liu [12] for further details). Normalized postfix expressions are characterized by strings of alternating '$*$' and '$+$' operators separating the rectangle IDs. Figure 1(a) illustrates a slicing floorplan, its slicing tree representation, and the corresponding normalized postfix expression.

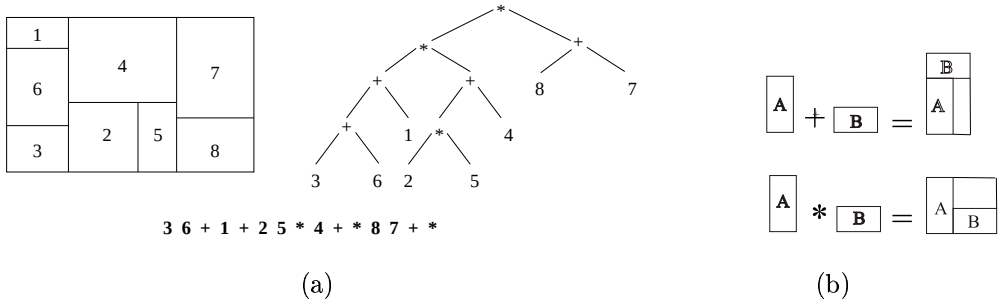

Figure 1: Slicing Trees and Normalized Postfix Strings

From a bottom–up perspective, the slicing tree describes how pairs of rectangles can be combined recursively to yield larger rectangles. Figure 1(b) shows the actions of the binary operations '$+$' and

'$*$' on the two rectangles A and B: '$+$' puts B on top of A and '$*$' puts 'B' on the right of A. Note that wasted space can appear within the enclosing rectangles that result from these binary operations. By repeatedly combining pairs of rectangles together with sub-assemblies of rectangles, a complete layout can be generated. Note that a complete postfix expression of n rectangles will contain exactly $n-1$ operators. Also, at any point during the evaluation of a postfix expression, the cumulative total of operators must be less than the cumulative total of rectangles.

4 The Representation and Decoder

The representation used for our GA is order based and consists of an array of records, with one record for each of the basic rectangles of the data set. Each record contains three fields:

- *a rectangle ID field*: this identifies one of the basic rectangles from the set {1, 2, 3,..., n}
- *an op-type flag*: this boolean flag distinguishes two types of normalized postfix chains, $T = +**+**+\ldots$ and $F = **+**+*\ldots$
- *a chain length field*: this field specifies the maximum length of the operator chain consecutive with the rectangle identified in the first field.

Starting with a given array of records produced by the GA, our decoder will follow Algorithm 1 and produce a legal normalized postfix expression.

1. Examine next (first) record; copy the rectangle ID.
2. Generate a chain of alternating operators of op-type specified in op-type flag. This chain should have length defined in the length field.
3. Copy the operators, in sequence, from the chain generated in 2) until either you get to the end of the chain or more operators would invalidate the expression (see section 3).
4. If there are more records left to process then go to 1) else complete the normalized postfix expression by printing further operators at the end of the postfix string until the number of operators is one less than the total number of rectangles in the expression.

Algorithm 1 Outline of Decoder Algorithm

5 The Genetic Algorithm

The simple genetic algorithm (GA) used here is an example of a steady state GA. It uses the weaker parent replacement strategy first described in [1]. The GA applies the genetic operators to permutations. We employ *cycle crossover, CX* [7] and three different mutations: M1 swaps the position of two rectangles, M2 switches the op-type flag from '$+$' to '$*$' or vice versa, and M3 increments or decrements (with equal probability) the length field.

The objective function of Tam and Li is used as our fitness value and pairs of parents are selected in the following way: the first parent is selected deterministically in sequence, but the second parent is selected in a roulette wheel fashion. The selection probabilities for each genotype are calculated using the formula: $Selection\ Probability = \frac{Rank}{\sum Ranks}$ where the genotypes are ranked according to the values of their objective function with the worst assigned rank 1. Single offspring are produced by crossover, and then a single mutation selected from M1, M2, and M3 is applied.

Once decoded, the postfix expression is evaluated and the corresponding slicing layout generated. As part of the layout generation routine, an area optimization procedure is included which optimizes the shapes of the individual facilities and minimizes the total area of the layout. Our area optimization procedure is an exact computation which, given a particular slicing layout, will guarantee optimal area utilization for that floorplan. Its average case run time is $O(n \log n)$ and the procedure works by storing key height and width dimensions of flexible modules on *shape curves* and then combining the shape curves in pairs during the evaluation of the postfix expression. A range of alternative shapes for individual facilities and a range of possible dimensions for the final layout are produced. The best version of the final layout, i.e. the one having the smallest area is then selected and a backtracking routine invoked to evaluate the dimensions and positions, and hence the *flow* of all the facilities corresponding to the best layout. (See [10, 11] for more details.)

6 Results

Our GA results are presented in Table 2. Column VA01-AVG gives our best flow obtained for the various TL91 data sets averaged over five replicate runs of our GA, and column VA01-BEST gives the best of these five results. These results are compared with the genetic programming results of Garces-Perez *et al* [2] in column GP96 and use the same population sizes that they do as shown in column GP-POP. Column GP-FIRST lists the generation in which the best solution first appeared for the GP96 experiments. The final column of the table, VA01-FIRST (AVG) gives the generation in which the best solution appeared for our experiments, averaged over the five replicate runs. Our GA continued to run for 40 further generations after this. Finally, column BEST shows the previously best published results for these data sets (as revealed by our literature search) with the indicated sources. Better solutions than this are reported in [6] for TL91-8, TL91-12 and TL91-15, but different aspect ratios were used for the facilities, and thus the results are not comparable with our other sources [2, 3, 9]. The best published results for TL91-20 were obtained by Garces-Perez *et al* by post-processing their initial result from GP96 using Kado's STS conversion code. The GA ran on a Pentium III 600 MHz processor and took a few seconds for the smaller problems and up to 7 minutes for the largest problem.

PROBLEM	GP96	GP-FIRST	GP-POP	BEST	VA01-BEST	VA01-AVG	VA01-FIRST (AVG)
TL91-5	226	24	600	226[2]	217	217	1
TL91-6	384	33	600	361[3]	368	371	11
TL91-7	568	175	600	559[9]	564	570	26
TL91-8	878	1,657	1,500	839[9]	821	828	36
TL91-12	3,220	168	1,500	3,162[9]	3,015	3,084	96
TL91-15	7,510	1,383	1,500	5,862[9]	7,059	7324	90
TL91-20	14,033	2,521	1,500	14,026*	14,167	14,609	133
TL91-30	39,018	2,933	1,500	39,018[2]	37,181	38,402	181

Table 2: Flows for the TL91 Benchmark Problems

Overall, our results improve those presented by Garces-Perez and were obtained more cheaply (see the FIRST and VA01-FIRST (AVG) columns). For four of the eight problems, we beat the previously best published results taken from a number of sources. As an example, the layout which our GA obtained for the TL91-5 benchmark described earlier is given in Table 3. Columns XC and YC list the midpoint coordinates of the facilities and the other columns list size and area information about the facilities. The layout is shown in Figure 2.

Facility	XC	YC	Width	Height	Area	Width/Height	Height/Width
1	2.19089	7.1499	4.38178	5.47723	24	0.8	1.25
2	6.69118	6.14334	4.6188	3.4641	16	1.33333	0.75
3	4.08044	2.20564	8.16088	4.41129	36	1.85	0.540541
4	5.18532	10.3644	1.60709	4.97795	8	0.322841	3.0975
5	8.09817	10.3644	4.2186	4.97795	21	0.847458	1.18

Table 3: Facility Layout for the TL91-5 Benchmark Problem

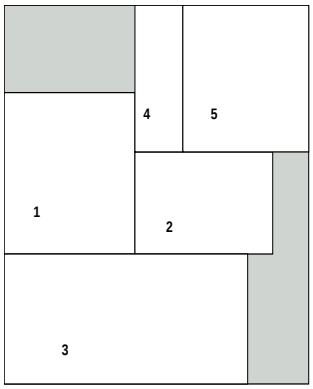

Figure 2: Facility Layout for TL91-5: width = 10.2, height = 12.9

References

[1] D.J. Cavicchio. *Adaptive Search Using Simulated Evolution.* PhD dissertation, University of Michigan, Ann Arbor, 1970.

[2] Jaime Garces-Perez, Dale A. Schoenefeld, and Roger L. Wainwright. Solving facility layout problems using genetic programming. In *Proceedings 1st Annual Conference in Genetic Programming*, pages 182–190, 1996.

[3] K. Kado. An investigation of genetic algorithms for facility layout problems. University of Edinburgh, MSc Dissertation, 1995.

[4] Kazuhiro Kado, Peter Ross, and David Corne. A study of genetic algorithm hybrids for facility layout problems. In *Proceedings 6th International Conference on Genetic Algorithms*, pages 498–505, 1995.

[5] A. Kusiac and S. Heragu. The facility layout problem. *European J. Oper. Res.*, 29:229–251, 1987.

[6] T.A Lacksonen. Preprocessing for static and dynamic facility layout problems. *International Journal of Production Research*, 35(4):1095–1106, 1997.

[7] I.M. Oliver, D.J. Smith, and J.R.C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 224–230, 1987.

[8] R.H.J.M. Otten. Automatic floorplan design. In *Proceedings of the 19th ACM-IEEE Design Automation Conference*, pages 261–267, 1982.

[9] K.Y. Tam and S.G. Li. A hierarchical approach to the facility layout problem. *International Journal of Production Research*, 29(1):165–184, 1991.

[10] C. L. Valenzuela and P.Y. Wang. VLSI Placement and Area Optimization Using a Genetic Algorithm to Breed Normalized Postfix Expressions. Under review.

[11] C. L. Valenzuela and P.Y. Wang. A Genetic Algorithm for VLSI Floorplanning. In *Parallel Problem Solving from Nature – PPSN VI*, Lecture Notes in Computer Science 1917, pages 671–680, 2000.

[12] D.F. Wong, H.W. Leong, and C.L. Liu. *Simulated Annealing for VLSI Design.* Kluwer Academic Publishers, Norwell, Massachusetts, 1988.